eXtreme Programming in der Praxis - das Sentinet-Projekt

Informatikprojekt
der Philosophisch-naturwissenschaftlichen Fakultät der Universität Bern

vorgelegt von

Beat Halter, Mauricio Seeberger,
Susanne Wenger, Vivian Kilchherr

2002

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Michele Lanza

Software Composition Group,
Institut für Informatik und angewandte Mathematik,
Universität Bern

Zusammenfassung

Wir machten in den Frühlingsferien beziehungsweise während des Sommersemesters 2002 unser Projekt zusammen mit dem Bundesamt für Gesundheit. Unsere Aufgabe war eine Applikation mit generischen Eingabeformularen zu schreiben, welche den Austausch von Konsultationsdaten zwischen verschiedenen Ärzten und dem Bundesamt ermöglicht. Jedoch lag dabei unser Hauptinteresse nicht nur bei der Implementation, sondern es sollte ein richtiges Extreme Programming (XP) Projekt werden. So hatten wir wöchentliche Iterationen sowie Planning Games und wendeten auch für Design und Entwicklung XP-Techniken an. Es stellte sich heraus, dass viele der XP-Regeln und Techniken Sinn machen - andere jedoch weniger. In dem folgenden Dokument wird zuerst unser Projekt genau vorgestellt, dann kommen Kapitel über die Projektorganisation, Projektmanagement, Iteration und Entwicklung. Im letzten Kapitel diskutieren wir, wie sich XP auf unser Projekt ausgewirkt hat.

Die Kapitel haben wir dabei meist in zwei Teile geteilt: zuerst einen allgemeinen Teil mit Theorie und Informationen und danach einen Teil, wo es um unsere Erfahrungen geht. Am Ende jedes Kapitels versuchen wir schliesslich unseren Gesamteindruck nochmals kurz zusammenzufassen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

Im Rahmen eines Informatikprojektes für Hauptfachstudierende konnten wir in der Software Composition Group (SCG) der Universtität Bern eine Applikation entwickeln. Aufgrund unserer guten Erfahrungen im Praktikum Software Engineering (PSE) mit Extreme Programming (XP), wollten wir wiederum etwas mit dieser Entwicklungsmethode machen. Da wir aber auch dieser Methode kritisch gegenüberstehen, versuchten wir diese so intensiv wie möglich anzuwenden, um am Ende Vor- und Nachteile von XP herauskristalisieren zu können.

Im Bundesamt für Gesundheit (BAG) fanden wir einen Kunden, der bereit war, ein Projekt im XP-Stil durchzuführen. Allerdings hatten sie noch keine Erfahrungen damit, weshalb sie sich zuerst die Theorie aneignen mussten.

In diesem Dokument diskutieren wir die verschiedenen Aspekte des Extreme Programming im Zusammenhang mit unserem Projekt. Dabei wechseln wir zwischen normaler und kursiver Schrift. Normalgeschriebenes handelt von technischen oder theoretischen Aspekten eines Kapitels und wie wir diese in unserem Projekt angewendet haben. Kursiv spiegelt unsere Erfahrungen und Meinungen wieder. Jedes Kapitel wird mit einem Fazit abgeschlossen.

Zu diesem Dokument gehört eine CD mit weiteren Projektdaten und Dokumentationen.

1.1 Inhalt

Zuerst wird erklärt, was das Projekt **Sentinet** ist und was das BAG darunter versteht. Die Anforderungen an uns stehen in den Requirements[1]. Um ein Projekt erfolgreich durchführen zu können, braucht es eine **Projektorganisation** und ein **Projektmanagement**. Zur Organisation gehören Rollenveteilung und Infrastruktur. Das Management steht für Tracking, Report und Analyse vom laufenden Projekt.

Ein XP-Projekt wird in **Iterationen** unterteilt. Im gleichnamigen Kapitel schauen wir uns die Gliederung einer Iteration genauer an. Danach kommen Extreme Programming spezifische Aspekte für die **Entwicklung** wie Design, Entwicklungsmethodologien und Software-Validierung.

Abschliessend geben wir im Kapitel **Konklusionen** ein Gesamtfazit über das Projekt, insbesondere im Hinblick auf Extreme Programming.

[1] Requirements.pdf (auf CD)

1.2 Übersicht

Sentinet Das Kapitel beschreibt die Idee des Projektes Sentinet und die Anforderungen, die der Kunde an uns stellte. Es wird lediglich ein Grobeindruck vermittelt. Details kann man in den Requirements nachlesen.

Projektorganisation Was muss man bei der Organisation eines XP-Projektes alles beachten? Die wichtigsten Punkte sind Infrastruktur und Rollenverteilung.

Projektmanagement Das Management regelt den Ablauf eines Projektes. Statusreporte und Analysen (Risikoanalyse, Iterationsanalyse) sind ebenso wichtig wie das Tracking.

Iteration Beim Extreme Programming wird ein Projekt in Iterationen aufgeteilt, die alle einen bestimmten Aufbau haben. Eine Iteraton geht von Planning Game, Iterationsplan bis zu Prototypen und Demo.

Entwicklung Das Besondere beim Extreme Programming sind die Methodologien bei der Entwicklung. Diese umfassen die Art wie man ein Design macht, regelt die Entwicklung der Software und nennt Methoden für die Softwarevalidierung.

Conclusion Das Gesamtfazit über das Projekt im Hinblick auf Extreme Programming. Hat es sich gelohnt? Was sind positive Aspekte von Extreme Programming und was hat die Projektentwicklung eher behindert?

Anhang Eine Abbildung des ersten und letzten UML-Diagramms zum vergleichen. Danach werden technische Einblicke in unsere wichtigsten Prototypen vermittelt. Zuletzt sind alle in diesem Dokument gängigen Abkürzungen aufgeführt.

Kapitel 2

Sentinet

Im BAG steht Sentinet für ein Projekt, welches Kontakte zwischen Ärzten und Patienten erfassen soll. Das Ziel ist, ungefähr 200 Ärzte zu überzeugen, welche unentgeltlich regelmässig Konsultationsdaten ihrer Patienten ans BAG senden. Das BAG gibt den Ärzten eine Liste mit Themen, welches elektronische Formulare über zum Beispiel Röteln oder Masern sind. Die Ärzte füllen für jeden Patienten ein Thema aus, falls der Fall des Patienten einem Thema zugeordnet werden kann. Dies ergibt jeden Tag ein Dossier mit ausgefüllten Themen.

Das BAG kann dann die Daten der Ärzte auswerten. Aus den Themen kann zum Beispiel gelesen werden, dass bei einem Arzt von 50 Patienten, 20 wegen Grippe mit Antibiotika behandelt wurden und die restlichen aus anderen Gründen in der Praxis waren. So kann das BAG das vermehrte Auftreten einer Krankheit erkennen und darauf gegebenenfalls reagieren.

Die Aufgaben der Applikation werden im folgenden Abschnitt über die Requirements erläutert.

2.1 Requirements

Das BAG besitzt eine Applikation, die den Anforderungen des Projekts Sentinet genügt. Diese entspricht jedoch nicht den Wünschen des BAG, weshalb wir den Auftrag erhielten, eine Applikation zu entwickeln, die kundenfreundlicher ist als die bestehende. Die Requirements[1] wurden anhand der Erfahrungen aus der bestehenden Applikation erstellt.

Das Programm funktioniert in den Schritten, welche in Abbildung 2.1 dargestellt werden:

1. Der Client überprüft, ob sich auf dem Server andere Daten über das Aussehen des Graphical User Interfaces (GUI) befinden, als er lokal gespeichert hat.

2. Der Server schickt gegebenenfalls diese Daten, welche der Client dann bei sich abspeichert.

3. Anhand dieser Daten wird das GUI generiert. Der Arzt kann die Konsultationsdaten seiner Patienten eingeben. Beim Ausfüllen eines Themas, muss laufend überprüft werden, ob die Eingabe möglich ist, so kann ein Patient zum Beispiel nicht über 42 Grad Fieber haben. Zudem können Eingaben des Arztes andere Eingabefelder einschränken oder gar überflüssig machen.

4. Die neuen Daten werden in einer lokalen XML-Datei abgespeichert.

5. Per Mausklick werden die gespeicherten Daten mit dem Server abgeglichen. Dabei werden die geänderte Daten zum Server hochgeladen.

[1] Requirements.pdf (auf CD)

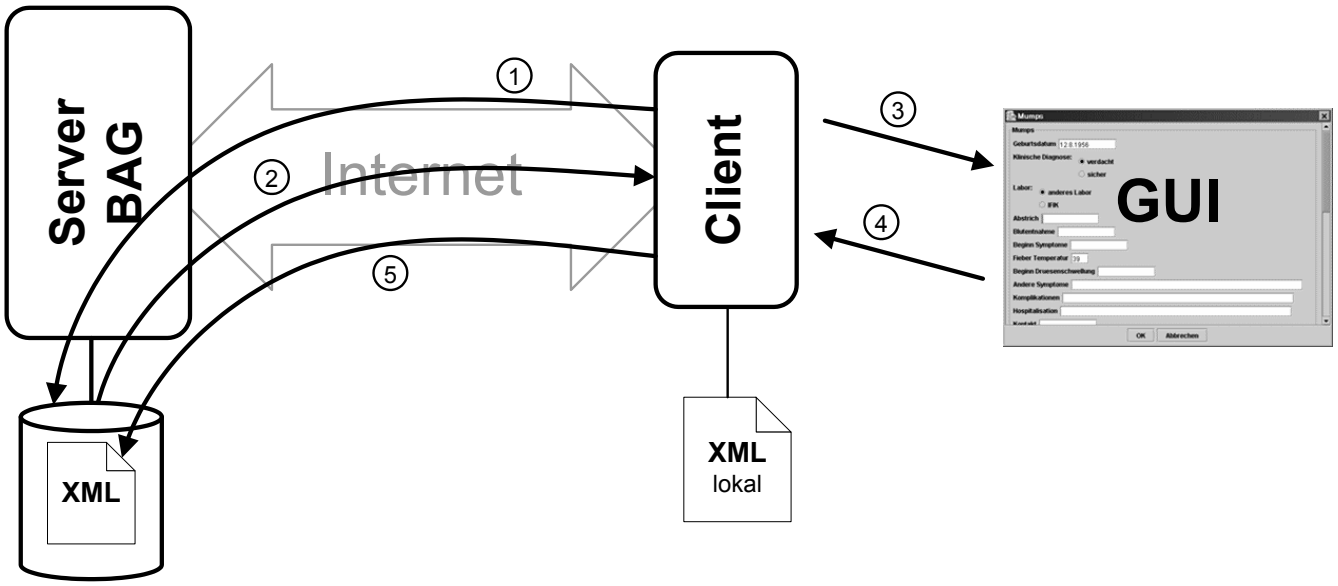

Abbildung 2.1: Einfaches Usecase Diagramm

Das BAG machte sich zu Nutzen, dass bereits eine Applikation bestand und sie deswegen wussten, was sie anders haben wollten und was bleiben sollte. Dies war einerseits von Vorteil, da die Requirements sehr detailliert waren, andererseits schränkte es uns ein, da auch die funktionellen Anforderungen klar gegeben waren und wir neue Ideen nur schwer einbringen konnten.

Aus den Requirements war leider nicht gut ersichtlich, was das Programm genau können musste, weshalb viele Punkte in Sitzungen noch diskutiert werden mussten und auch dann noch gelegentlich falsch verstanden wurden. Dies raubte viel Zeit.

Leider konnte uns das BAG nicht von Beginn weg die nötigen Requirements zur Verfügung stellen. Weshalb wir die ersten Wochen nur mündlich erfuhren, wie das Programm am Ende genau aussehen musste. Wenn wir nach dem Wasserfallprinzip entwickelt hätten, hätte dies Konsequenzen für das ganze Projekt gehabt. Da man ohne Requirements nicht mit Entwickeln anfangen kann.

Im flexibleren XP genügte es zu wissen, wie die ersten Schritte des Projektes aussehen. Deshalb erstellten wir in dieser Zeit wichtige Prototypen (siehe Abschnitt 5.3 Prototyping). So waren die Auswirkungen des Fehlens der Requirements in unserem Projekt nicht von entscheidender Bedeutung. Dies zeigte uns, dass XP gut geeignet ist, falls nur mangelhafte oder noch nicht fertiggestellte Requirements vorhanden sind.

2.2 Fazit

Wir haben mit Sentinet ein vielseitiges Projekt entwickelt, welches einige Knacknüsse wie sichere Datenverbindung mit Zertifikaten oder ein generisches GUI besass. Zudem hatten wir zum ersten Mal einen echten Kunden, was die Sache interessanter machte.

Leider haben wir nicht stark genug darauf beharrt, dass Änderungen der Requirements nicht nur mündlich geschehen sollten, sondern vor allem auch schriftlich. Dies hätte uns einige Unklarheiten erspart, da in den Requirements - nicht in mündlichen Bemerkungen - der verbindliche Projektauftrag definiert werden sollte.

Kapitel 3

Projektorganisation

Die Projektorganisation stellt sicher, dass die passende Infrastruktur wie Arbeitsumgebung und die nötigen Hilfsmittel zur Verfügung stehen. Des Weiteren spielt auch die Rollenverteilung eine wichtige Rolle. Unklarheiten bezüglich Projektorganisation stellen ein grosses Risiko dar, welches den Terminplan und auch die Realisierung gefährden.

3.1 Infrastruktur

Die Infrastruktur beinhaltet folgende 2 Punkte:

1. Arbeitsplatz definiert das Arbeitsumfeld, welches zur Entwicklung bereitsteht. Dies umfasst unter anderem geeignete PCs/Workstations in einer ähnlichen Testumgebung und manchmal auch in einem entsprechenden Testnetzwerk wie das Programm am Schluss laufen sollte.

2. Tools sind die verschiedenen Hilfsmittel, die das Entwickeln vereinfachen, beziehungsweise erst ermöglichen.

3.1.1 Arbeitsplatz

Grundsätzlich sollte man immer in der gleichen Umgebung entwickeln, auf welcher das Programm am Ende laufen sollte. So ist es möglich zu testen, ob alles wie erwünscht funktioniert oder ob ein wichtiger Faktor vergessen worden ist. Dies ist auch der Grund, weshalb immer empfohlen wird, auf der Zielplattform in einem dem Zielnetzwerk entsprechenden Netz zu arbeiten - auch mit der plattformunabhängigen Sprache Java sind solche Faktoren nicht immer auszuschliessen.

Im BAG stand uns kein Arbeitsplatz zur Verfügung. Häufige Demos sicherten deshalb ein reibungsloses Laufen der Applikation im Zielnetzwerk. Das Programm musste nach Requirements unter Windows und Mac OS laufen. Jedoch hatte niemand von uns einen Mac zum Testen und die verwendete Java-Version war für Mac OS auch noch nicht veröffentlicht. Das Problem, dass wir unser Programm nicht auf dem Mac testen konnten, lösten wir, indem wir Kontakt mit Kollegen aufnahmen, die bereit waren unser Programm auf ihrem Mac zu testen. Sonst jedoch arbeiteten wir zu Hause auf unseren Windows-PCs.

Eine Folge davon, dass wir zu Hause entwickeln mussten und keinen Arbeitsplatz vom BAG zur Verfügung gestellt bekamen, wurde beim ersten Demo ersichtlich. Wir hatten nämlich eine Story verkauft, wo wir mit Hypertext Transfer Protocol (HTTP) Dateien über das Netz holen mussten, wobei das Ganze mit Java Web Start [1] laufen sollte. Bei uns zu Hause lief alles, doch beim Demo im BAG

ging plötzlich nichts mehr. Da niemand von uns zuvor mit Java Web Start gearbeitet hatte, nahmen wir an, dass das Laden unseres Programms durch die Firewall verhindert wurde. Jedoch fanden wir dann fast zufälligerweise heraus, dass alle Verbindungen über einen Proxy gehen mussten. Von einem automatischen Erkennen von Proxies und Firewalls war aber nie die Rede gewesen, so dass wir diesen Punkt nicht berücksichtigten. Diesen Faktor kannten wir nicht vom Anfang an und kam erst im Verlaufe des Demos hervor. Es ist natürlich schwierig zu entwickeln, wenn man immer erst jede zweite Woche während den Demos neue, wichtige und noch bisher unbekannte Komponenten entdeckt.

Dass wir keinen gemeinsamen Arbeitsplatz hatten, erschwerte zudem die Kommunikation bei der Entwicklung. Wenn nun zwei Teams am Arbeiten waren, konnte es trotz Concurrent Versions System (`CVS`) [2] leicht vorkommen, dass im gleichen Moment beide Teams gerade die gleichen Klassen änderten. Mit `ICQ` [3] konnten wir uns abzusprechen und zum Teil diskutierten wir so sogar Design-Fragen und sonstige Probleme. Daneben ist ein gemeinsamer Arbeitsplatz auch insofern hilfreich, dass man fixe Arbeitszeiten hat und nicht so häufig abgelenkt wird.

3.1.2 Tools

Wenn die passenden Hilfsmittel nicht vorhanden sind, kann dies ein ganzes Projekt gefährden. Ohne CVS können mehrere Teams nicht gleichzeitig programmieren, ohne `SWiki` [4] braucht es einen höheren administrativen Aufwand und ohne `Apache Ant` [5] wird das Kompilieren des Sourcecodes mühsam. Folglich dauert alles länger als mit den passenden Hilfsmitteln.

Auf einem Account beim Institut für Informatik und Angewandte Mathematik (IAM) befand sich unser CVS-Repository, worauf wir über eine Secure Shell (SSH) Verbindung zugreifen mussten. Als Client setzten wir `WinCVS` [6] ein. Das SWiki brauchten wir für das Projektmanagement sowie zur Dokumentation. Beim Programmieren benutzten wir Java Developement Kit `JDK 1.4` [7] und Apache Ant.

Wir hatten am Anfang viele Probleme, diese nötigen Hilfsmittel bereitzustellen. Für uns war zuerst kein CVS-Server erhältlich. Der normalerweise dafür zur Verfügung stehende `pserver` des SCG stand für uns nicht zur Verfügung. Jedoch kann ein Team von vier Personen nicht effizient entwickeln ohne CVS. Eine Möglichkeit wäre schon, dass man die ganze Zeit Emails mit dem Sourcecode an alle anderen schickt. Doch mit der Grösse des Projektes führt dies mit der Zeit unweigerlich in ein Chaos, zumal wir auch gleichzeitig an verschiedenen Orten programmierten. Mit der jetzigen Lösung können wir auf das CVS-Repository über SSH zugreifen, was uns zu einer speziellen Konfiguration von `WinCVS` zwang (siehe „WinCVS mit SSH einrichten"[1]). Der damit verbundene Arbeitsaufwand von mindestens drei Tagen war nötig, weil wir sonst unser Projekt hätten abbrechen müssen.

3.2 Rollenverteilung

Auf der ExtremeProgrammingRoadmap-Wikiseite [8] kann man nachlesen wie im Idealfall eine Rollenverteilung[2] aussehen kann. Wichtig sind folgende Rollen:

Customer schreibt UserStories und spezifiziert FunctionalTests. Der Kunde kann Prioritäten setzten und Stories erklären. Er hat die Autorität bezüglich den Stories zu entscheiden, obwohl er nicht unbedingt Endanwender sein muss.

[1] swiki/26.html (auf CD)

[2] ressources/ExtremeRoles.htm (auf CD)

Programmer schätzt Stories, definiert die verschiedenen Subtasks der Stories und implementiert die verkauften Stories und die dazugehörigen UnitTests.

Tester implementiert FunctionalTests und stellt bei Nichterfüllen der Tests sicher, dass die Programmierer davon wissen. Dies hat nichts mit UnitTests zu tun sondern betrifft nur reine funktionale Requirements.

Tracker stellt ein- bis zweimal pro Woche den Fortschritt jedes Programmierers fest. Seine Verantwortung ist es, Massnahmen einzuleiten, wenn sich herausstellt, dass etwas nicht in der Zeit sichergestellt werden kann. Zudem schlägt er Design-Sitzungen vor und organisiert Meetings mit dem Kunden.

Coach überwacht alles und stellt sicher, dass das Projekt wie geplant abläuft.

Manager ist zuständig für Meetings jedoch nicht für den Projektablauf. Er organisiert Sitzungen zum Beispiel für das Erstellen des Iterationsplans oder des CommitmentSchedule. Er führt diese Treffen auch durch und informiert über den Inhalt und Beschlüsse.

Customer	Andreas Birrer, Christian Valentin
Programmer	Beat Halter, Mauricio Seeberger, Susanne Wenger, Vivian Kilchherr
Tester	Beat Halter
Tracker	Susanne Wenger
Coach	Susanne Wenger
Manager	Beat Halter, Mauricio Seeberger, Susanne Wenger, Vivian Kilchherr

Tabelle 3.1: Rollenverteilung

Unser Kunde hatte noch keine Erfahrung mit Extreme Programming. So mussten wir zum Beispiel zuerst Überzeugungsarbeit leisten, dass er überhaupt die Requirements schreibt. Das BAG war sich gewohnt, nach der konventionellen Methode einfach zu sagen, was sie wollen und dann dies nach einiger Zeit auch mehr oder weniger zu erhalten. Wir jedoch erwarteten eigentlich ausführliche, schriftliche Requirements. Laut BAG würden sie uns aber die Arbeit wegnehmen, da sie quasi eine Dokumentation zu dem Sentinet schreiben müssten, was nach ihnen unsere Aufgabe sei. Somit war die ganze Rollenverteilung zwischen Kunde und Team nicht geregelt. Deshalb mussten wir lange ohne Requirements arbeiten - mit der Folge, dass es ab und zu ein paar Missverständnisse bezüglich Anforderungen gab. Es musste alles mündlich vermittelt werden und da können schnell wichtige Informationen vergessen werden. Jedoch lief es sonst gut und das BAG wuchs im Verlaufe des Projekts in seine Customer-Rolle hinein. Wir hatten meistens jede zweite Woche ein Demo und ein Planning Game, also eigentlich Iterationen von nur zwei Wochen. So stellten wir sicher, dass wir die Anforderungen erfüllten.

Tabelle 3.1 zeigt die Rollenverteilung im Idealfall, was ziemlich schwierig ist, in der Wirklichkeit umzusetzen. Wir mussten zum Teil davon abweichen. Jede Person hatte zwar eine Verantwortung, doch eigentlich musste sie nur schauen, dass ihre Aufgabe von jemanden in der entsprechenden Iteration erfüllt wird. In unserer Gruppe wollten zudem alle programmieren und wir machten auch ab, dass jeder den ganzen Code kennen sollte. Eine Folge davon war, dass schliesslich alle alles machen mussten. Jedoch wäre es sehr gut gewesen, wenn sich jemand nur um das Administrative wie Tracking usw. gekümmert hätte. Dies erfordert aber eine Hierarchie, in der nicht alle Gruppenmitglieder bei allem mitbestimmen können. Bei uns, in einer gleichberechtigten Gruppe, war dies nicht

zu verwirklichen.

Ein weiterer Ansatz wäre gewesen, wenn wir das ganze Projekt in Teilmodule unterteilt hätten. Diese Teilmodule wären dann durch Interfaces genau definiert worden und jeder hätte sein Modul gehabt. So wie wir es jetzt gemacht haben, kam es häufig zur Situation, dass schliesslich doch nur eine Teilgruppe genauer ein Teilmodul kannte und dieses dann den anderen erklärt werden musste, ohne dass diese es anwenden konnten. So ging das Wissen schnell wieder verloren.

3.3 Fazit

In unserem Projekt kam bald zum Ausdruck, dass eine bessere Rollenverteilung wünschenswert gewesen wäre. In PSE gab es in der Gruppe immerhin manchmal Nebenfachstudenten, die sich gerne auf Administratives beschränkten, weil sie Informatik nicht so wichtig nahmen. Jedoch bei uns musste am Schluss irgendwer diese Aufgabe übernehmen. Es zeigt sich, dass XP nicht für homogene Gruppen geeignet ist, da spezielle Funktionen nicht ideal oder gar nicht besetzt werden können.

Eigentlich hätten wir die Rollenverteilung schon vor Projektbeginn regeln sollen - nicht nur teamintern. So sollten die Requirements schon bei Projektbeginn vollständig fertig vorliegen. Da diese nebenläufig zum Projekt erstellt wurden und zum Teil Punkte nur mündlich vermittelt wurden, waren sie nie ganz klar. Manche Punkte gingen auch vergessen.

Kapitel 4

Projektmanagement

Das Projektmanagement dient dazu, den Ablauf des Projektes zu regeln und Probleme bzw. Risiken zu minimieren. Dafür stehen unter anderem die in den folgenden Unterkapiteln erklärten Hilfsmittel zur Verfügung: Tracking, Iterationsanalyse, Statusreport, Risikoanalyse. Beispiele findet man auf der beiliegenden CD[1] oder im Projekthandbuch[2].

4.1 Tracking

Tracking ist ein mächtiges Hilfsmittel, um den Zeitaufwand und die Effizienz jedes Teammitglieds abzuklären. Der Hauptzweck ist aber, zu erkennen wie weit die einzelnen Tasks schon sind und wie man im Zeitplan steht. Wir haben das ganze Projektmanagement, unter anderem auch das Tracking, im SWiki gemacht. Wir verwendeten dazu die Formatvorlage von Tabelle 4.1:

Beginn-Ende	Person	Beschreibung	Story-Nr.	Status	Aufwand
00.00-00.30	name	HTTP-get mit Zertifikatsprüfung impl.	1	23	30
...					

Tabelle 4.1: Template für Tracking

Das Ganze wird nach Tag und Zeit geordnet, damit der aktuelle Status abgelesen werden kann. Nachdem wir die verschiedenen Stories verkauft hatten, erstellten wir einen Iterationsplan mit Story-Nummern. Im Tracking wurde unter Story-Nr. die Nummer des Milestones eingetragen und unter Beschreibung kamen dann noch wichtige Bemerkungen dazu, wie zum Beispiel mögliche Probleme bei der Implementation.

Ein weiterer möglicher Punkt ist, dass man nebst Aufwand und Status noch eine Schätzung über den Aufwand bis zur Fertigstellung der Milestones macht. Dies ermöglicht bei kritischen, verzögerten Subtasks weitere Programmierer mit diesem Subtask zu beauftragen oder Design-Sitzungen zu planen, wo Wege zur Lösung von Problemen dieses Subtasks ergriffen werden können. Des Weiteren kann der Kunde rechtzeitig informiert werden, so dass eventuell die kritische Story aus dem Scope genommen werden kann.

Beim Erstellen des Iterationsplans wurde darauf geachtet, dass die einzelnen Stories in Subtasks mit eigenem Zeitbudget unterteilt werden konnten. Diese Subtasks wurden durchnummeriert beginnend mit 1.a (Story 1, Milestone a). Nun stimmte leider unsere Abschätzung häufig nicht,

[1] swiki/11.html (auf CD)

[2] ressources/projektHandbuch/index.html (auf CD)

beziehungsweise wir vergassen zum Teil sogar einzelne Subtasks. Durch die Unterteilung beim Tracking zwischen Story-Nummer und Beschreibung, konnten wir immer genau abschätzen, wie weit wir waren. Falls nun eine Gruppe einen Subtask von einer anderen Gruppe weiterführt, sieht sie, was für Probleme es gibt und wie weit die andere Gruppe gekommen ist.

In unserem Team war die Motivation fürs Tracking eher klein. Eine Woche nach der Arbeit kann man halt einfach nicht mehr genau einschätzen, was wie viel Zeit beanspruchte. Jedoch ist es zugegebenermassen auch wiederum mühsam, immer nach der Arbeit, wenn man sich über Erfolg freut beziehungsweise über Misserfolg aufregt, noch gerade den Arbeitsaufwand und Status einzutragen. Das Tracking sollte man jedoch immer während des Programmierens machen. Dies ergibt zum einen den kleineren Aufwand und zum anderen verhindert, dass man nicht vergisst, was man genau gemacht hat. Deshalb mussten wir uns dazu entschliessen, jeden Mittwoch und Samstag eine Art Deadline für das aktuelle Tracking zu setzen. So konnten der Statusreport am Sonntag rechtzeitig erstellt werden und die Daten waren noch einigermassen genau. Zusammenfassend war das Tracking ein heikles Thema. Ein nachlässiges Tracking kann man nämlich auch als Desinteresse am ganzen Projekt verstehen. Eine Lösung für die obigen Probleme wäre ein externes Coaching.

4.2 Iterationsanalyse

Die Iterationsanalyse dient dazu Fehler und sonstige Probleme der letzten Iteration aufzudecken und zu analysieren. Das Ziel dabei ist, diese Fehler dann bei der nächsten Iteration nicht mehr zu machen. Dabei muss man sich auch zur teaminternen Kommunikation oder zum Schätzen der Stories Gedanken machen.

Folgende Fragen gehören unter anderem zur Iterationsanalyse[3]:

1. Waren die Stories klar? Zu gross? Zu klein? Zu wenig gut erklärt?

2. Warum haben gegebenenfalls die Abschätzungen der Stories nicht funktioniert?

3. Wie läuft es mit der gruppeninternen Kommunikation? Was kann getan werden, um sie zu verbessern?

4. Wie wurden Abhängigkeiten zwischen den einzelnen Tasks gelöst? Wie kann man die Arbeit besser planen (oder mit der Dummy-Technik arbeiten), so dass niemand untätig herumsitzen muss, während er auf ein Modul wartet?

5. Wer hat wie viel gearbeitet (in Stunden)? Wie kann die Arbeitsbelastung besser verteilt werden?

6. Wie wurden die fachlichen Aufgaben (in Prozent) an einzelne Teammitglieder verteilt? Gab es einen Engpass weil nicht genug Leute die richtigen Fähigkeiten hatten? Wie kann die interne Ausbildung verbessert werden?

7. Alle Programmierer sollen ihren Aufwand bezüglich Implementation von Funktionalität, Tests, Administratives und Lernen von neuen Technologien usw. analysieren. Wie viel Zeit (in Prozent) wurde für bestimmte Kategorien aufgewendet? Welches Fazit wird gezogen?

[3] ressources/projektHandbuch/midterm.html (auf CD)

Die ersten zwei Iterationsanalysen machten wir noch in einer Teamsitzung. Die dritte Iterationsanalyse erledigte eine einzige Person, und danach wurden keine Iterationsanalysen mehr durchgeführt.

Nicht alle Teammitglieder waren an der Iterationsanalyse interessiert. In einem Projekt wie bei unserem geht es vielleicht noch, wenn keine Iterationsanalysen gemacht werden, jedoch ist diese Strategie langfristig schlecht. Der Einfluss der Iterationsanalyse bei nur fünf Iterationen ist eher gering. Bei den ersten zwei Iterationsanalysen haben noch alle mitgewirkt, was auch zu einer besseren Storyabschätzung führte. Denn vor allem das Schätzen der Stories ist Übungssache und beim Programmieren merkt man nicht richtig, wie viel Zeit man tatsächlich für eine gewisse Story investiert hat.

Die eher minimalistische dritte Iterationsanlyse war auch nicht mehr besonders aussagekräftig, da sich der Einfluss von nur einer einzigen Person bemerkbar machte. Viele der oben erwähnten Punkte lassen sich nur im Team besprechen und falls diese Punkte nicht ehrlich und seriös diskutiert werden, bringt die Iterationsanalyse nichts. Inzwischen lagen wir aber mit dem Storyschätzen nicht mehr so viel und häufig daneben, beziehungsweise kalkulierten wir eigentlich schon beim Planning Game ein, dass unsere Schätzung zwischen 50% - 100% danebenliegen und passten deshalb einfach unser Zeitbudget entsprechend an. Des Weiteren war die teaminterne Kommunikation gut, da wir einander schon kannten. Die Iterationsanalyse wäre nämlich auch der richtige Ort, wo man alles, was einen gestört hat, sagen kann und die Kommunikation innerhalb der Gruppe analysiert. Jedenfalls fanden wir gegen Ende keine Zeit mehr für weitere Iterationsanalysen und zudem konnten wir keinen weiteren Nutzen mehr daraus ziehen.

4.3 Statusreport

Der Statusreport ist vor allem für den Kunden gedacht. Er zeigt auf, was für Probleme aufgetaucht sind und wo wir mit der Entwicklung stehen. Beim grossen Statusbericht wird eine ganze Iteration abgeschlossen, während beim kleinen Statusbericht der Zwischenstand wöchentlich beschrieben wird. Speziell wenn nicht so häufig Demos anstehen, zeigt dies dem Kunden, dass man am Arbeiten ist. Auch orientiert ein guter Statusbericht über den aktuellen Status.

Dabei werden mindestens folgende Informationen angegeben:

- Header mit Übersicht mit Projekttitel, Fortlaufende Nummer des Reports, Datum, Projekt-Team, AutorIn des Reports, Aktuelle Phase, Status, Zielerreichung (erledigte Arbeitspakete/ Arbeitspakete in der Iteration gemäss Planung), Wahrscheinlichkeit der Erreichung aller Iterationsziele (in Prozent).
- Offene Fragen, d.h. was man wissen muss, damit weitergearbeitet werden kann.
- Wichtige Informationen, d.h. eine generelle Beschreibung des aktuellen Status.
- Story Tracking, d.h. welche der Stories abgeschlossen beziehungsweise in das nächste Release genommen werden und wie weit die einzelnen Stories sind.
- Wichtige Pendenzen, Top-5-Risiken und nächster Meilenstein.

Auf der CD[4] kann man verschiedene Templates für einen kleinen Statusreport finden. Der

[4] ressources/projektHandbuch/statusReports.html (auf CD)

Statusreport setzt voraus, dass man die Risikoanalyse und das Tracking seriös gemacht hat.

Bei unserem Projekt war ein Statusreport eigentlich nicht nötig, da der Kunde uns machen liess, was wir wollten. So liessen wir gegen Ende ein paar Statusberichte weg. Dies durften wir aber nur, weil wir kurze Iterationen hatten und uns häufig zum Planning Game trafen. So hatte der Kunde immer gerade den aktuellsten Stand direkt von uns. Unsere Hauptarbeitszeit war sowieso das Wochenende. Somit machte ein Statusbericht während der Woche keinen Sinn, da der Entwicklungsverlauf nicht linear war und am Montag trafen wir uns mit dem Kunden. Wenn jedoch seltener Meetings stattgefunden hätten und der Kunde auch etwas für unseren Dienst hätte zahlen müssen, wäre wohl auf einen seriösen Statusreport nicht zu verzichten gewesen.

Der Nutzen eines Statusreports als Projektstatusübersicht ist sicher vorhanden, jedoch ist der Aufwand zum Erstellen des Berichts eher gross - auch wenn ein seriöses Tracking und eine gute Risikoanalyse vorliegt. Im Allgemeinen bringt ein Statusbericht dem Entwicklerteam wohl kaum was - im Gegenteil, es wird sogar wertvolle Zeit dafür abgezweigt. Als Nebeneffekt erhält der Kunde einen besseren Einblick ins Geschehen und verzeiht deshalb auch mal kleine Fehlerchen. Speziell im PSE konnten wir feststellen, wie sehr der erste Eindruck zählt. Abgesehen von den sonstigen Leistungen führte ein guter Statusbericht mit seriöser Risikoanalyse dazu, dass wir mehr Freiheiten beim Programmieren hatten.

4.4 Risikoanalyse

Die Risikoanalyse dient dazu, Risiken einzuschätzen und geeignete Massnahmen dagegen zu ergreifen. Als Risiko gelten alle Faktoren, die das Projekt gefährden oder verzögern. Somit kann die Risikoanalyse dazu führen, dass manche Entscheidungen schon an gewisse Risiken angepasst gefällt werden, weil das Eintreffen dieser Faktoren eigentlich erwartet wird. Der Vorteil dabei ist, dass zum Beispiel nicht plötzlich das ganze Team auf einen Programmierer und sein Teilmodul warten muss. Schliesslich dient die Risikoanalyse dazu, den Ablauf möglichst harmonisch zu gestalten und auszuschliessen, dass nicht jeder Zwischenfall das ganze Projekt kippen kann. Dies führt zu einem gewissen Grad an Stabilität und Zuverlässigkeit.

Wir verwendeten für die Risikoanalyse das Format von Tabelle 4.2.

Priorität	Beschreibung	W'keit	Ausmass	Massnahmen
1	Fehlende Infrastruktur: CVS	50	95	Kontaktaufnahme mit M. Locher, evtl. cvs auf IAM-Account evtl. eigener cvs-Server
...				

Tabelle 4.2: Template für Risikoanalyse

Die Felder haben dabei folgende Bedeutung:

Priorität wie sehr muss das Risiko in der Projektplanung berücksichtigt werden.

Beschreibung detailliert die verschiedenen Risiken und gibt an, unter welchen Bedingungen sie eintreten können.

W'keit die Eintreffenswahrscheinlichkeit (in Prozent) des Risikosereignisses.

Ausmass Da wir in unserem Projekt keine finanziellen Auswirkungen haben, schätzten wir mit einer Zahl zwischen 0 und 100, wie sehr das Eintreten eines Risikoereignis den Projektablauf beeinflussen würde.

Massnahmen zeigt auf, wie man das Eintreten des Risikoereignisses verhindern kann und was beim Eintritt unternommen werden muss.

Unsere Top-Risiken waren fehlende Infrastruktur, keine schriftlichen Requirements und die Zertifikat-Story.

Vor allem das Fehlen eines CVS-Servers machte uns grosses Kopfzerbrechen. Trotz verschiedener Massnahmen verloren wir doch schlussendlich etwa drei Arbeitstage. Dieser Aufwand war nötig, um ein normal funktionierendes cvs zu haben und ohne diesen Aufwand zur Lösungsfindung wäre unser Projekt ernsthaft gefährdet gewesen.

Das Problem mit den Zertifikaten lösten wir mit Prototyping (siehe Abschnitt 5.3 Prototyping). Damit bauten wir kleine Module, um die Machbarkeit abzuklären. Am Anfang war nicht klar, ob wir alles mit Java implementieren konnten und deshalb war dies wichtig. Diese Strategie war dann auch erfolgreich. So hatten wir die einzelnen Module und diese mussten für das Endprodukt nur noch zusammengesetzt werden.

Wie schon im PSE wirkt unsere Risikoanalyse zum Teil etwas gestellt. Zum Beispiel das Risiko, dass ein Mitglied aussteigt, ist zwar vorhanden, aber minimal. So verhält es sich auch mit anderen Risiken. In einem Uniprojekt ist es eben äusserst schwierig, externe Risiken zu simulieren, so dass die ganze Risikoanalyse gestellt wirkt. Immerhin hat sich jedoch an unseren drei Top-Risiken gezeigt, dass die Risikoanalyse doch manchmal etwas bringt.

4.5 Sitzungen und Protokolle

Wir hielten regelmässig Sitzungen. Dabei schrieb im Wechsel eine Person das Protokoll. Dieses wurde im SWiki publiziert[5].
Bei allen Protokollen wurden folgende Punkte angegeben:

- Titel mit Datum, Zeit und Anwesendenliste.
- Traktandenliste.
- Actions und Decisions, d.h. was nun gemacht wird und was wir entschieden haben.

Unser Hauptproblem war, dass unsere Meetings sehr lange dauerten. Wir haben eine Beschränkung der Meetingzeit zwar vereinbart, aber konnten sie nicht umsetzen. Wir kamen zum Schluss, dass eventuell längere Treffen auch gut sind, da nicht nur übers Projekt gesprochen werden konnte, sondern auch andere Punkte diskutiert wurden. Dies führte zu einer entspannten Atmosphäre an den Meetings.

Wir hatten insoweit einen Nutzen aus den Protokollen, dass abwesende Mitglieder nachlesen konnten, was wir besprochen und entschieden hatten. Die Protokolle waren zwar vollständig, aber dennoch manchmal zu wenig ausführlich, so dass man sich trotzdem noch bei einem anwesenden Mitglied informieren musste. Eine wichtige Funktion der Protokolle war zudem, dass man nach einer Weile unklar gewordene Punkte nachlesen konnte. Auch in einigen Jahren kann man noch aus den Protokollen schliessen, was wir wann und wieso genau gemacht haben. Zusammenfassend haben die Protokolle jedoch mehr Arbeit gemacht, als dass sie uns kurzfristig genützt haben.

[5] swiki/4.html (auf CD)

4.6 Fazit

Projektmanagement ist mühsam, wenn man es selber machen muss. Es ist ziemlich zeitintensiv (insgesamt mindestens 10 Stunden pro Woche). Auch ist der Nutzen nicht offensichtlich überwältigend. Besonders ins Gewicht fielen zudem die Gruppensitzungen, wo zwar zum Beispiel Iterationsanalyse geplant war, jedoch wir nicht die ganze Zeit beim Thema blieben und doppelt so viel Zeit wie geplant brauchten. Ein weiteres Problem war beim Projektmanagement, dass nicht alle dazu Zeit hatten, so dass es meistens an einer Person hängen blieb. Nun wissen wir eigentlich immer noch nicht genau, welcher der folgenden Ansätze am besten ist:

- *Wenn das ganze Projektmanagement von einer einzigen Person übernommen wird auf die Gefahr hin, diese damit zu demotivieren,*
- *wenn das ganze Projektmanagement von der ganzen Gruppe immer zusammen gemacht wird. Hier ist die Gefahr von Missmut über sich drückende Mitglieder gross, oder*
- *wenn das Projektmanagement quasi im Wechsel von verschiedenen Mitgliedern beziehungsweise Teilgruppen erledigt wird.*

Bei umfangreichen Projekten, wo ein professioneller Eindruck erweckt werden muss, führt seriöses Projektmanagement zu grösserem Vertrauen beim Kunden. Jedoch sind Uniprojekte zu klein und in einer klar definierten und kontrollierten Umgebung, so dass es ein Zeitverlust fürs Team bedeutet.

Kapitel 5

Iteration

Eine Iteration ist ein Entwicklungszeitraum, welcher mit einem Planning Game beginnt und mit einem Demo endet. Dazwischen entwickelt das Team nach einem Iterationsplan. Dieser definiert Milestones, welche beim Demo alle erfüllt sein müssen. Die Dauer einer Iteration hängt einerseits vom Kunden und andererseits von den Arbeitszeiten der Gruppe ab.

5.1 Planning Game

Mit einem Planning Game sollen die Wünsche des Kunden mit den Möglichkeiten der Programmierer abgestimmt werden. Das Planning Game läuft in den folgenden Schritten ab:

- Der Kunde schreibt Stories.
- Die Entwickler schätzen den Zeitaufwand der Stories.
- Der Kunde wählt Stories im Wert des Zeitbudgets der Gruppe und gibt ihnen Prioritäten.
- Das Team übernimmt den Auftrag.

Beim Planning Game wird also abgemacht, was im nächsten Demo gezeigt werden soll.

Wir entwickelten die Stories gemeinsam mit dem Kunden. Er nannte den Titel und eine Beschreibung und wir notierten diese genauer. Anschliessend schätzten wir die benötigte Zeit für jede Story, indem wir sie in Milestones (vgl. Abschnitt 5.2.1 Milestones) unterteilten. So konnte jeder Milestone einzeln geschätzt werden und das Total ergab dann den ganzen Storyaufwand. Der Kunde konnte anschliessend für ein von uns vorgegebenes Zeitbudget Stunden einkaufen.

Eine Iteration dauerte bei uns anfangs eine oder zwei Wochen (während den Semesterferien) und später vier Wochen (während dem Semester).

Das Planning Game ist eine sehr gute Variante, um sich mit dem Kunden über den Iterationsaufwand zu einigen. Probleme gibt es einzig beim Einschätzen der Stories. Man muss die eigenen Fähigkeiten und die Fähigkeiten der Gruppe gut kennen, um den Aufwand einer Story nicht zu über- oder unterschätzen. Wenn man das Schätzen jedoch ehrlich und konsequent macht, kann man sich viel Stress ersparen, da man tendenziell eher zu wenig Zeit für eine Story einkalkuliert.

5.2 Iterationsplan

Ein Iterationsplan zeigt, was die einzelnen Teammitglieder im Verlaufe der Iteration tun müssen. Er ist in Stories gegliedert. Wir teilten jede im Planning Game vereinbarte Story in Milestones auf. Jedes Gruppenmitglied übernahm die Verantwortung für einen Milestone, dass dieser bis zum Demo fertiggestellt wird.

Im Iterationsplan ist zu erkennen, wie die Gruppe im Zeitplan liegt, da der Soll-Aufwand jedes Milestones im vornherein geschätzt wird und der Ist-Aufwand ständig aktualisiert wird. Welche Story zuerst entwickelt wird, legt der Kunde mit den Prioritäten im Planning Game fest.

Ein Beispiel eines Iterationsplans[1] wird in Tabelle 5.1 gezeigt.

Nr.	Beschreibung	Verantwortung	Soll	Ist	Status
	Story 1: **Proxy**: Aufwand:30h - Priorität:1				
1.a	HTTP mit Proxy	swenger	6h	1h	100%
1.b	Proxy-Server und Port anbieten	mseeberger	3h	2h	100%
1.c	Test am Mi, 14.00 im BAG	alle	3h	3h	100%
1.d	Property-File einrichten Inhalt/Form des Property-Files definieren	mseeberger	8h	3h	100%
1.e	Default-Proxies aus Property-File automatisch erkennen lassen	bhalter	10h	14.5h	70%

Tabelle 5.1: Auszug aus dem Iterationsplan.

Ein Iterationsplan ist ein wirksames Werkzeug zur Kontrolle des Arbeitsablaufs, da auf den ersten Blick ersichtlich ist, wo noch Arbeit vorhanden und wer dafür verantwortlich ist. Verantwortlich für einen Milestone sein heisst nicht, dass man ihn alleine erledigen muss, sondern vielmehr, dass man die anderen frühzeitig darauf aufmerksam macht, wenn der Milestone eventuell nicht fertiggestellt werden kann und Gegenmassnahmen ergriffen werden können wie zum Beispiel die Unterstützung der anderen oder eine Rücksprache mit dem Kunden.

Einziges Problem dabei ist der administrative Aufwand, der damit entsteht, dass man für jede Arbeit am Projekt die Zeit misst und diese beim richtigen Milestone im Iterationsplan eintragen muss.

5.2.1 Milestones

Ein Milestone steht für ein Zwischenziel, welches man erreichen muss, um der Erfüllung der Stories einen Schritt näher zu kommen. Die Milestones sind ein Mass für den Entwicklungsstand der Gruppe. Je mehr Milestones erreicht werden, desto weiter ist die Gruppe in der Iteration.

Durch die Aufteilung der Stories ergaben sich pro Iteration ca. 25 Milestones. Wenn alle Milestones erfüllt wurden, war das Iterationsziel erreicht.

Mit den Milestones konnten wir jederzeit auf einfache Weise feststellen, wie weit wir mit der aktuellen Iteration fortgeschritten waren. Durch die Aufteilung der Stories wird aus einem grossen, unübersichtlichen Task ein einfacher, überschaubarer Subtask, welcher unabhängig von den anderen Milestones gelöst werden kann. Das Erreichen eines Milestones bedeutet immer, dass man dem Iterationsziel einen Schritt näher gerückt ist.

[1] swiki/40.html (auf CD)

5.3 Prototyping

Prototyping wird durchgeführt um entweder die Requirements zu validieren oder das Design zu testen. Wir mussten die Realisation, d.h. dass mit Java die Requirements überhaupt erfüllbar sind, zuerst mit Prototypen beweisen. Dies betraf Fragen bezüglich Java-Version, XML-Parser, Distribution und Zertifikate (siehe Anhang B Prototyping).

Obwohl wir zuerst dazu tendierten, Java Web Start zu verwenden, wechselten wir dann doch auf einen Installer als Folge des Prototypings. Denn obwohl JavaWebstart von Sun grossgepriesen wird, ist es doch nur eine attraktive Alternative zu einem Java-Applet und nicht als Installationsersatz einer Applikation zu sehen. Will man also eine Java-Applikation als echte Applikation an den Endanwender bringen, führt kein Weg um einen Installer herum. Und wenn dieser Installer nicht plattformunabhängig ist, dann gilt dies zwangsläufig für die ganze Java-Applikation, was den wohl grössten Vorteil von Java zunichte macht.

Das Prototyping motivierte uns, verschiedene Hilfsklassen zu schreiben. Diese erwiesen sich als sehr nützlich. Zum Beispiel fürs XML-Handling bemühten wir uns die Klassen so allgemein wie möglich zu halten, damit wir sie überall einsetzen konnten, wo einfache Zugriffe auf eine XML Struktur nötig waren. Zum Schluss hatten wir einen Document Object Model (DOM) Wrapper geschrieben, der die Unzulänglichkeiten des DOM Interfaces - die umständliche Handhabung der Nodes - transparent machte. Die so entstandenen Klassen sind gut gelungen und ermöglichten uns keine Zeit mehr mit XML spezifischen Methoden zu verschwenden. Somit konnten wir vieles, was wir für Prototyping geschrieben hatten auch für unsere Applikation weiterverwenden. Somit mussten nun nicht alle Mitglieder die neue Technologie voll beherrschen. Auch erweitert ein solches Vorgehen die Skalierbarkeit der Software ungemein. Beim Wechsel auf eine anderes Application Programmer Interface (API) - z.B. beim XML-Parser Interface - kann man einfach die entsprechenden Hilfsklassen neu implementieren.

Das Schreiben von Prototypen ist die Gelegenheit für die Entwickler, sich mit den neuen Technologien tiefer auseinanderzusetzen und diese zu lernen. Zudem decken Prototypen Lücken in den Requirements auf. Wichtig ist auch, dass Prototypen schnell und zu Beginn eines Projektes oder einer Iteration geschrieben werden, um die Kosten bei Projektabbruch oder Neuorientierung möglichst gering zu halten.

5.4 Demo

Am Ende jeder Iteration steht das Demo. Ein Demo soll dem Kunden zeigen, was die Entwickler in dieser Iteration geleistet haben. Hierbei stehen die abgemachten Stories im Vordergrund. Eine Person der Gruppe zeigt dem Kunden die erreichten Stories, indem er diese direkt am Rechner demonstriert. Sie werden mit Hilfe eines GUI präsentiert, damit der Kunde einen visuellen Eindruck erhält.

Wir zeigten unsere Arbeit jeweils im BAG. Der Kunde überliess uns die Präsentation und stellte am Ende Fragen dazu. Daraus erstellten wir mit dem Kunden bereits neue Stories für das anschliessende Planning Game der nächsten Iteration.

Das wichtigste bei einem Demo ist, dass der Kunde etwas zu sehen bekommt. Ein weiterer Faktor beim Demo ist die Präsentation des GUIs. Das Ziel eines Demos ist es auch, den Kunden von der Kompetenz und dem Fleiss der Gruppe zu überzeugen. Der Kunde leitet aus der Art der Präsentation ab, wie sich die Gruppe beim Entwickeln verhält. Also ist eine engagierte Demonstration von Vorteil.

Wichtig ist auch, dass jede geleistete Arbeit, auch wenn sie aus dem GUI nicht direkt ersichtlich ist, hervorgehoben wird, so dass der Kunde am Ende der Demo seine eingekauften Stories gesehen hat und den Arbeitsaufwand hinter dem GUI erahnt.

Zu Beginn unseres Projekts wurde vom Kunden noch kein GUI erwartet. Trotzdem haben wir eine einfache Oberfläche entwickelt, um dem Kunden unsere Fortschritte zu zeigen, was auch gut ankam (siehe Abbildung 5.1).

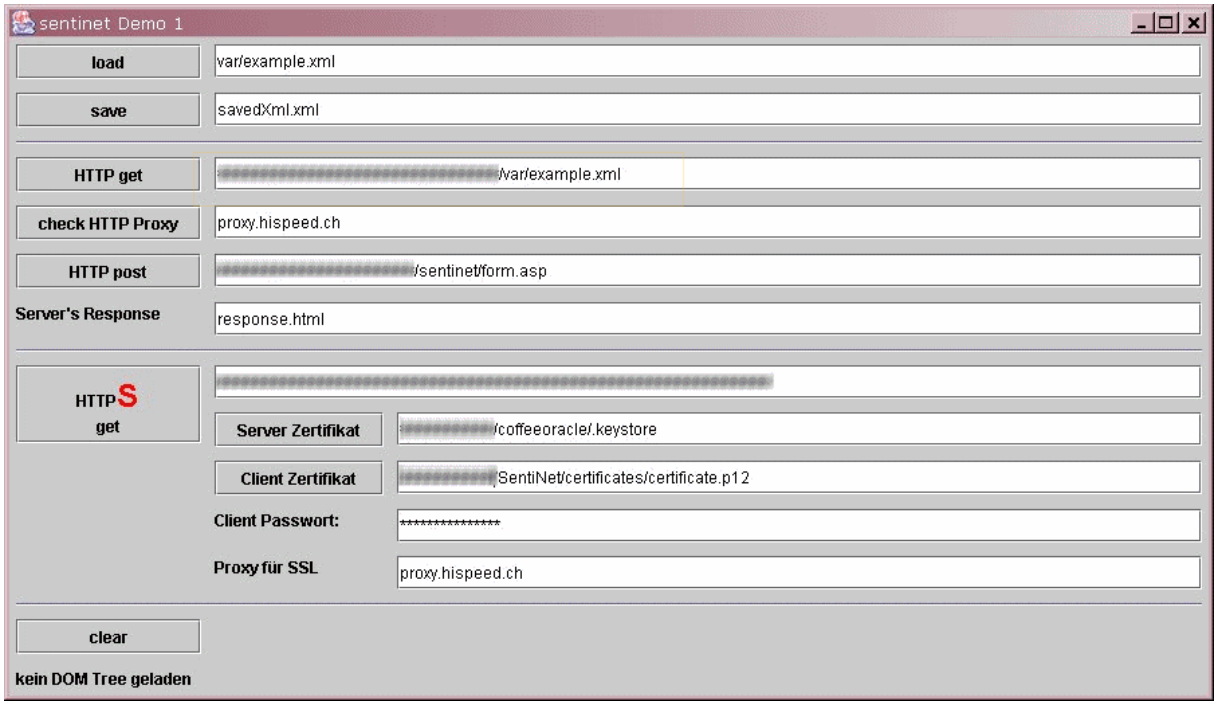

Abbildung 5.1: Screenshot des GUI am Ende der zweiten Iteration.

5.5 Fazit

Der Aufbau einer Applikation in Iterationen hat sich bei uns bewährt. Ohne regelmässige Planning Games mit dem Kunden können wir uns nicht vorstellen, ein Produkt zu erstellen, welches im Sinne des Kunden ist. Auch regelmässige Demos sind für uns wichtig, da wir einerseits dem Kunden regelmässig zeigen können, was wir gemacht haben und andererseits findet man dort auch immer wieder die Bestätigung, ob man auf dem richtigen Weg ist. So kann die Entwicklung noch rechtzeitig in die richtige Bahn gelenkt werden.

Das einzig Mühsame am Iterationsplan ist dessen regelmässige Aktualisierung. Wir vereinbarten zwar anfangs, dass dies jeder für sich macht, jedoch wurde dies nicht immer von allen Mitgliedern gleich sorgfältig getan. So wechselten wir schliesslich dahingehend, dass diese Aufgabe nur eine Person macht, was für diese sehr mühsam war. Ansonsten ist ein Iterationsplan sehr gut für die Moral, da immer wieder kleine Erfolge erzielt werden können wie zum Beispiel der Abschluss einer Story oder eines Milestones. Zudem ist auch eine gute Kontrolle über den Stand der Gruppe vorhanden und man merkt ziemlich früh, ob das Ziel der Iteration nicht erfüllt werden kann.

Kapitel 6

Entwicklung

Wichtige Aspekte beim Extreme Programming sind neben dem Organisatorischen auch die Methodologien bei der Entwicklung. Das Design ist wichtig für die Struktur eines Programms. Dafür existieren verschiedene Techniken, wie Class Responsibility Collaborator (CRC) und Unified Modeling Language (UML). Die Programmierkonventionen definieren, welche Regeln beim Bearbeiten des Quellcodes gelten. Die Validierung sorgt dafür, dass bestehender Code gut wartbar bleibt und regelt gegebenenfalls dessen Überarbeitung.

6.1 Design

Ein Responsibility-driven Design wird mit den folgenden zwei Hilfsmitteln aufgebaut: CRC und UML. Am Beginn des Projekts wird ein Design mit einer CRC-Sitzung entworfen, woraus das erste UML entsteht. Danach finden regelmässig Redesigns statt .

6.1.1 CRC

CRC ist ein wichtiges Werkzeug für einen Designentwurf bei objektorientierten Programmiersprachen. Die Entwickler treffen sich an einem Tisch und besprechen anhand der Requirements ein mögliches Design für die Software. Dabei kommen Kärtchen im Postkartenformat zum Einsatz. Jedes Kärtchen steht für eine Klasse.

Folgende Punkte werden auf den Kärtchen notiert:

Klassenname Nebst dem Namen der entsprechenden Klasse werden Super- und Subklassen vermerkt. Die Klassennamen sollten den Coding Conventions entsprechen und aussagekräftig sein.

Responsibilities Was die Klasse prinzipiell für Funktionalität bereitstellt, dies betrifft bereitgestellte Informationen oder Aktionen.

Collaborations Wie ist die Kommunikation mit anderen Klassen, welche Dienste braucht die entsprechende Klasse, um ihre Aufgabe zu erfüllen.

Wir führten nur einmal eine CRC-Sitzung durch, weil wir die Hälfte der Zeit zum Entwickeln der Prototypen brauchten. Jedoch wäre nun wieder ein Redesign und somit eine CRC-Sitzung nötig. Für unsere CRC-Sitzung vereinbarten wir, dass jeder die Requirements genau kennt und dass wir uns genügend Zeit dafür lassen. Schlussendlich brauchten wir über 6 Stunden, was eigentlich zu lange ist. Jedoch waren dann alle Fragen bezüglich Design geklärt und basierend auf dem Design sahen

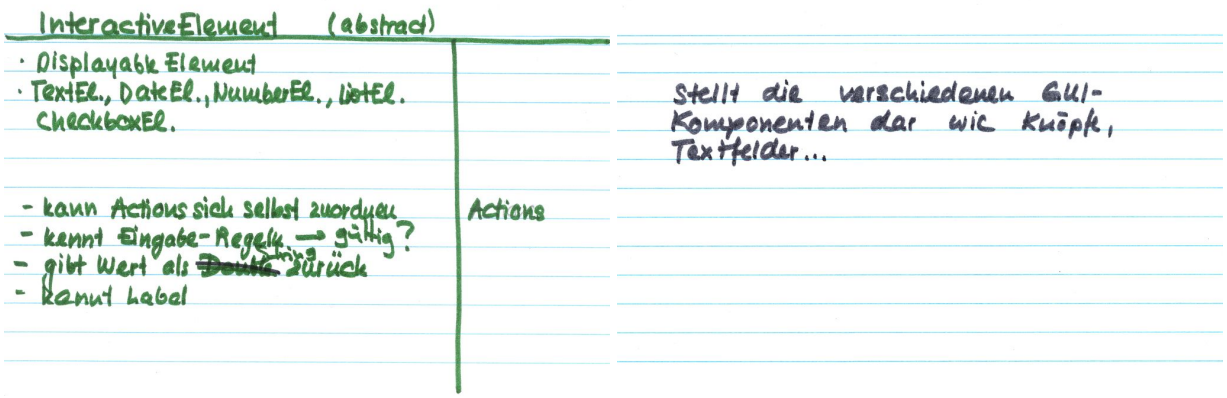

Abbildung 6.1: CRC-Karte

wir auch, wie alle Requirements erfüllt werden könnten. Durch das Prototyping hatten wir schon viele Teilmodule implementiert und daher konnten wir gerade die ganzen Requirements im Design verwirklichen. In PSE hingegen hatten wir zuerst Grundfunktionaltiäten implentiert und im Verlaufe des Projekt immer mehr Anforderungen aus den Requirements ins Design eingebunden.

Bei der CRC-Sitzung kamen viele Unklarheiten bezüglich den Requirements zum Vorschein. Wir trafen dann einfach Annahmen, was wohl der Kunde damit gemeint haben könnte, um ein dementsprechendes Design entwickelt. Bei den kommenden Sitzungen konnten dann diese Fragen geklärt und das Design an die neuen Erkenntnisse angepasst werden.

CRC ermöglicht dynamisch ein Design zu entwickeln, da man ganz einfach Klassen und ihre Kommunikation manipulieren kann. Somit ist eine CRC-Sitzung die beste Gelegenheit, revolutionäre Ideen zu testen und das Design total umzukrempeln. Im Gegensatz zum UML ist ein CRC viel flexibler, es zeigt nicht nur den Status quo. Die den Klassen entsprechenden Objekte werden besser visualisiert als bei einem UML, da es auf dem Level des CRC noch keine Methoden gibt sondern nur Verantwortungen und Aufgaben.

6.1.2 UML

Aufbauend auf die CRC-Sitzung haben wir unser erstes UML gezeichnet und entsprechend unsere ersten Java-Klassen aufgebaut. Von da an haben wir versucht das UML auf dem aktuellsten Stand zu halten. Bei grösseren Änderungen gab es eine neue Version des UML, ansonsten wurde die alte entsprechend angepasst. Diese Arbeit hat nur ein einziges Mitglied des Teams gemacht. Damit nicht alle Gruppenmitglieder die UML-Software erlernen mussten und um das bereits erworbene Knowhow des UML-Zeichners auszunutzen, blieb diese Aufgabe während dem ganzen Projekt beim gleichen Teammitglied.

Für das Zeichnen des UMLs benutzten wir ausschliesslich die Vektorzeichnungs-Software `Visio` [9] und haben das UML selbständig mit Grafikelementen von Grund auf aufgebaut. Die in Visio eingebaute UML-Verwaltung und andere UML-Tools haben unseren Ansprüchen nicht genügt.

Damit der UML-Zeichner nicht immer den ganzen Quellcode nach Änderungen durchsuchen musste, haben wir abgemacht, dass man UML relevante Änderungen beim Einchecken ins CVS kommentiert.
Unsere erste und letzte UML-Version stehen im Anhang A.

Das Zeichnen des UMLs erwies sich als grosse Hilfe für das Schreiben des weiteren Codes.

Man hatte eine feste Grundlage, wo man bei Unsicherheit nachschauen konnte. Programmabläufe und Optimiereungsmöglichkeiten wurden sichtbar. Grössere Designentscheidungen trafen wir häufig mithilfe des UML-Diagramms.

Das Problem waren die häufigen Änderungen, was aber bei XP normal ist. Damit das UML überhaupt auf dem aktuellsten Stand blieb, mussten beinahe täglich Anpassungen vorgenommen werden. Dies kostete viel Zeit, im Durschnitt etwa eine bis zwei Stunden pro Tag. Kaum hatte man das UML wieder aktualisiert, konnte man es am Abend schon wieder auf den Kopf stellen, was sehr frustrierend war.

Der Beschluss der Gruppe, Änderungen im CVS zu kommentieren, vereinfachte und beschleunigte das Nachführen des UML-Diagramms. Aber auch so kostete das Führen des UMLs noch viel Zeit - Zeit, die wir nie richtig in die Planung einbezogen haben und die dann meistens gegen Schluss der Iteration gefehlt hat. Man sollte nie unterschätzen, wie viel Arbeit das Führen eines UML-Diagramms benötigt. Diese Zeit muss auch unbedingt investiert werden, denn wenn man kein aktuelles UML hat, dann lieber keines machen als soviel Zeit für halbfertige UMLs zu verlieren. Die beste Lösung ist, wenn man zum Vornherein mindestens 10 Stunden pro Woche für das UML-Zeichnen beiseite legt, dafür das UML aber dann auch ausführlich und auf dem aktuellsten Stand hält.

Bei dieser Art ein UML zu führen, hinkt das Diagramm immer dem Quellcode hinterher. So bleibt das UML nur eine Referenz zur Software und bietet keine Möglichkeit den Quellcode direkt damit zu editieren und zu verändern. Abhilfe könnte `Together/J` *[10] bieten, doch hat solche Software, neben den Kosten, auch wiederum ihre Nachteile [11].*

6.2 Programmierkonventionen

Die meisten Programmierer entwickeln ihren Code alleine. In Extreme Programming wird stattdessen in Zweiergruppen gearbeitet, was Pairprogramming genannt wird. Dabei konzentriert sich eine Gruppe nicht auf einen bestimmten Codeteil. Deshalb ist es wichtig, dass der ganze Code einheitlich geschrieben wird. Dies wird in den Coding Conventions verfasst.

6.2.1 Pairprogramming

Beim Pairprogramming[1] arbeiten zwei Programmierer vor demselben Bildschirm und bearbeiten einen bestimmten Codeteil gemeinsam. Während ein Programmierer schreibt, beobachtet der andere dessen Arbeit. Der Beobachter leistet den grossen Teil der Denkarbeit und ist dafür zuständig, Fehler zu finden, die der Schreiber in seinem Codeabschnitt macht.

Während unserem Projekt Sentinet haben wir so oft als möglich mit Pairprogramming gearbeitet. Dies war jedoch nicht immer so einfach wie wir uns dies zu Beginn des Projekts vorgestellt haben. Zwei Personen arbeiteten nebenbei und eine andere war jedes Wochenende abwesend. Es gestaltete sich deshalb schwierig, Termine zu finden, an denen alle verfügbar waren. Wir lösten dies, indem wir durch die Woche mit einer Gruppenkonstellation arbeiteten und am Wochenende mit einer anderen. Aus dem selben Grund waren oft nur drei Personen gleichzeitig am Arbeiten. Dies implizierte, dass zwei Personen als Gruppe arbeiteten und die dritte Person alleine programmierte.

[1] ressources/PairProgramming.html (auf CD)

Da mussten wir manchmal ein wenig diskutieren, wer alleine arbeiten soll, denn natürlich ist es viel angenehmer, in einer Gruppe zu arbeiten als alleine. Wir haben auch versucht eine Dreiergruppe zu bilden, doch dies war dann eindeutig eine zu grosse Gruppe um effizient arbeiten zu können.

Das Pairprogramming ist sicher ein guter Ansatz und in vielen Fällen hat es in unserem Projekt für Code mit wenigeren Fehlern gesorgt. Doch in einigen Fällen wäre es sicher sinnvoller gewesen, alleine zu programmieren. So zum Beispiel, wenn man an einem bereits begonnenen Codeteil weiterentwickeln soll, den man selbst durchgedacht hat, aber den der Mitprogrammierer noch nicht kennt.

Natürlich ist es ein Grundkonzept von Extreme Programming, dass jeder den gesamten Code versteht, doch in vielen Fällen lohnte es sich zeitlich nicht, dass sich jeder in den gesamten Code einarbeitet. In manchen Fällen haben wir vereinbart, dass gewisse Codeabschnitte nur von einigen Programmierern bearbeitet werden. In diesen Fällen mussten wir dann auch einige Einschränkungen bei der Gruppenbildung eingehen, es konnte ja nicht mehr jeder jeden Codeteil übernehmen. Doch dies war für uns weniger schlimm als der Zeitaufwand, der sich durch die Einarbeitung ergeben hätte. So bearbeitete besonders gegen Ende des Projekts eine Gruppe meistens ein bestimmter Codeteil, während die andere für einen anderen Teil zuständig war.

6.2.2 Coding Conventions

Coding Conventions sind beim Extreme Programming wichtig, da hier jeder Programmierer mit dem gesamten Code vertraut sein muss. Da jeder Programmierer einen eigenen Programmierstil pflegt, müsste sich ein Entwickler, der bestehende Codeteile von anderen Programmierern anpassen oder erweitern möchte, immer wieder an einen anderen Programmierstil gewöhnen. Mit den Coding Conventions kann dies vermieden werden, das heisst, Quellcode kann einheitlich gehalten werden, was die Wartung erleichtert und die Anzahl der Fehlerquellen minimiert. Im Übrigen sind Coding Conventions nicht nur in mit Extreme Programming entwickelten Programmen sehr zu empfehlen, sondern auch in solchen, wo das ganze Programm oder Codeteile von einem Programmierer geschrieben und gewartet werden, so werden zum Beispiel spätere Wartungen an Programmen oft nicht mehr vom Programmierer durchgeführt, der den Code auch geschrieben hat. Auch wenn der ursprüngliche Programmierer diese Wartungen durchführt, nach einiger Zeit muss auch er sich zuerst wieder in den eigenen Code einlesen.

Beim Schreiben unserer Coding Conventions[2] haben wir uns einerseits an den offiziellen Java Coding Conventions[3] von Sun orientiert und an denjenigen, die wir früher an der Universität gebraucht haben[4].

Nachdem die Coding Conventions von einer Person aufgesetzt waren, wurde kurz besprochen, ob alle mit dem gemachten Vorschlag einverstanden seien. Es mussten noch einige Anpassungen gemacht werden. Doch wahrscheinlich haben sich bis zu diesem Zeitpunkt die anderen Teammitglieder noch zu wenig mit den Coding Conventions auseinandergesetzt. Auf jeden Fall mussten sie während dem Projekt noch einige Male angepasst werden, weil jemand mit etwas nicht einverstanden war. Des Weiteren wurde während dem Projekt zu wenig auf diese Coding Conventions geachtet. Während dem Projekt wurde einmal ein Code Review gemacht, bei welchem nur auf die Einhaltung der Coding Conventions geachtet wurde. Doch diese Kritiken wurde grösstenteils nicht gross beachtet und nicht umgesetzt, da bei uns im Vordergrund stand, dass wir mit den Milestones termingerecht fertig werden. Und da uns Korrekturen gemäss den Coding Conventions Zeit kosteten, uns aber den vereinbarten

[2] Coding Conventions swiki/16.html (auf CD)

[3] ressources/CodeConventions.pdf (auf CD)

[4] ressources/projektHandbuch/codeInspections/JavaStyleGuide.html (auf CD)

Milestones nicht näher brachten, wollten wir meistens nicht Zeit dafür aufwenden. Weil die Coding Conventions einige Male änderten, wusste man oft nicht, welche Version aktuell war und konnte sie auch deshalb nicht einhalten. Wahrscheinlich haben wir uns auch noch zu sehr in Kleinigkeiten verloren, als uns auf die wesentlichen Dinge der Coding Convention zu konzentrieren.

Wenn neuer Code geschrieben wurde, dann wurden die Coding Conventions recht gut eingehalten, jedoch besonders dann, wenn man schnell etwas in bestehenden Code eingefügt hat, wurden sie weniger gut beachtet.

6.3 Validierung

Bei der Validierung wird mit Tests sichergestellt, dass der Code wirklich macht, was man möchte. Des Weiteren muss sichergestellt werden. dass die implementierte Funktionalität dem entspricht, was sich der Kunde wünscht. Dafür werden regelmässig Demos mit dem Kunden durchgeführt. Das Demo kann auch aus einem anderen Blickwinkel, nämlich als Bestandteil einer Iteration, betrachtet werden (siehe dafür Abschnitt 5.4). Damit Code einfach wartbar bleibt, werden regelmässig Code Reviews durchgeführt. Dies hilft viele Fehler zu vermeiden.

6.3.1 Code Review

Beim Code Review wird bestehender Code durchgearbeitet und es werden Mängel behoben. Das heisst, es wird keine neue Funktionalität hinzugefügt, sondern der bestehende Code wird von Fehler bereinigt sowie effizienter und lesbarer gestaltet. Einerseits achtet man darauf, dass die Coding Conventions eingehalten werden (siehe Abschnitt 6.2.2), andererseits spürt man zum Beispiel doppelt vorhandener Code auf und lagert ihn in Methoden aus oder man löscht Methoden, die nie aufgerufen werden. Es wird bei objektorientierten Sprachen auch überprüft, ob die Aufteilung in die vorhandenen Klassen sinnvoll ist. Eine Klasse sollte eine ganz bestimmte Aufgabe umfassen. Häufig geschieht es jedoch, dass man einer Klasse während der Entwicklung immer mehr Aufgaben zukommen lässt. In diesem Fall sollte man diese Klasse in zwei oder mehr Klassen aufteilen. Oder man merkt mit der wachsenden Zahl von Klassen, dass eine Methode besser in eine andere Klasse - oft eine Super- oder Subklasse - passen würde. Es wird auch überprüft, ob nur diejenigen Variablen und Methoden öffentlich sind, die es auch wirklich sein müssen. Die anderen sollten je nach Einsatz entweder als geschützt oder als privat deklariert werden. Des Weiteren sollte dafür gesorgt werden, dass der Code immer ausreichend kommentiert ist, in Java bietet sich dafür Javadoc [5] an. Das heisst, es wird alles überarbeitet, was den Code ineffizient und schlechter lesbar macht. Wenn man den Code überarbeitet, bezeichnet man dies als Refactoring. In vielen Fällen kann der Code auf diese Weise sogar schneller gemacht werden.

Wir haben regelmässig überprüft, ob das aktuelle Design Sinn macht (siehe Abbschnitt 6.1.1 Design) oder ob sich etwas optimieren lässt. Zudem haben wir einmal den Code durchgesehen, ob die Coding Conventions eingehalten wurden.

Wir haben - wahrscheinlich wie die meisten Programmierteams - uns viel zu wenig um Code Reviews gekümmert. Doch wir fanden es nicht so wichtig, den bestehenden Code gemäss Coding Conventions zu überarbeiten, weil wir meistens genug mit den Stories zu tun hatten. Wir haben die Arbeit, welche die Stories zu tun gaben, meistens sogar unterschätzt. So waren wir froh, wenn wir die Stories erfüllen konnten. Für die Überarbeitung des Codes, was nicht direkt zur Erfüllung der Stories

[5] ressources/WritingDocComments.html (auf CD)

beitrug oder zumindest vom Kunden nicht bemerkt wurde, hatten wir dann meistens keine Zeit mehr. Momentan sind wir aber an der Stelle angelangt, an welcher wir zugeben müssen, dass der gesamte Code einmal überarbeitet werden sollte. So sollte auch das Design überarbeitet und der Code muss gemäss den Coding Conventions umformatiert werden. Des Weiteren sollte auch das Javadoc überall vervollständigt und einheitlich gestaltet werden.

6.3.2 Tests

Tests sind ein wichtiger und oft zu wenig beachteter Bestandteil von Extreme Programming. Tests sollen helfen, Code so stabil als möglich zu schreiben. Sie werden vor oder während der Implementation geschrieben. Dann kann man den gerade geschriebenen Code auf seine Korrektheit überprüfen und man kann sich sicher sein, dass ein bestimmter Codeteil das macht, was man möchte. In Java-Projekten wird für die Tests meistens `JUnit` [12] verwendet.

Wir haben zuerst einen Teil implementiert und erst im Nachhinein die Tests geschrieben, und nicht vor oder während der Implementation. Nun war es aber beim Testschreiben ähnlich wie beim Code Review, wir mussten Funktionalität vorweisen können. Deshalb haben wir unsere Zeit eher dafür verwendet, neue Funktionalität zu implementieren und haben folglich nicht so viele Tests geschrieben.

Niemand wollte gerne Tests schreiben. Doch waren einmal Tests vorhanden, sind alle froh gewesen, dass der Code auf seine Korrektheit überprüft werden konnte. Wir haben eingesehen, dass Tests sinnvoll sind, doch war es halt nicht besonders attraktiv, diese zu schreiben. Es ist natürlich viel interessanter, neue Funktionalität hinzuzufügen, als die gegebene Funktionalität auf ihre Korrektheit zu überprüfen. Zu Beginn des Projekts haben wir uns seriöser um die Tests gekümmert als gegen Ende des Projekts. Doch richtige quiet tests waren es oft nicht, wir wandelten zum Beispiel Daten in XML um und schrieben sie in eine Datei. Dies lässt sich nur schwer als quiet test implementieren. So haben wir in den Tests einfach die Daten gespeichert und der Test wurde erfolgreich durchlaufen, wenn die Datei auf die Festplatte geschrieben werden konnte. Wir mussten dann von Hand überprüfen, ob unsere Daten korrekt in XML umgewandelt wurden. Wahrscheinlich wäre es viel komplizierter gewesen, dies mittels richtigem quiet testing zu machen. Deshalb haben wir uns für diese Variante entschieden. In der zweiten Hälfte des Projekts konnten wir praktisch keine Tests mehr schreiben, da wir ein dynamisches GUI generieren mussten, und dies lässt sich nun einmal nicht in Tests überprüfen.

Besonders hilfreich fanden wir Tests, wenn wir bestehenden Code abänderten. So konnte auf einfache und schnelle Art festgestellt werden, ob der Code nach der Änderung immer noch die alten Anforderungen erfüllt. Man musste deshalb keine Angst haben, dass man bei der Anpassung eines bestimmten Codeteils irgendetwas übersieht und deshalb ein Teil des Programms nicht mehr funktioniert. Diese Fehler lassen sich erfahrungsgemäss nur mit viel Aufwand finden.

6.3.3 Demo

Das Demo dient dazu, - wenn man es aus der Sicht der Validierung betrachtet - sich zu vergewissern, dass man die Wünsche und Anforderungen des Kunden richtig verstanden hat. In der Regel beginnt man eine Iteration mit einem Planing Game und präsentiert am Ende ein entsprechendes Demo.

Wir haben meistens alle zwei Wochen ein Demo durchgeführt. Die geforderte Funktionalität konnte immer geliefert werden, jedoch kam es manchmal zu Missverständnissen. So sagte uns der

Kunde nicht immer alles, was wir an Informationen für die folgende Iteration brauchten. So waren wir gezwungen, gewisse Dinge explizit nachzufragen oder selbst auszuprobieren.

Eine Funktion des Demos ist abzuklären, ob wir die Wünsche des Kunden korrekt umsetzen. Falls wir nämlich nicht gewünschte Funktionalität implementieren, ist dies in einem frühen Stadium einfacher anzupassen als wenn das Projekt schon weiter fortgeschritten ist. Und wenn wir dem Kunden ein gelungenes Demo präsentieren konnten und wir merkten, dass dieser zufrieden war, so war dies eine Motivation für uns in der folgenden Iteration.

Das Demo war auch für unseren Kunden sehr wichtig, so konnte er sich vom Fortschreiten des Projekts überzeugen und er erkannte, dass er ausführlicher Informationen geben musste. Denn manchmal erläuterte uns der Kunde nicht alles, was für uns und das Entwickeln nötig war. Vielleicht wusste er auch gar nicht, was wir alles wissen mussten. Auf jeden Fall konnte auch der Kunde so frühzeitig einschreiten, wenn ein Demo nicht seinen Vorstellungen entsprach.

Das Demo muss die Funktionalität beinhalten, die man in der Iteration dem Kunden versprochen hat. Es soll nicht - wie dies vielleicht im herkömmlichen Sinn verstanden werden könnte - eine praktisch fertige Version sein, sondern es soll die Arbeit wiederspiegeln, die in der letzten Iteration gemacht wurde.

6.4 Fazit

Obwohl alle Punkte in diesem Kapitel Elemente von Extreme Programming sind, ist nicht alles gleich wichtig. Auf ein übersichtliches Design kann man wahrlich nicht verzichten und so hätten wir auch UML Diagramme geführt, wenn wir das Projekt nicht mit XP realisiert hätten. Eine CRC-Sitzung hingegen hätte ohne XP wohl nicht stattgefunden. Es gibt andere Möglichkeiten ein Design aufzustellen, auch ohne direkt mit Kärtchen zu hantieren. Trotzdem sind wir froh, so viel Zeit in ein gutes Design investiert zu haben.

XP-Programmierkonventionen sind nur bedingt vorteilhaft. Vor allem beim Pairprogramming kann man sich streiten, ob es wirklich einen Vorteil bringt. Bei so einem kleinen Team und sowieso knappem Zeitbudget, verliert man zu zweit mehr Zeit, als einem der Vorteil von weniger Fehler im Code nützt. Wir hätten besser eine Person nur für Tests schreiben verpflichtet und hätten, nebst einer guten Validierung, die Fehler auch so gefunden.

Nicht jeder kann oder will mit jedem gleich gut zusammenarbeiten. So birgt Pairprogramming die Gefahr einer Spaltung des Teams in Gruppen. Dies kann zu Unzufriedenheit und Ärger führen, und ist kontraproduktiv für den Teamgeist. Auch bei uns kamen an einem arbeitsintensiven Wochenende deswegen Spannungen vor. Andererseits hat Pairprogramming auch für interessantere Programmierstunden gesorgt. Dieser soziale Aspekt ist auch wiederum für die Arbeitsmoral wichtig.

Eine seriöse Validierung mit Code Reviews und ausgiebigen Tests haben wir vernachlässigt und mehr auf das Urteil des Kunden beim Demo gesetzt. Uns mangelte es meistens an Selbstdisziplin um Tests zu schreiben. Die Testerrolle in XP wollen wir in zukünftigen Projekten ernsthafter zuteilen, um nicht nur das Demo als Validierung zu haben.

Kapitel 7

Konklusionen

Unser Projekt war für das BAG ein Test. Sie wollten herausfinden, ob sich die bestehende Applikation Sentinet verbessern lässt und mit welchem Aufwand dies verbunden ist. Diese Abklärungen waren für das BAG jedoch nicht dringend und wir konnten dieses Projekt vor allem durchführen, weil es das BAG nichts kostete. Dies hat jedoch zur Folge, dass dem BAG nun kein Budget zur Verfügung steht, um das halb fertige Projekt noch zu Ende zu führen. Dies ist besonders hart, da unser Projekt eigentlich als gelungen betrachtet werden kann. Es konnten alle Anforderungen mit Java erfüllt werden.

Die Anwendung von Extreme Programming hat uns insofern viel gebracht, dass wir genau den Bedürfnissen des Kunden entsprechend entwickeln konnten. Das Problem vieler konventioneller Projekte, dass das Projekt nicht innerhalb der Zeit und dem Budget fertiggestellt werden kann, hatten wir auch. Der Grund davon liegt aber in den anfangs vagen Anforderungen und nicht in sonstigen Faktoren. Unser Fazit zu Extreme Programming hat zwei Facetten, einmal alle projektspezifischen und dann noch entwicklungsspezifischen Aspekte.

7.1 Projektspezifisches Fazit über XP

Eine eindeutige projektspezifische Aussage über XP zu machen, gelingt uns auch nach diesem Projekt nur ansatzweise. Zu stark sind die Abweichungen zu einer Software-Entwicklungsgruppe in einer Firma. Grösstes Hindernis war die homogene Gruppe, welche es uns nicht erlaubte, eine ausgeglichene Rollenverteilung zu organisieren.

Vorteile gegenüber herkömmlichen Entwicklungsmethodologien zeigen sich besonders im Projektmanagement. Das iterative Entwickeln erlaubt es, immer wieder die eigene Arbeit zu reflektieren und die gewonnen Erkenntnisse direkt ins Projekt einfliessen zu lassen. Dabei können neu auftretende Probleme früh erkannt werden. Im Gegensatz zum Wasserfallprinzip werden regelmässige Risikoanalysen abgehalten, um neue Risiken zu erkennen. Dort kann nicht auf vergangene Iterationen zurückgeschaut werden, um aus den begangenen Fehlern zu lernen.

Die Aufteilung des Projektes in Iterationen macht es erst möglich ein Produkt zu erstellen, welches den Anforderungen des Kunden wirklich entspricht. Wir hatten zu Beginn des Projekts eine völlig andere Vorstellung, wie unsere Arbeit am Ende aussehen würde. Nur dank den regelmässigen Sitzungen und den kurzen Iterationen erhielten wir ein aktuelles Feedback. So entdeckte der Kunde bei fast jedem Demo noch Kleinigkeiten, die ihm nicht gefielen, obwohl wir in jeder Iteration die Stories des Kunden erfüllen konnten. In einer wasserfallartigen Entwicklung fehlt ein solches Feedback im laufenden Projekt gänzlich, was in einer Enttäuschung des Kunden enden muss.

Das heisst also, dass dem Kunden in einem XP-Projekt besser gedient werden kann. Auch für die Programmierer ist es angenehmer, eng mit dem Kunden zusammen zu arbeiten, da sie besser auf die Bedürfnisse des Kunden eingehen können und schliesslich ein besseres Produkt abliefern können. Der einzige Nachteil dieser Techniken ist deren hoher Zeitaufwand.

7.2 Entwicklungsspezifisches Fazit über XP

Die verschiedenen XP-Methodologien bei der Entwicklung sollen zu stabilen und hochwertigen Code führen. CRC-Sitzungen und UML-Diagramme ermöglichen es, bereits vor dem Implementieren herauszufinden, welches das beste Design zum Entwickeln ist. Ein munteres Drauflos-Programmieren führt bereits in kleineren Projekten zum Chaos, besonders wenn in einer Gruppe gearbeitet wird. Das aktuelle UML-Diagramm sorgt zudem dafür, dass man eine überschaubare Referenz zum Design hat.

Die Methode des Pairprogrammings ist empfehlenswert. Um sie erfolgreich anzuwenden, braucht es ein gut eingespieltes Team, welches unbedingt auf demselben technischen Niveau sein muss. Beide müssen einen ähnlichen Programmierstil entwickeln, so dass sie sich nicht andauernd am Stil des anderen stören. Doch in diesem Fall helfen Coding Conventions. Fürs Implementieren von einfachen Grundstrukturen ist es manchmal jedoch besser, wenn dies nur eine Person allein macht.

Um möglichst fehlerfreien Code zu produzieren, sind regelmässige Validierungs-Massnahmen sehr wichtig. XP gibt gute Richtlinien vor, wie das gemacht wird. Ein Code Review ist allerdings nur bedingt zu empfehlen. Falls man sich anschliessend nicht die Zeit nimmt, die Kritikpunkte des Code Reviews zu korrigieren, hat man nur einen begrenzten Nutzen. Auch wurden in Einführung Software Engineering (ESE) und Praktikum Software Engineering (PSE) diese zu spät abgehalten. Es lohnte sich nicht mehr, grosse Änderungen am Design vorzunehmen, nur des schöneren Codes willen. Am meisten haben sich Demos und Tests zur Validierung bewährt.

Anhang A

UML-Diagramme

Die hier aufgeführten UML-Diagramme sind zum groben Vergleich zwischen der ersten und der letzten Version gedacht. Für detailliertes Studieren der UML-Diagramme, verweisen wir auf die CD[1].

A.1 Erste UML-Version

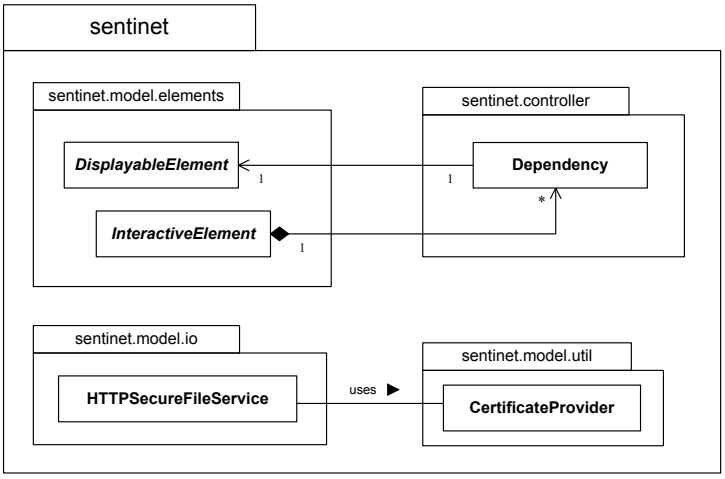

Abbildung A.1: Übersicht der Pakete

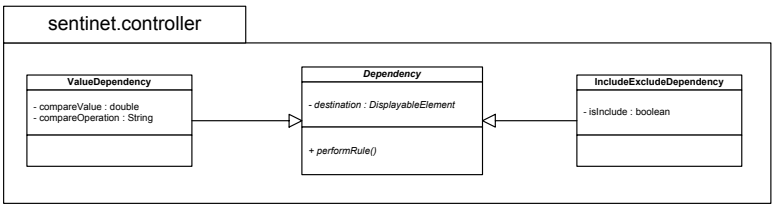

Abbildung A.2: Controller-Paket

[1] Verzeichnis uml/ (auf CD)

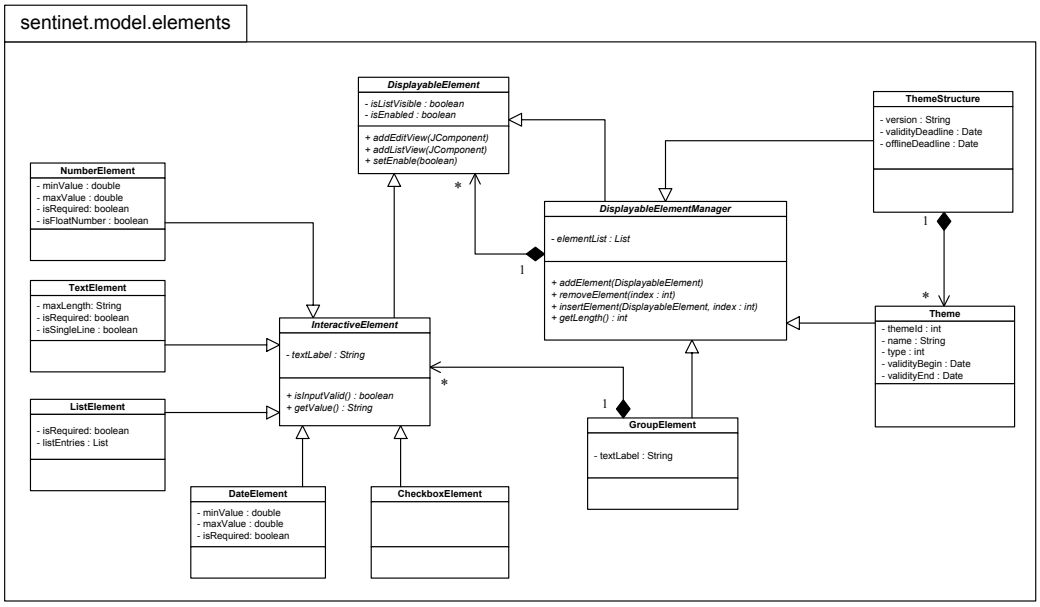

Abbildung A.3: Model-Paket der Elemente

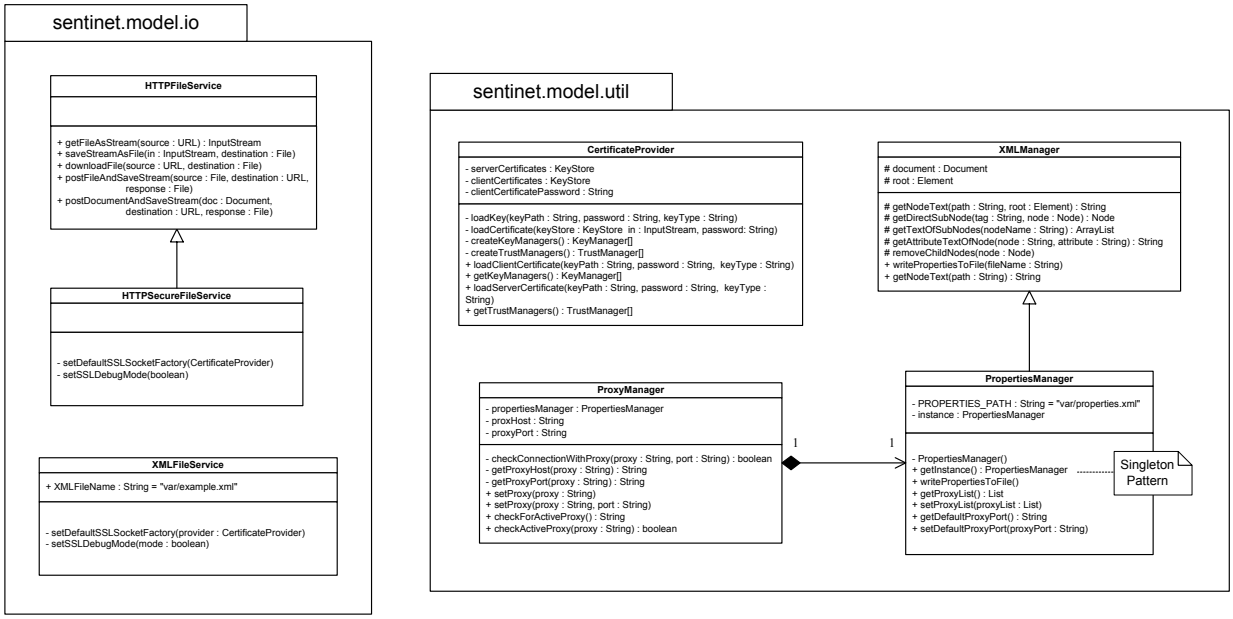

Abbildung A.4: Input/Output- und Hilfspaket

A.2 Letzte UML-Version

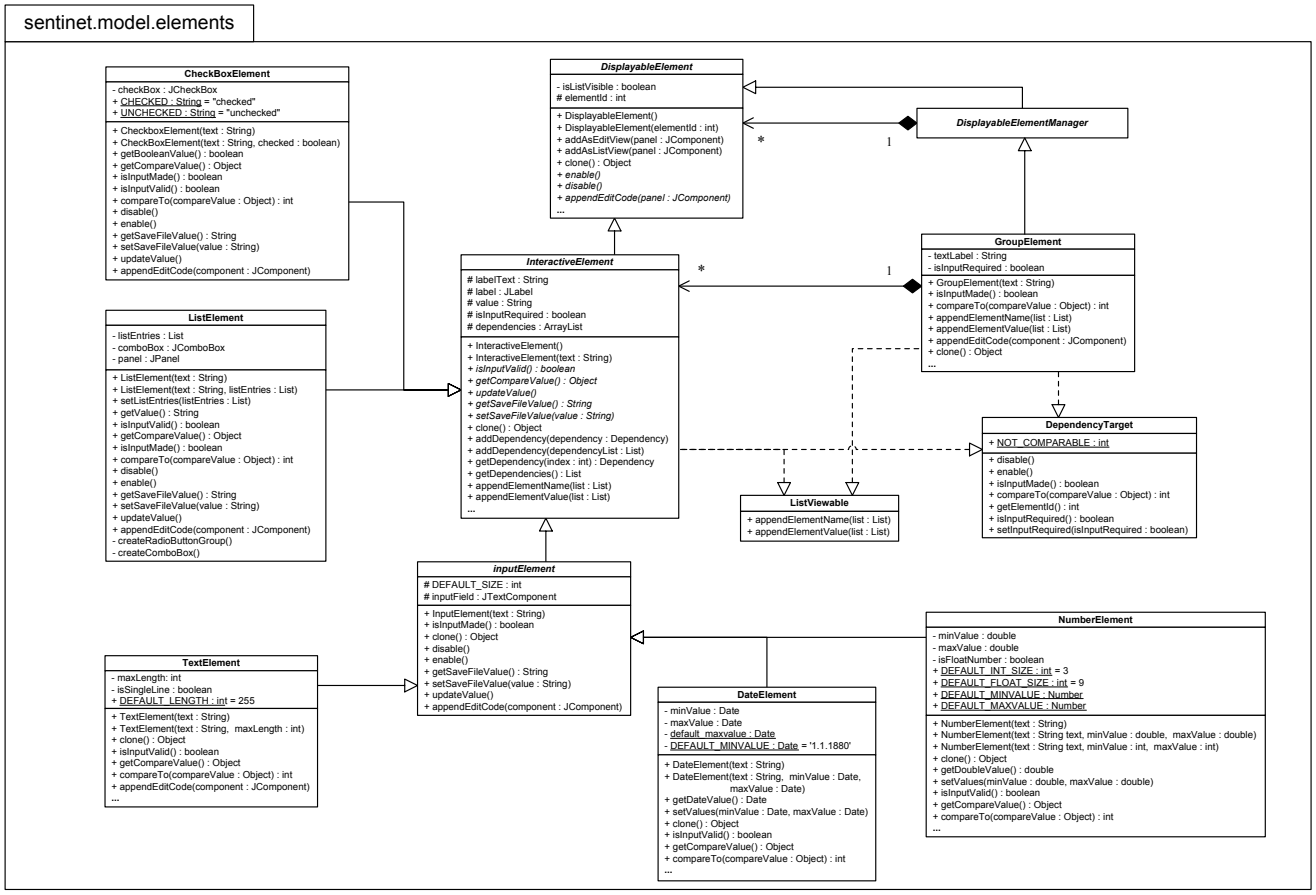

Abbildung A.5: 1. Teil des Model.Elements Paketes

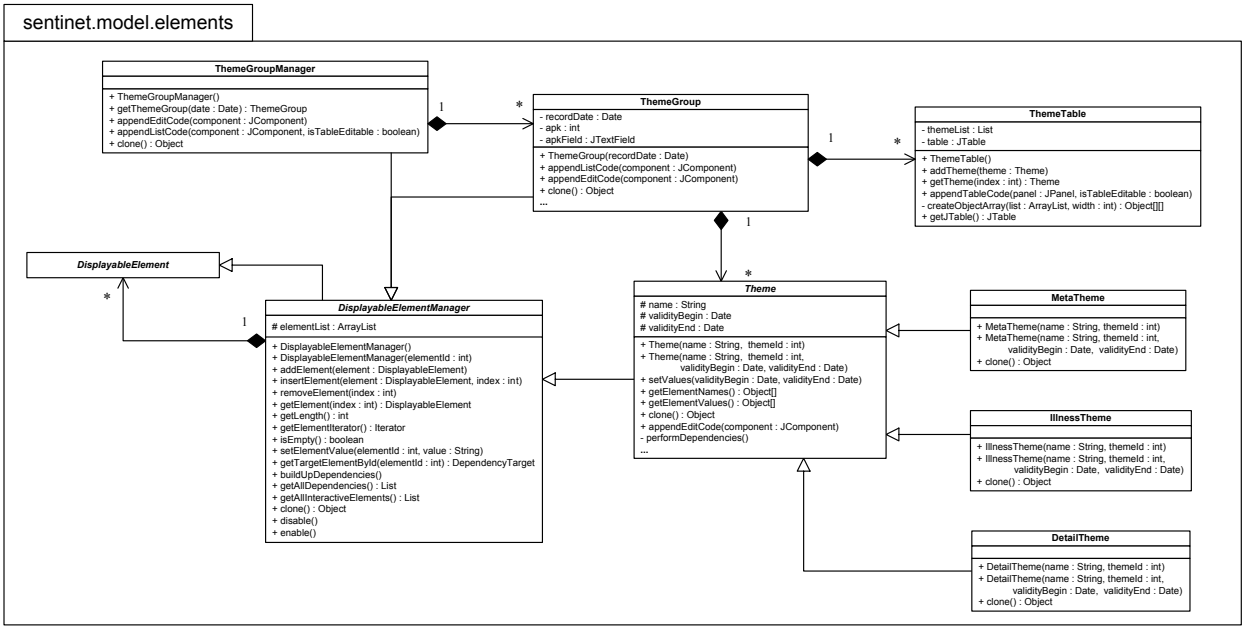

Abbildung A.6: 2. Teil des Model.Elements Paketes

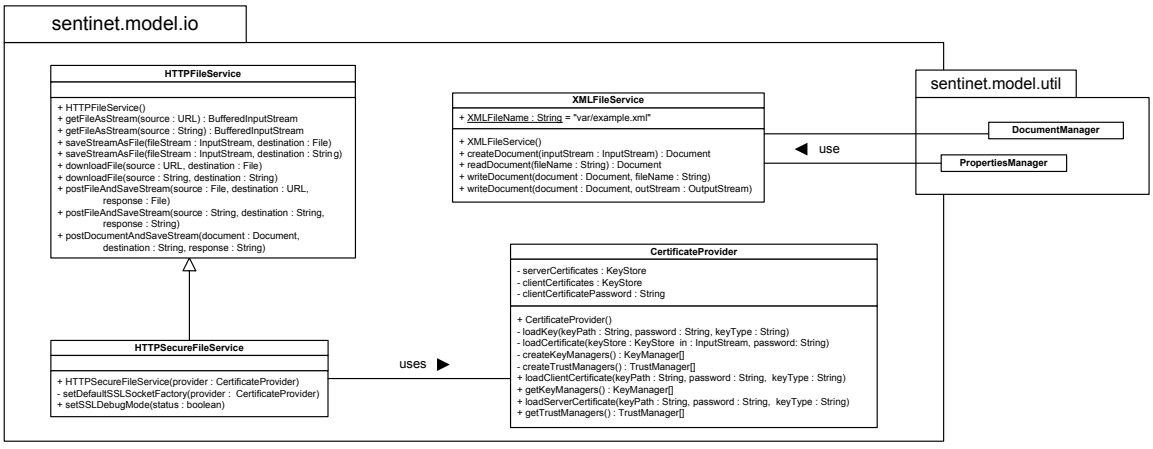

Abbildung A.7: Input/Output-Paket

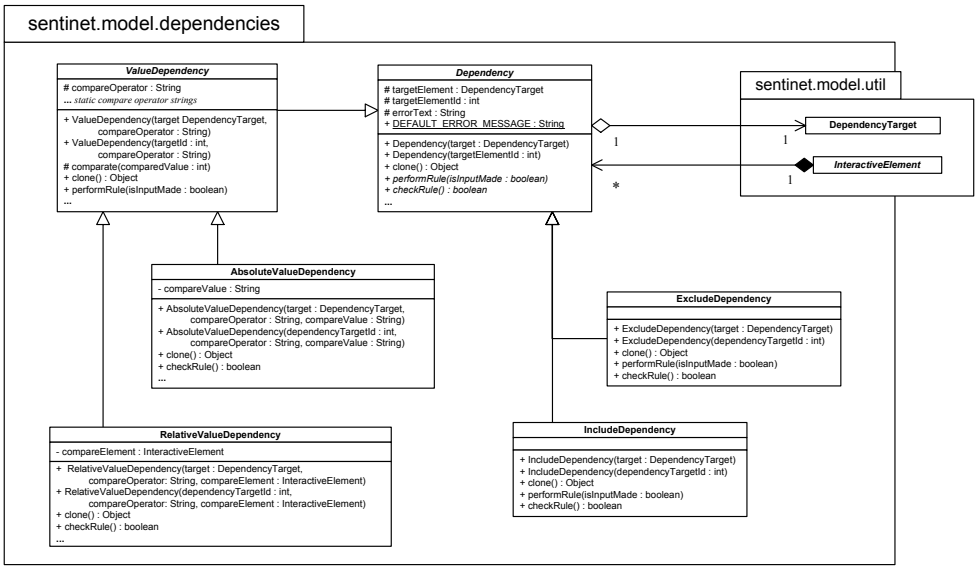

Abbildung A.8: Abhängigkeitsklassen für die Elemente

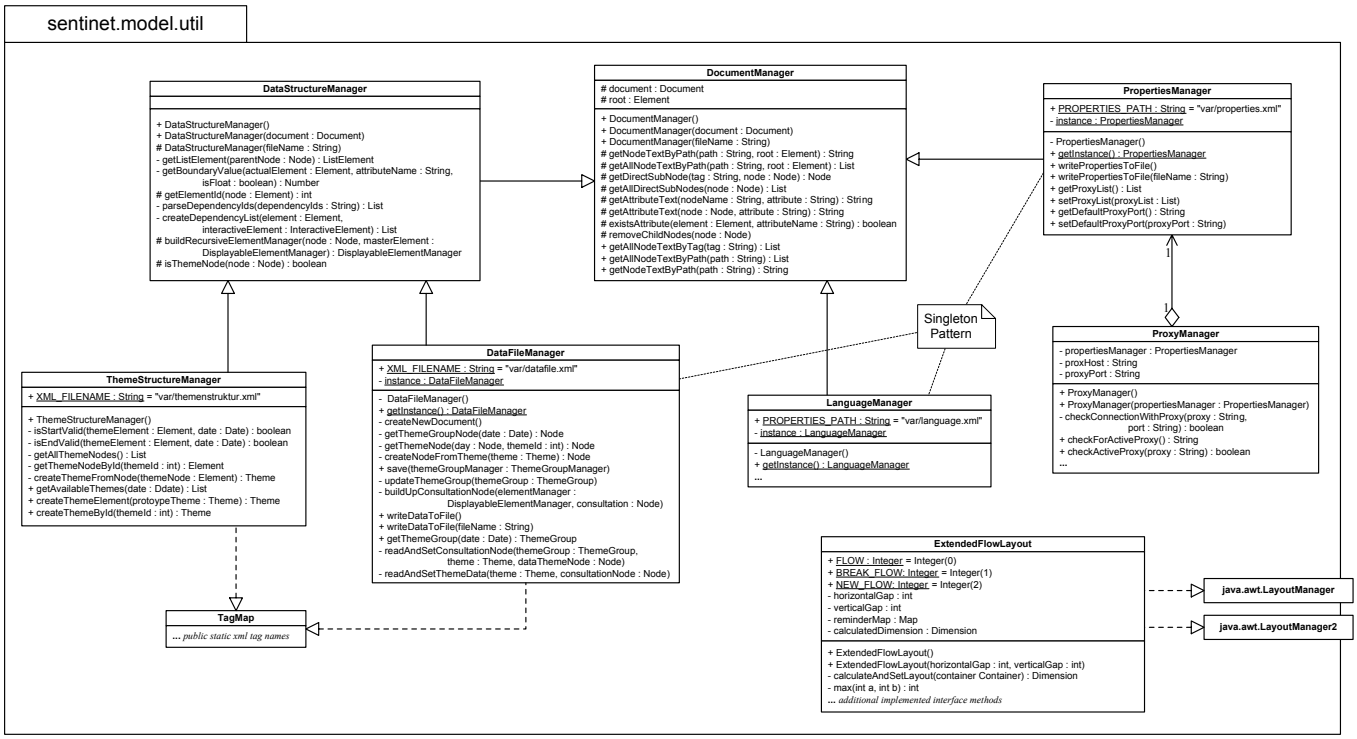

Abbildung A.9: Hilfsklassen

Anhang B

Prototyping

Bei allen Softwareprojekten wo man sich an etwas Unbekanntes wagt, seien es neue Technologien oder auch neue Funktionalitäten der Software, ist es von Vorteil - wenn nicht sogar zwingend - dass man Unsicherheiten und Fragen klären und/oder aus dem Weg räumen kann.

Hier seien unsere wichtigsten Prototypen beschrieben, insbesondere die Frage nach einer einfachen Distribution der Applikation wie auch die in den Wolken stehende SSL-Verbindung mit Zertifikatsauthentifizierung über einen Reverse-Proxy.

B.1 XML

Die Requirements verlangten Umgang mit XML-Dateien, was einige Fragen aufwarf. Folgende Punkte mussten mit einem Prototypen geklärt werden:

- Sollen wir `JDK 1.3` mit einem externen XML-Parser oder die neue, aber noch unbekannte, Version 1.4 mit integrierter XML-API benutzen?
- Im Falle eines externen Parsers, welchen wollen wir benutzen?
- Welches Parser-Interface wollen wir benutzen? Simple API for XML (SAX) oder Document Object Model (DOM)[1]?

Als Resultat schrieben wir folgende zwei Hilfsklassen:

- `DocumentManager` Einfache Handhabung eines DOM-Trees, wie bequeme Zugriffsmöglichkeiten auf XML-Tags und deren Attribute. Also ein Wrapper über die DOM-API.
- `XMLFileService` Lesen und Speichern von XML-Dateien bzw. korrektes Initialisieren des XML-Parsers.

Als Konsequenz des Prototypen beschlossen wir fortan auf `JDK 1.4` und der DOM-API zu setzen, und dessen integrierten XML-Parser Xerces [13] zu benutzen.

B.2 Distribution

Eine einfache Installation der Software war eine der dringlichsten Anforderungen vom Kunden. Auch wenn unsere Java-Applikation an und für sich überall mit "`java -jar sentinet.jar`" gestartet werden konnte, sind solche Schritte dem Endanwender nicht zumutbar. Die Installation unserer Applikation musste folgenden Anforderungen genügen:

[1] ressources/SAXvsDOM.pdf (auf CD)

- Die Software muss vom Internet oder von CD mit wenigen Mausklicks installiert werden können und ebenso leicht zum Laufen gebracht werden.
- Die Installation muss auf jedem der heute üblichen Betriebsysteme gleich einfach ausgeführt werden können.
- Der Installationsvorgang sollte keine oder nur ganz wenige Einstellungen von Anwender benötigen, da keine Systemkenntnisse vom Anwender angenommen werden dürfen.

Im folgenden geht es um die Evaluation zwei verschiedener Ansätze zur Distribution einer Software: Java WebStart und Installer.

B.2.1 JavaWebStart

Suns angebotene Lösung heisst JavaWebstart [1]. Per Mausklick auf einen Link im Webbrowser wird die Applikation automatisch vom Internet heruntergeladen und gestartet. Auch werden die Java-Dateien lokal auf der Festplatte zwischengespeichert, so dass beim nächsten Start nicht alles nochmals vom Netz geholt werden muss.

Wir haben folgende Vor- und Nachteile evaluiert:

Vorteile:

- Distribution auf beinahe jeder Plattform möglich.
- Automatisches Update der Anwendung, d.h. dynamische Distribution.

Nachteile:

- Relativ grosser Download nötig.
- JAR-Archiv muss signiert werden. Zertifikatswarnmeldungen vom Browser irritieren den Benutzer.
- Kein Zugriff auf relative Dateipfäde.
- Bedienung vom JavaWebstart-Container ist für einen Durchschnittsanwender zu umständlich.

B.2.2 Installer

Eine Alternative zu JavaWebstart ist ein einfacher Installer, der unsere Software auf dem Betriebsystem des Anwenders einrichtet und startet.

Einen eigenen Installer zu schreiben kommt aus Zeitgründen und wegen mangelnder Erfahrung auf den verschiedenen Betriebsystemen nicht in Frage. Es bleibt der Ausweg mit einem kommerziellen Installer. Das beste Produkt für unsere Zwecke ist InstallAnywhere [14].

Wir haben folgende Vor- und Nachteile evaluiert:

Vorteile:

- Unser Programm läuft als echte Applikation, d.h. keine Zertifikate für Festplattenzugriff nötig.

- Sehr einfache Installation für Endbenutzer.
- Mitinstallation von Java möglich.

Nachteile:

- Kommerzieller Installer kostet Geld.
- Keine dynamische Distribution möglich.

B.2.3 JavaWebstart vs. Installer

Eine Software kann eine noch so durchdachte Bedienung haben, wenn die Installation bzw. Distribution nicht reibungslos vonstatten geht wird der Endanwender sie meistens nicht einsetzen können. Darum hat der Kunde sehr viel Wert auf dieses Element gelegt, was uns zwang diesen Punkt so früh wie möglich abzuklären.

Wir haben uns nach langem Abwägen der Pro und Kontra[2] für den kommerziellen Installer entschieden. Trotzdem haben wir beide Lösungen für das Demo implementiert um den Kunden entscheiden zu lassen. Schliesslich ist der Installationsvorgang entscheidend für die Präsentation der Software.

B.3 Netzwerkverbindung mit Proxy

Die erstellten XML Dateien sollten schliesslich über das Internet via HTTPS an einen Server vom BAG geschickt werden. Der BAG-Server liegt hinter einem Reverse-Proxy, der bestimmte Daten abändert - Pfäde und URLs - und als Authentifizierung Zertifikate verlangt.

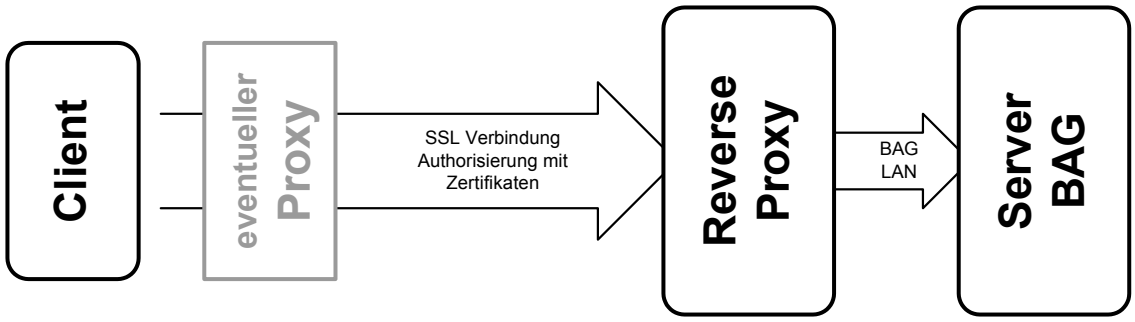

Abbildung B.1: Internetverbingung zum BAG

Da noch nie jemand eine HTTPS-Verbindung mit Java geschrieben hat - von Zertifikaten ganz zu schweigen - beschlossen wir zuerst einige Hilfsklassen für eine normale HTTP-Verbindung zu einem ganz gewöhnlichen Server zu schreiben. Und erst dann diese mit HTTPS und Zertifikaten zu erweitern. Dazu siehe das nächste Untertkapitel.

Wir implementierten ein einfaches Interface, welches ein GET und ein POST einer Datei zu einem HTTP-Server erlaubt. Falls vorhanden, musste die Verbindung über einen Proxy umgeleitet werden. Auch sollte ein Proxy nach Möglichkeit automatisch erkannt werden. Das automatische Erkennen

[2] swiki/19.html (auf CD)

funktionierte jedoch nicht fehlerfrei und wurde zurückgestellt. Als Resultat hatten wir Hilfsklassen für eine HTTP-Verbindung über einen bekannten Proxy.

B.4 HTTPS mit Zertifikaten über Reverse Proxy

Die Implementation einer Verbindung über das HTTP-Secure Protokoll wird von Java bereits angeboten. Leider nicht mit Zertifikats-Authentifizierung. Zudem war die Unterstützung des Microsoft-Zertifikatsformats vom Proxy mehr oder weniger mangelhaft:

- Java unterstützt nur Java Key Store (JKS) und X.509 kodierte Schlüssel vollständig. PKCS#12 Schlüssel, wie wir den privaten Schlüssel des Benutzers vom BAG kodiert bekamen, werden nur lesend unterstützt, und dies auch nur wenn der Schlüssel im Base64 Format vorlag.
- Die `ThrustManager`-Klasse von Java, die dann öffentlichen Schlüssel vom Reverse-Proxy als gültig anerkennt, will alle öffentlichen Schlüssel ausschliesslich im JKS Format.

Die HTTPS-Verbindung mussten wir selber zusammenbasteln: Mit eigenen Thrust- und KeyManager eine `SocketFactory` Klasse von Java initialisieren um damit einen `SSLSocket` zu erzeugen. Nach explizitem Aufrufen des Handshakes mit dem Proxy konnten wir dann die Verbindung einer HTTPSConnection weitergeben und dann wie gewohnt das ganze unseren HTTP-Hilfsklassen (siehe Abschnitt B.3 Netzwerkverbindung mit Proxy) übergeben um Dateien vom und zum BAG-Server zu transferieren.

Anhang C

Abkürzungen

API Application Programmer Interface

BAG Bundesamt für Gesundheit

CRC Class Responsibility Collaborator

CVS Concurrent Versions System

DOM Document Object Model

ESE Einführung Software Engineering

GUI Graphical User Interface

HTTP Hypertext Transfer Protocol

HTTPS HTTP over SSL

IAM Institut für Informatik und Angewandte Mathematik

JAR Java Archive

JDK Java Developement Kit

JKS Java Key Store

PSE Praktikum Software Engineering

SAX Simple API for XML

SCG Software Composition Group

SSH Secure Shell

UML Unified Modeling Language

XML Extended Markup Language

XP Extreme Programming

Literaturverzeichnis

[1] Java™Web Start. http://java.sun.com/products/javawebstart/

[2] Concurrent Versions System CVS. http://www.cvshome.org/

[3] ICQ. http://web.icq.com/

[4] What Is Wiki. http://wiki.org/wiki.cgi?WhatIsWiki

[5] Apache Ant. http://jakarta.apache.org/ant/

[6] WinCVS. http://www.wincvs.org/

[7] The Source for Java™Technology. http://java.sun.com/

[8] Extreme Roles. http://c2.com/cgi/wiki?ExtremeRoles

[9] Microsoft Visio. http://www.microsoft.com/office/visio/

[10] TogetherSoft. http://www.togethersoft.com/

[11] Together/J, A Review in Progress. http://www.objectsbydesign.com/tools/tj/tj_intro.html

[12] JUnit, Testing Resources for Extreme Programming. http://www.junit.org/

[13] Xerces: XML parsers in Java and C++. http://xml.apache.org/

[14] InstallAnywhere. http://www.zerog.com/