Andrea Herrmann
(AndreaHerrmann3@gmx.de)
ist freiberufliche Trainerin und Beraterin für IT-Management (www.herrmann-ehrlich.de). Sie blickt zurück auf zwanzig Berufsjahre, davon sieben als Beraterin und Projektleiterin, zehn Jahre in der Forschung. Mehr als 100 Fachpublikationen, offizielle Supporterin des IREB, Mitautorin von Lehrplan und Handbuch für die CPRE Advanced Level Zertifizierung in Requirements-Management, Regionalgruppensprecherin der Gesellschaft für Informatik in Stuttgart-Böblingen.

Agiles Requirements-Engineering statt Konzeption?
Wie Sie Ihre agil entwickelte Software mit RE refaktorieren

In der agilen Softwareentwicklung kommt man eine Weile ohne Konzeption aus, da Product Owner und Entwickler eng miteinander kommunizieren. Allerdings sieht man der Benutzeroberfläche irgendwann ihr natürliches Wachstum an. Dieser Artikel stellt eine Fallstudie vor, in der es um eine Praxis-Software geht, in der Patienten- und Behandlungsdaten verwaltet werden. Um ihre Qualität zu verbessern, wurde klassisches Requirements-Engineering (RE) durchgeführt mit einer Ist- und Soll-Analyse.

Ausgangslage der Fallstudie

Bei dieser Praxis-Software war die agile Entwicklung an ihre Grenzen geraten. Man sah der Benutzeroberfläche ihr natürliches Wachstum an, und die Position der Felder auf den Reitern spiegelte eher die Chronologie der Umsetzung als ergonomische Grundsätze wider. Folgerichtig fanden die Mitarbeiter öfter mal nicht den richtigen Reiter oder das richtige Feld. Besonders die Such- und die Eingabemaske wurden oft verwechselt, was dazu führte, dass die vermeintliche Suche in der Eingabemaske keine Ergebnisse erbrachte und der Patient als Dublette neu angelegt wurde. Gerade bei einer solchen Anwendung sind jedoch Dubletten fatal, da dann der Arzt keinen Überblick mehr über den gesamten Behandlungsverlauf eines Patienten hat.

Vor drei Jahren hatte der Arzt, ein Spezialist für Augenkrankheiten, für seinen eigenen Bedarf eine Datenbankanwendung entwickelt, in der er die Adressen seiner Patienten und die durchgeführten Behandlungen verwaltete. Später hatte er dann einen Studenten der Medizininformatik eingestellt, der die Software weiterentwickelte.

Dabei waren sie agil vorgegangen: Der Arzt (als Product Owner) notierte seine Ideen und Bedürfnisse, die ihm während der Benutzung kamen, in einem Ticketsystem und priorisierte sie. Der Student nahm sich jeweils für die nächste Iteration die am höchsten priorisierten Anforderungen vor, setzte sie um, präsentierte sie anschließend dem Arzt zur Abnahme und schulte dann das restliche Personal der Praxis in der Verwendung.

Das hatte gut geklappt. Die Anwendung wuchs und erfüllte die funktionalen Anforderungen der Praxismitarbeiter/innen. Die umfangreiche Datensammlung unterstützte nicht nur die Behandlung der Patienten, sondern erlaubte auch wissenschaftliche statistische Auswertungen, mit denen der Arzt auf Expertenkonferenzen Erfolge feierte.

Andere Ärzte fragten nach den Kosten für die Lizenz der Anwendung. Doch vor der Kommerzialisierung musste nun unbedingt eine Überarbeitung der Benutzeroberfläche erfolgen. Wenn selbst die langjährigen Mitarbeiter, deren Bedienungserfahrung mit dem System gemeinsam gewachsen war, Benutzungsfehler machten, dann erfüllte diese Software eindeutig nicht das Kriterium der einfachen Erlernbarkeit. Darum wurde dann ich damit beauftragt, mir die Software genauer anzusehen und Verbesserungsvorschläge zu erarbeiten.

Iterative Verwahrlosung

Die agile Community kennt diesen Effekt der iterativen Verwahrlosung der Softwarequalität und empfiehlt darum ein regelmäßiges Refactoring. Dabei bezieht

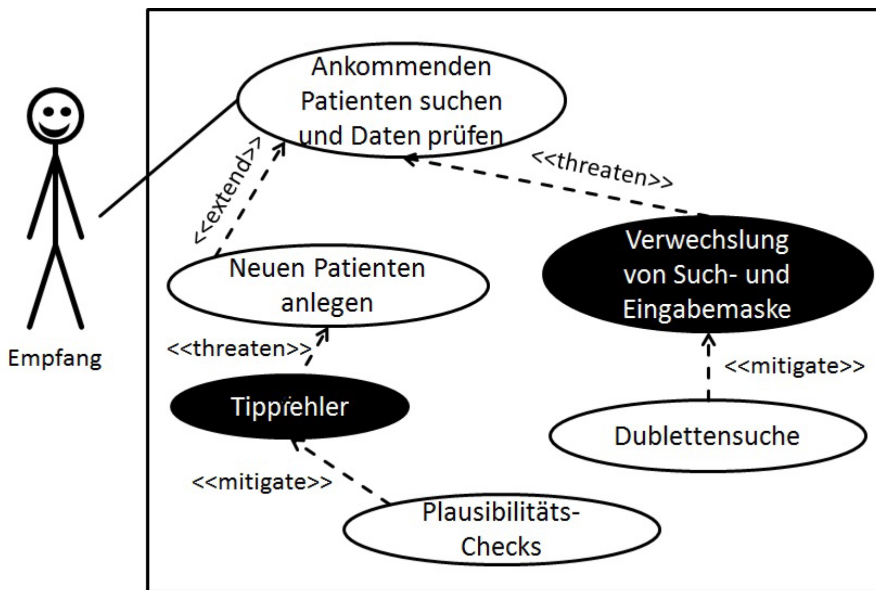

Abb. 1: Ausschnitt aus dem Use-Case-und-Misuse-Case-Diagramm

man sich meist auf den Code. Doch auch die Benutzeroberfläche erleidet dasselbe Schicksal, ebenso wie Software, die nicht agil, sondern durch mehrere Wasserfallprojekte hintereinander entsteht. Laut den Lehman'schen Gesetzen [LR97] erhöhen sich während der Softwarewartung beziehungsweise -weiterentwicklung der Umfang, die Komplexität und die Entropie (=Unordnung), während gleichzeitig die Qualität sinkt, falls man nicht ausdrücklich etwas dagegen unternimmt.

Diese Phänomene sind zu Recht in Form von Naturgesetzen formuliert. Je öfter man eine Software erweitert oder ändert, umso größer wird die Gefahr, dass sie sich entweder von ihrem ursprünglichen Design entfernt oder auch das frühere Design zu den neuen Anforderungen nicht mehr passt. Und ändern und wachsen muss Software, um in einer dynamischen Welt nützlich zu bleiben, auch das besagen die Lehman'schen Gesetze.

Im Folgenden beschreibe ich das Vorgehen, die Ergebnisse und was man aus dieser Fallstudie lernen kann. Obwohl vor allem die Benutzerfreundlichkeit und die Datenqualität erhöht werden sollten, sollte als Grundlage zur Schwachstellenanalyse immer auch eine Analyse der funktionalen Anforderungen durchgeführt werden. Dabei wurde wie folgt vorgegangen: Zunächst erfolgte eine Ist-Analyse der Benutzergruppen, Prozesse und Architektur, und anschließend eine Misuse-case-basierte Analyse der Qualitätsanforderungen, gefolgt von der Erarbeitung und Priorisierung von Verbesserungsvorschlägen. Am Ende stand dann ein Aktionsplan in Form von neuen, priorisierten Einträgen im Product Backlog, sodass die Verbesserung direkt in die agile Arbeitsweise integriert werden konnte.

Mein Ansprechpartner für alle Fragen war der Entwickler, der diejenigen Fragen, die er nicht selbst beantworten konnte, an die anderen Stakeholder weitergab, nämlich Arzt, Mitarbeiter im Empfang und Krankenschwester.

Ist-Analyse

Das Ziel der Ist-Analyse war es, eine Grundlage für die Schwachstellen- und Ursachenanalyse zu legen. Das System war bisher nicht als Handbuch dokumentiert. Da die Benutzerschar der Anwendung aber gering und der Einsatzzweck klar begrenzt war, konnte innerhalb weniger Stunden das System nachdokumentiert werden.

Benutzer- und Stakeholder-Analyse: Insgesamt gab es vier Benutzergruppen, die in diesem Fall durch jeweils eine einzige Person vertreten waren:

- Der Empfang, der Termine vereinbarte,
- der Arzt, der sich vor der Behandlung anhand der Daten einen Überblick über die Behandlungshistorie verschaffte, und
- die Krankenschwester, die hier die durchgeführten Untersuchungen und deren Ergebnisse sowie die verschriebenen Medikamente dokumentierte.
- Hinzu kam noch der Programmierer, der gleichzeitig der Administrator war und regelmäßig Datenbereinigung betrieb, zum Beispiel die entstandenen Dubletten suchte und zu einem einzigen Datensatz zusammenführte.
- Die fünfte Stakeholdergruppe, die Patienten, benutzten das System nicht selbst, hatten aber natürlich ebenfalls eigene Erwartungen an die Qualität der Abläufe in der Praxis.

Für jede Stakeholdergruppe wurde erhoben, welche Kenntnisse und Erfolgskriterien sie haben, welche Aufgaben sie durchführen und welche Use Cases (Anwendungsfälle) des Systems sie dazu verwenden.

Prozessanalyse: Die Abläufe in der Praxis wurden dokumentiert. Anschließend erfolgte eine Analyse der Use Cases, wobei in einer Use-Case-Vorlage vor allem der Aktor (=die ausführende Benutzergruppe) und Schritt für Schritt die Szenarien beschrieben wurden. Die Use-Case-Analyse erfolgte durch eine Demonstration der Arbeitsabläufe am Computer. Hier wurden auch schon Fragen nach Fehler- und Sonderfällen gestellt und erste Usability-Risiken entdeckt, zum Beispiel doppeldeutige Beschriftungen oder ungeschickte Platzierungen von Feldern.

Analyse der Qualitätsanforderungen

Bei der Analyse der Qualitätsanforderungen erfolgten Ist- und Soll-Analyse gleichzeitig. Es wurde eine MOQARE-Analyse durchgeführt (beschrieben beispielsweise in [HP08]). Dabei wurden ausgehend von den Geschäftszielen „gute Behandlung der Patienten" und „korrekte Daten für wissenschaftliche Auswertungen" als die beiden wichtigsten Qualitätsattribute der Software die Benutzerfreundlichkeit und die Integrität der Daten identifiziert. Auch andere Qualitätseigenschaften wurden als wichtig erkannt, nämlich der Datenschutz, doch dieser steht im Folgenden nicht im Fokus der Beschreibung.

Als Nächstes wurden Misuse Cases [SO05] identifiziert, das heißt, schädliche Anwendungsfälle, die man vermeiden möchte oder zumindest ihre Auftretenswahrscheinlichkeit verringern, beziehungsweise deren negative Folgen abfangen. Diese waren zumeist Sonder- und Fehlerfälle der Use Cases, bei denen die Benutzer die nötige Funktionalität nicht fanden oder falsch verwendeten. Somit waren die Use Cases eine gute Ausgangslage für diese Analyse. Bei jedem einzelnen Schritt des Use Case konnte man sich fragen, was hier schief gehen könnte bezüg-

lich Benutzerfreundlichkeit und Datenintegrität.

Die bereits erwähnte Verwechslung von Such- und Eingabemaske stellte einen solchen Misuse Case dar. Für den Use Case des Empfangs „Ankommenden Patienten suchen und Daten prüfen“ wurde die falsche Maske verwendet, nämlich die Maske für das Anlegen eines neuen Patienten statt der Suchmaske. So führte die Suche zu keinem Ergebnis, und eine Dublette wurde angelegt.

Die Datenbank enthielt auch viele unplausible Daten wie ein Geburtsjahr 2085 oder 1005. Diese entstanden durch Tippfehler im Use Case „neuen Patienten anlegen“.

Das Ziel war nicht die vollständige Erfassung aller potenziellen Schwachstellen, sondern da das System bereits in Benutzung war, konnten wir uns auf die tatsächlich aufgetretenen Misuse Cases beschränken. Die Qualitätsanalyse einer bereits im Einsatz befindlichen, existierenden Software erwies sich als unvergleichlich einfacher als die spekulative Diskussion über Qualitätsanforderungen eines noch nicht existierenden IT-Systems am grünen Tisch. Man kann sich bei einer Usability-Analyse viele Schwachstellen denken, aber manche davon umgehen die Benutzer mit schlafwandlerischer Sicherheit, während sie umgekehrt unerwartete Schwierigkeiten haben können.

Hier nun lagen aber genügend Erfahrungen und Daten vor, um sofort prüfen zu können, ob und wie oft ein Benutzbarkeitsproblem tatsächlich auftritt. Fragen während dieser Analyse lauteten beispielsweise: „Haben die Benutzer Schwierigkeiten, dieses Feld zu finden?“ „Wie viele Dubletten werden pro Woche angelegt?“ „Wie viele Dubletten entstehen durch Tippfehler im Namen des Benutzers?“ „Wie oft wird ein unplausibles Geburtsjahr eingetragen?“

Jeder Misuse Case wurde risikobasiert priorisiert, das heißt, die Häufigkeit und der entstehende Schaden bezüglich der Geschäftsziele auf einer Skala von 1 bis 10 geschätzt. Häufigkeit und Schaden wurden multipliziert, und das sich so ergebende relative Risiko erlaubte eine klare Priorisierung der Misuse Cases. Hierbei gehörten alle Misuse Cases, die Dubletten erzeugen, zu den am höchsten priorisierten, weil sie erschreckend häufig auftraten und schlimmste Folgen haben konnten, wenn sie Behandlungsfehler verursachten. Die schlimmste Folge aufgrund eines Systemfehlers wäre die Erblindung eines Patienten gewesen.

Erarbeitung und Priorisierung von Verbesserungsvorschlägen

Die relevantesten Misuse Cases mit den höchsten Risikowerten wurden einer weiter gehenden Ursachenanalyse unterzogen. Landeten wir bei wiederholtem „Warum“-Fragen schließlich bei einer Schwäche der Benutzeroberfläche, so war eine konkrete neue Softwareanforderung entstanden.

Zur Priorisierung dieser Gegenmaßnahmen wurde noch abgeschätzt, wie weit diese den zugehörigen Misuse Case verhindern oder dessen Wahrscheinlichkeit oder Schaden mindern kann. War ein völliges Verhindern möglich, erhielt die Anforderung als Prioritätswert den Risikowert des Misuse Case, bei einer Risikominderung den entsprechend anteiligen Wert. Verringert beispielsweise eine Gegenmaßnahme die Wahrscheinlichkeit eines Misuse Case um die Hälfte, ohne am Schaden etwas zu ändern, so war die Priorität gleich dem halben Risikowert des Misuse Cases.

Die bisher schon praktizierte Dublettensuche durch Skripte und anschließendes manuelles Zusammenführen von Dubletten war zwar vermutlich recht effizient und konnte ca. 90 Prozent der Dubletten finden. Als alleinige Anti-Dubletten-Maßnahme genügte sie uns jedoch nicht, da Vorbeugen vor Heilen geht und die manuelle Datenbereinigung recht viel Aufwand verursachte.

Wir definierten darum für möglichst viele Eingabefelder entweder Eingabehilfen (z. B. Wertelisten, wo noch nicht vorhanden) oder Plausibilitätsprüfungen. Die erlaubten Geburtsjahre wurden auf 1900 bis heute begrenzt und das Geburtsdatum musste zwingend ein festgelegtes Format haben. Zu diesen Gegenmaßnahmen führte uns der Misuse Case, dass ein Patient am Empfang bereits beim Anlegen des Datensatzes mit Tippfehlern registriert wurde, was später sein Auffinden verhinderte (Das selbst entwickelte System hatte leider keine Schnittstelle zum Kartenleser für die Versichertenkarte).

Tippfehler im Namen sind leider nicht zu verhindern, aber auch das Geburtsdatum erwies sich als wichtig. Hatte der Patient einen häufigen oder komplizierten Namen, diente als Suchbegriff nämlich nicht sein Name, sondern das Geburtsdatum. War das Geburtsjahr falsch eingetippt oder fehlten im Geburtsdatum die Trennpunkte, wurde der Patient nicht gefunden und eine neue Dublette entstand. Solche Tippfehler waren sehr, sehr häufig, weil der Empfang während der Dateneingabe üblicherweise mit den Patienten sprach und eventuell noch durch einen Telefonanruf unterbrochen wurde.

Außer Softwareanforderungen wurden auch Anforderungen an den Arbeitsprozess, an den Entwicklungs- und Wartungsprozess und an die Schulung der Mitarbeiter identifiziert, was bei so einer ganzheitlichen Analyse dazugehört. Am Ende stand dann ein Aktionsplan in Form von neuen, priorisierten Einträgen im Product Backlog, sodass die Verbesserung direkt in die agile Arbeitsweise integriert werden konnte. Auch Maßnahmen wie die Migration der bereits eingegebenen Geburtsdaten in das nun verpflichtende Format können in der agilen Entwicklung ja als Backlog Item eingeplant werden, nicht nur Softwareanforderungen.

Lessons Learned

In dieser Fallstudie spielten klassisches Requirements-Engineering und agile Entwicklung konstruktiv zusammen. Für das Refactoring der Benutzeroberfläche war es sinnvoll, in regelmäßigen Abständen, die von der Entwicklungsgeschwindigkeit abhängen, den Ist-Zustand zu dokumentieren und eine fachliche und technische Konzeption zu erstellen, aus der sich dann wiederum Anforderungen herleiten, die in der Folge agil implementiert werden können. Dieses Wechselspiel aus agilem Vorgehen und klassischem Requirements-Engineering ist wichtig und hilfreich, um die Softwarequalität sicherzustellen und zu erhalten. Ohne regelmäßige Gesamtbegutachtung von System und Konzeption entfalten die Lehman‘schen Gesetze bei der agilen Entwicklung ihre volle Wirkung.

Diese „erweiterte Retrospektive“ war in diesem Fall besonders nützlich, weil der Entwickler alleine das Entwicklungsteam stellte und ihm so ein Diskussionspartner für Retrospektiven fehlte. Der Product Owner machte am Iterationsende einen Produkt-Review, gab also nur Rückmeldung zur Umsetzung der Anforderungen, die er gestellt hatte. Die agile Entwicklung setzt aber eigentlich ein Entwicklungsteam und intensive technische, konzeptionelle und prozessbezogene Diskussionen voraus.

Die in der Fallstudie erstellten Konzepte stellten nicht nur eine Momentaufnah-

me der Funktionalität und Produktqualität dar, sondern können bei einer wiederholten Retrospektive als Baseline dienen. Unter anderem war geplant, zur Vorbereitung der geplanten Produktvermarktung auch ein Benutzerhandbuch und ein Administratorhandbuch zu schreiben. Die in der Anforderungsanalyse identifizierten Beschreibungen von Benutzerrollen, Arbeitsprozessen, Use Cases und Systemarchitektur konnten hierfür als Grundlage dienen. Umgekehrt hätten bei Vorliegen solcher Handbücher diese die Grundlage zur Schwachstellenanalyse darstellen können.

Die Methoden des klassischen Requirements-Engineering, des Usability-Engineering und der Softwarearchitektur müssen durch die agile Entwicklung nicht automatisch aus der Mode kommen, sondern können diese gut ergänzen. Statt Konzepte vor der Entwicklung hypothetisch zu erstellen und damit eventuell an der Realität vorbei zu planen, kann man in der agilen Entwicklung Konzepte jederzeit dann erstellen, wenn sie nötig werden. Requirements-Engineering-Methoden unterstützen auch die Retrospektive und das Refactoring in der agilen Entwicklung. ■

Literatur & Links

[HP08] A. Herrmann, B. Paech, MOQARE: Misuse-oriented Quality Requirements Engineering, in: Requirements Engineering Journal, Vol. 13, No. 1, Jan. 2008, S. 73-86

[LR97] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, W. M. Turski, Metrics and Laws of Software Evolution - The Nineties View, in: Proc. of the 4th International Symposium on Software Metrics, METRICS ,97, IEEE Computer Society Washington, S. 20-32

[SO05] G. Sindre, A. L. Opdahl, Eliciting security requirements with misuse cases, in: Requirements engineering Journal, Vol. 10, No. 1, Jan. 2005, S. 34-44