Agile Softwareentwicklung und Architekturplanung

Ich weiß was ich tue!

Leon Fausten

Hochschule für angewandte Wissenschaften Hamburg
Master Informatik - Grundseminar
Email: leon.fausten@haw-hamburg.de

I. Motivation und Problemstellung

Für ein erfolgreiches Softwareprojekt ist eine geeignete Architektur notwendig. Eine Software Architektur umfasst eine Reihe von signifikanten Entscheidungen, über die Organisation einer Software, inklusive der Strukturelemente aus der die Anwendung zusammengebaut wird. Anforderungen über Eigenschaften, wie Performance, Wiederverwendbarkeit, Wartbarkeit, Skalierbarkeit und weitere sind Faktoren, die durch die Wahl einer geeigneten Software Architektur beeinflusst werden können. [1] Diese Architektur sollte mit der restlichen Anwendung schrittweise innerhalb des Entwicklungsprozesses entstehen und angepasst werden. Derzeit entstehen die meisten Architekturen ohne eine bewusste Planung, durch individuelle Designentscheidungen der Entwickler, während der Entwicklungszeit. Grady Booch nennt diese Architekturen passenderweise *"Accidental Architecture"*[2]. Dadurch schleichen sich viele verschiedene Designstile ein, die nicht immer gut miteinander harmonisieren und die Zieleigenschaften nicht miteinbeziehen.[2]

Für die Softwareentwicklung wird heutzutage zum Großteil ein agiles Vorgehen verwendet. Wie eine jährlich durchgeführte Studie der Firma Version One herausgefunden hat, steigt der Einsatz agiler Praktiken weiterhin. Im Jahr 2013 haben 88% der an der Umfrage teilnehmenden Firmen agile Entwicklungsmethoden eingesetzt. Vom Jahr 2011 bis zum Jahr 2012 steigerte sich der Einsatz von 80% auf 84% [3]. Die weite Verbreitung zeigt, dass die Agile Softwareentwicklung ein wichtiges Vorgehen mit Zukunft ist.

A. Das Agile Manifest

Im Agilen Manifest existieren keine konkreten Aussagen, wie eine passende Architektur entwickelt wird. Stattdessen sind eher sich Widersprechende Punkte zu finden. Zwischen diesen muss eine geeignete Balance gefunden werden. Die Prinzipien

> "The best architectures, requirements, and designs emerge from self-organizing teams."

und

> "Simplicity–the art of maximizing the amount of work not done–is essential."

sind schwer zu vereinen, da es schwer, bis unmöglich ist eine Software möglichst einfach zu gestalten und gleichzeitig die bestmögliche Architektur umzusetzen. Oft sind bei der Umsetzung Umwege notwendig, um gewisse Architekturstile durchzusetzen, die die Implementierung allerdings komplexer machen. Ein weiteres Beispiel dazu sind die Punkte

> "Continuous attention to technical excellence and good design enhances agility."

und

> "Working software is the primary measure of progress."

Dadurch, dass die Lauffähigkeit der Software am wichtigsten ist, wird es schwierig vollständig auf die technische Exzellenz und gutes Design zu achten. [4]

B. Problemfälle

Für ein erfolgreiches Projekt ist es wichtig, dass die Architektur zum richtigen Zeitpunkt in der richtigen Art entwickelt wird. Folgende exemplarische Beispiele zeigen typische Fehler, die bei der Architekturentwicklung gemacht werden können.

1) Ausschließlich initiale Planung: Im Rahmen des Projektes sollte eine Verwaltungssoftware für spezielle Abteilungen in niederländischen Krankenhäusern entwickelt werden. Das Projekt wurde mit sieben Entwicklern und einem Architekten über einen Zeitraum von zwölf Monaten durchgeführt. Das Team hat Scrum mit einem zweiwöchigen Sprint Zyklus verwendet. Zusätzlich hat ein weiteres Team von sieben Personen einen speziellen Teil des Backends entwickelt. Dadurch entstand eine starke Abhängigkeit zwischen den Teams.
In einer ersten Phase entwickelte der Architekt in Zusammenarbeit mit dem Management des Unternehmens eine generische, wiederverwendbare Architektur. Diese sollte auch in einer anderen ähnlichen Anwendung verwendet werden. Das Entwicklerteam wurde lediglich über die Entscheidungen informiert. Die Bedenken ihrerseits wurde allerdings weitestgehend ignoriert. In der zweiten Phase hat das Projektteam die Verantwortung übernommen. Die Entwickler haben die Anwendung ohne Beachtung der zuvor getroffenen Entscheidungen entwickelt. Der Architekt wurde über die neu getroffenen Entscheidungen informiert, hat sich aber nicht mehr für das Projekt verantwortlich gefühlt. Dadurch, dass die Verbindung zum anderen Entwicklerteam weiterhin bestand, war die Validierung der Entscheidungen verzögert. Trotzdem wurde der Entscheidungszyklus erheblich verkleinert.[5]
Dieses Beispiel aus der Praxis, zeigt deutlich, dass eine rein initiale Planung nicht sinnvoll ist. Hier wurde versucht ein Phasenmodell mit einem agilen Modell zu verbinden. Es ist

deutlich zu erkennen, dass die Kommunikation zwischen den Entwicklern und dem Architekten vollständig gescheitert ist. Die initiale Planung war überflüssig und hat nur zusätzliche Mehrkosten verursacht, da die getroffenen Entscheidungen von den Entwicklern nicht umgesetzt werden konnten. Neben den fachlichen und technischen Schwierigkeiten kamen persönliche dazu, da sich das Projektteam nicht reichlich einbezogen gefühlt hat.

2) Keine Architekturplanung: Wenn keine Planung durchgeführt wird, ist das Risiko groß, dass das Projekt scheitert oder nur durch nachträgliche Ausbesserung mithilfe von Refaktorisierung gerettet werden kann. Da Scrum keine explizite Architekturplanung vorsieht, dürfte dies in den meisten Fällen vorkommen. Der Erfolg hängt dann maßgeblich von der Erfahrung der Entwickler ab.

3) Nachträgliches Ausbessern durch Refaktorisierung: Refaktorisierung ist bei agilen Vorgehen fest vorgesehen, allerdings nicht in jedem Fall vollständig einsetzbar. Dabei spielt es keine Rolle, ob eine initiale Architekturplanung durchgeführt wurde oder diese ohne vorherige Planung entstanden ist.

In einem Beispiel hat ein Entwicklerteam während der Entwicklung bemerkt, dass die Software nicht mehr den Ansprüchen bezüglich der Kapazität und Performance entsprach und bei den Vorgesetzten um eine Refaktorisierungsmöglichkeit gebeten. Das Projektmanagement war zufrieden, dass das Team auf gute Codequalität wert legt und willigte die gewünschte Refaktorisierung ein. Die Arbeit dauerte allerdings erheblich länger als erwartet, weshalb die Kosten und die Zeit des Projektes erheblich überschritten wurden. Einer der Hauptgründe für die daraus folgende Unzufriedenheit des Managements war ein unterschiedliches Verständnis von Refaktorisierung.

> ”Refactoring isn’t limited to code detail but can range up to the larger scale of a system’s software architecture.”

[6] Dieses Beispiel zeigt, dass zwischen Code und Architektur Refaktorisierung unterschieden werden muss. Code Refaktorisierung beschränkt sich dabei unter anderem auf Fehlerkorrekturen und z.B. dem Auslagern von Methoden. Diese Änderungen sollen unter anderem für eine bessere Wartbarkeit sorgen und die Komplexität verringern.

Größere Änderungen, die sich über mehrere Komponenten oder logische Schichten ziehen, werden als Architektur Refaktorisierung bezeichnet. Durch diese Art der Refaktorisierung können Eigenschaften, wie Skalierbarkeit, Performance, Testbarkeit und auch Robustheit erreicht werden. Architektur Refaktorisierung kann dadurch, dass sie mehrere Teile der Anwendung betrifft sehr aufwendig werden. Außerdem kann sie im Gegensatz zur Code Refaktorisierung die Arbeit anderer Entwickler einschränken. Diese können ihre eigentliche Arbeit an betroffenen Komponenten nicht fortführen, bis die Refaktorisierung abgeschlossen ist. Dadurch ist ein Fortschritt im Projekt schwer festzustellen, da keine neuen Features hinzukommen. [5] [7] Dieses Beispiel zeigt, das Refaktorisierung alleine nicht die Lösung für das Architektur Problem ist. Wenn keine regelmäßigen Reviews und die daraus resultierenden Refaktorisierungen durchgeführt werden, werden die benötigten Änderungen schnell unübersichtlich und beanspruchen viel Zeit. Refaktorisierung ist jedoch nicht überflüssig, da sich Anforderungen schnell Ändern können und auch mit vorheriger Planung fehlerhafte Entscheidungen getroffen werden können. Es ist ist aber wichtig, dass diese so früh wie möglich durchgeführt werden.

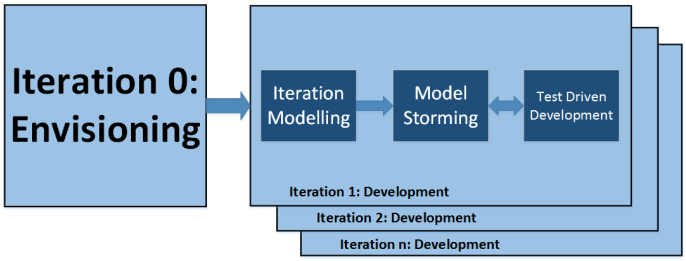

Abbildung 1. AMDD Livecycle

II. Vorstellung vorhandener Lösungsansätze

Im diesem Abschnitt wird der Lösungsansatz durch *Agile Model Driven Development* vorgestellt. Außerdem wird erleutert, wie *Pattern* und *Frameworks* die Problemlösung unterstützen können.

A. Agile Model Driven Development

Beim Agile Model Driven Development (AMDD) wird die Modellierung und Agilität vereint. Die Architektur wird schrittweise grafisch modelliert und als Grundlage für die Codegenerierung verwendet. Der Livecycle dieses Vorganges ist in Abbildung 1 zu sehen. Vor Beginn der Entwicklungs-Zyklen wird eine Initialisierungsphase durchlaufen. Diese Phase wird auch als Envisioning bezeichnet. Beim Envisioning werden die initialen Anforderungen ermittelt und eine anfängliche Architektur Vision entworfen. Diese anfängliche Phase kann unterschiedlich lange, je nach Aufwand des Projektes, dauern. Es sollte aber nicht zu viel Zeit eingeplant werden, da sonst die Gefahr eintritt zu detailliert zu modellieren.

Anschließend wird die Entwicklung iterativ durchgeführt. Teil jeder Iteration ist die Modellierung. Diese ermöglich es, den Aufwand des Durchlaufes zuverlässig schätzen zu können, um die Arbeit planen zu können. Model Storming wird erst durchgeführt, wenn es notwendig ist. Beim Model Storming werden erst beim Auftreten eines Problems Lösungsansätze gesucht. Stakeholder nehmen an diesem Abschnitt aktiv dran teil. Es werden nur Eigenschaften betrachtet, die aktuell wichtig sind und nicht erfüllt werden. Zukünftig wichtige Eigenschaften sollen erst beim Eintreffen betrachtet werden. Das Model Storming und Test Driven Development laufen wechselseitig. Beim Test Driven Development (TDD) wird die Anwendung nach den zuvor entwickelten Tests umgesetzt. Wenn beim TDD Probleme auftreten, wird das Model Storming durchgeführt. Die Entwicklung muss nicht zwingend durch TDD erfolgen, wird aber empfohlen. TDD vereinfacht es, die Konzentration bei der Entwicklung nur auf das aktuelle Teilproblem zu lenken, statt auf das große Ganze. Durch das regelmäßige Model Storming sind allgemeine Reviews innerhalb jeder Phase nur optional erforderlich und können nach Bedarf durchgeführt werden. [8]

Beim ursprünglichen Model Driven Development (MDD) steht das Model im Mittelpunkt, bei der agilen Entwicklung stattdessen lauffähiger Code. Zwischen beiden Punkten muss

eine Balance gefunden werden, die beides zufriedenstellend vereint.

Probleme, die in MDD vorhanden sind, treten ebenfalls beim AMDD auf. Dazu gehört unter anderem, dass die Generierung des Codes nicht vollständig möglich ist. Um die gewünschten Ziele zu erreichen muss deshalb manuell nachgebessert und erweitert werden. Da generierter Code meist nicht besonders lesbar für Menschen ist, ist dies teilweise sehr mühsam. Bei Quellcode, der manuell Eingefügt oder Bearbeitet wird, ist es bei einer Model-Anpassung Problematisch, diesen zu übernehmen. Durch die Vernachlässigung der Codequalität gehen wichtige Eigenschaften, wie Wiederverwendbarkeit, Portierbarkeit und z.B. Wartbarkeit verloren. Generell ist die Umwandlung von vorhandenem Code zurück zu einem Model problematisch. [9] [10]

B. Architektur-Pattern

Architektur-Pattern sind keine neue Idee, bereits 1996 diskutieren Bushmann u.a. [11] darüber. Pattern wurden als allgemeine Lösungen für ähnliche wiederkehrende Problemsituation entwickelt. Dabei beziehen sie sich nicht auf konkrete Lösungen in einer Programmiersprache, sondern müssen für den jeweiligen Anwendungsfall implementiert werden.

Architekturpattern befassen sich mit der gesamte Anwendung. Der Einsatz von ihnen, kann die Implementierung zum Erreichen einzelnen Qualitäts-Merkmalen sowohl vereinfachen, wie auch aufwendiger gestalten. Zu erreichende Qualitäts-Merkmale können z.B. Performance, Sicherheit, Skalierbarkeit, u.v.m. sein. Einzelne Pattern können sich widersprechen, es kann zum Beispiel schnell vorkommen, dass bei dem Wunsch nach guter Performance, die Sicherheit etwas vernachlässigt werden muss. Da es schon eine Vielzahl von Pattern gibt, ist es nur in Ausnahmefällen sinnvoll neue zu entwickeln. Ältere Pattern haben sich meist bereits mehrfach in der Praxis bewiesen und dadurch gezeigt, wo ihre Schwachstellen und Vorteile liegen und sollten deshalb bevorzugt verwendet werden. [12][11]

Ein sehr bekanntes Architekturpattern ist das *Model-View-Controller* Pattern, bei dem die Anwendung in die drei genannten Schichten aufgeteilt wird. Der Controller verwaltet die Usereingaben und kommuniziert entsprechend mit der View und der Model Schicht. Die View-Schicht sorgt für die Darstellung der Informationen. Das Model bildet das Verhalten ab und setzt die Schnittstelle zu den gespeicherten Daten um. [13]

C. Frameworks

Frameworks sind bereits allgemeine vorimplementierte Grundgerüste, die die Entwicklung unterstützen und beschleunigen. Sie sind in einer speziellen Programmiersprache entwickelt und setzen bereits einige Pattern um, dadurch geben sie eine Basis Architektur für die Anwendung vor. Viele hilfreiche Funktionalitäten, die die Sprache von Haus aus nicht bietet werden von einem Framework mitgeliefert.

Da Frameworks von drittanbietern Entwickelt werden, bieten sie den Vorteil, dass die Anbieter oder eine Community sie im Normalfall auch wartet und weiterentwickelt. Durch eine breite Community bietet sich der zusätzliche Vorteil, dass die Unterstützung bei Problemen einfacher ist.

Die meisten Frameworks bieten außerdem eine Art Plugin-System an, durch das weitere Funktionalitäten durch Fremdbibliotheken hinzugefügt werden können.

III. Architekturplanung in Scrum

Diese Arbeit konzentriert sich auf die Architekturentwicklung im agilen Framework Scrum. Es ist das am weitesten verbreitete agile Vorgehen. Beim Einsatz von agilen Methoden wurde 2013 bei 73% der Firmen Scrum oder ein Scrum ähnliches Framework eingesetzt.[3] Scrum macht Vorschläge zu Rollen, Meetings und der Aufgabenteilung. Über die Architekturentwicklung ist allerdings nichts im offiziellen Scrum Guide beschrieben. [14]

In diesem Abschnitt werden verschiedene Gesichtspunkte bei der Architektur Findung beschrieben. Es geht darum einen Zeitpunkt zu finden, wann Entscheidungen getroffen werden sollen, wer den Vorgang leiten soll und wie dies am besten durchgeführt werden kann. Die Modellierung soll mithilfe einer Anwendung geschehen, notwendige Eigenschaften werden hier diskutiert und vorgestellt. Außerdem ist es wichtig die Entscheidungen entsprechend zu dokumentieren, ohne das dies mit zu viel Aufwand verbunden ist. Da alle Entscheidungen hinsichtlich eines speziellen Zieles getroffen werden, ist es wichtig, diese regelmäßig zu prüfen, um festzustellen, ob die Entscheidungen korrekt sind oder sich die Ziele eventuell geändert haben.

A. Wer? - Die Rolle eines Softwarearchitekten

Für eine effiziente Architekturentwicklung ist eine zusätzliche Rolle für einen Softwarearchitekten, bzw. eine Softwarearchitektin sehr hilfreich. Im Unterschied zum ursprünglichen Softwarearchitekten begleitet dieser das Projekt über die gesamte Zeit, nicht wie früher, mit nur einer initialen sehr ausführlichen Planungsphase. Der Architekt ist dabei der Besitzer aller Architekturentscheidungen. Bei den Entscheidungen muss eine Balance zwischen Qualität, den Sorgen der Stakeholder und der Anforderungen gefunden werden. Er ist aber ein normales Teammitglied, wie die anderen Entwickler, ohne jegliche Weisungsbefugnisse seinen Kollegen gegenüber. Wenn allerdings Uneinigkeit bei der Entscheidungsfindung, ohne Aussicht auf eine Einigung herrscht, kann er im Ausnahmefall bestimmen, welche Entscheidung umgesetzt wird. Der Softwarearchitekt kann z.B. einer der Entwickler mit dieser zusätzlichen spezial Rolle sein. Dies bedeutet für den Architekten, dass er gleichzeitig auch mit programmiert. Um korrekte Entscheidungen treffen zu können muss er einen aktuellen technischen Wissensstand besitzen. Zusätzlich ist auch ein entsprechendes Fachwissen der Projekt Domäne empfehlenswert. Seine Aufgaben bestehen darin, die initiale Anforderungsaufnahme und Architekturdesignung zu leiten. Dadurch entsteht eine anfängliche Basis-Architektur. Alle getroffenen Entscheidungen werden von dem gesamtem Team beschlossen. Er leitet ebenfalls die Evolution der Architektur. Durch den Gesamtüberblick und seine bereits gesammelten Erfahrungen kann er andere Teammitglieder beraten und coachen. Es ist seine Aufgabe die maximal technische Einfachheit sicherzustellen. Die Dokumentation der Entscheidungen führt er durch. Dadurch sind diese zu einem späteren Zeitpunkt einfacher Nachvollziehbar und Wiederverwendbar. Bei Änderung der Anforderungen kann so schnell Nachvollzogen werden, ob die Entscheidungen noch relevant sind. Durch seine gleichzeitige Rolle als Entwickler weiß er immer über den genauen Projektstand bescheid und kann die getroffenen Entscheidungen selbst validieren. Der Gesamtüberblick über die Software ist für die Entscheidungsfindung besonders

wichtig.[15][8] [5] Die Rolle des Architekten darf nicht zu wenige und nicht zu viele Rechte und Pflichten besitzen. Bei zu wenigen ist der Einflusszu gering und bei zu vielen ist Widerstand vom Team zu erwarten, da er oder sie die meisten Entscheidungen ohne deren Einfluss trifft.[16][17]

B. Wann? - Das Sprint Planning Meeting

Die Arbeit für den nächsten Sprint wird im Sprint Planning Meeting geplant. Bei diesem Meeting ist das gesamte Scrum Team anwesend. Inhaltlich wird in dem Meeting beschlossen, welche Aufgaben mit in den nächsten Sprint genommen werden und wie die Arbeit erledigt werden soll. Das Team arbeitet an dem Verständnis der einzelnen Aufgaben gemeinsam. Nur das Entwicklungsteam wählt die nächsten Aufgaben aus. In der zweiten Phase, nach Auswahl der Aufgaben entscheidet das Team darüber, wie sie die Aufgaben erledigen wollen. Meist beginnt das Entwicklungsteam mit einem Systementwurf und der Festlegung wann welche Aufgaben innerhalb der Sprints erledigt werden sollen. Es ist möglich, weitere Teilnehmer für eine technische oder fachliche Unterstützung zum Meeting einzuladen. [14]

Bei Betrachtung können diese Eigenschaften ebenfalls auf die Architekturplanung bezogen werden. Genauer würde dies in die zweite Phase des Meetings zum Systementwurf passen. Es sollte immer nur so viel modelliert werden, wie für den nächsten Sprint notwendig ist. Für die Modellierung sollten die Stakeholder an dem Meeting teilnehmen. Die Stakeholder können wichtige fachliche Informationen, wie z.B. über die erwartete Last und Anzahl der zu händelbaren Benutzer liefern. Die Architekturentscheidungen sollten begründet festgehalten werden, dadurch kann später festgestellt werden, ob Entscheidungen noch relevant sind, obwohl sich die Anforderungen geändert haben. Die getroffenen Architekturentscheidungen sollten schnell verifiziert werden, indem sie realisiert werden. Um Fehler vorzubeugen ist es empfehlenswert das Model durch externe Personen begutachten zu lassen. Personen, die kein Vorwissen vom Projekt besitzen, können das Model unvoreingenommen betrachten und hinterfragen Eigenschaften, die andere für selbstverständlich halten. (siehe auch III-C) [12] [18]

C. Dokumentation der Entscheidungen

Einer der Hauptpunkte von agiler Entwicklung ist die minimale Dokumentation.

> ”Working software over comprehensive documentation.” [4]

Die Architekturdokumentation muss an die agile Denkweise angepasst werden und sollte nur minimalen Aufwand kosten, ansonsten wird diese schnell vernachlässigt. Die Dokumentation sollte ausreichend transparent, konkret und einfach verständlich sein. [19] Eine Dokumentation kann sehr unterschiedlich durchgeführt werden. Einerseits kann die Dokumentation durch Artefakte in dem umgesetzten Design selbst repräsentiert werden. Das Resultierende Wissen kann persönlich, über Gespräche, oder Formal, durch schriftliche Dokumentation und Modellierung (z.B. UML) geteilt werden. In Tabelle I sind unterschiedliche Arten der Dokumentation aufgelistet. Innerhalb eines Projektes wird es im Normalfall eine Mischung aus mehreren Arten geben.

Bezeichnung	**Beschreibung**
Mandatory Template-based Documentation	Diese Dokumente werden für das Angebot, den Vertrag oder dem Projektplan zwingend benötigt. Es werden Funktionale und Technische Design Elemente beschrieben.
Faculative Template-based Documentation	Die Entstehung dieser Dokumente ist optional. Für die Dokumente stehen Templates bereit. Durch die Templates ist die Erstellung meist zeitaufwendiger, hat aber den Vorteil, dass wichtige Designaspekte nicht vergessen werden.
Ad hoc Documentation	Alle Dokumente, die zufällig ohne ein Template enstehen. Es kann in einer Freigabe/Wiki strukturiert verteilt werden, unstrukturiert per E-Mail oder spontan. Dadurch, dass sie keinem Template folgen sind sie einfacher zu erstellen, erhöhen aber das Risiko wichtige Teile zu vergessen
Meeting Notes	Geschriebene oder anderweitig visualisierte Notizen zur Dokumentation innerhalb eines Meetings. Haben denn Vorteil, dass sie schnell erstellt werden können und die Details eines Meetings schneller in Erinnerung rufen können. Sie sind allerdings manchmal schwer zu verstehen, wenn Personen nicht am Meeting teilgenommen haben.
Direct Communication	Kann in einem Chat, per Telefon oder von Angesicht zu Angesicht durchgeführt werden. Dies ist die einzige Möglichkeit für Bidirectionale-Dokumentation.

Tabelle I. Dokumentationsarten

[8]

Name	**Beschreibung**
Name	Kurzer Name des Beschlusses der als Schlüssel weiterverwendet werden kann.
Status	aktueller Status.
Beschluss Gruppe	Kann mit einer oder mehreren Gruppen mit charakteristischen Gemeinsamkeiten verknüpft werden.
Problem	Beschreibung der Sachverhalte, weshalb eine Entscheidung zwischen einer oder mehreren Alternativen getroffen werden muss.
Beschluss	Das beschlossene Resultat.
Alternativen	Alternativ gefundene Lösungen.
zugehörige Beschlüsse	Alle Beschlüsse, die mit dieser in Verbindung stehen.
zugehörige System Aspekte	Funktionale und nicht-funktionale Anforderungen, Bedingungen, Business Ziele, Annahmen, Risiken, Design Richtlinien, die mit dem Beschluss in Verbindung stehen.
Historie	Historie des beschriebenen Beschlusses, inklusive aller Status Änderungen über vorgeschlagen, beschlossen, zugelassen, ...

Tabelle II. Entscheidungsdokumentation

Hadar u. a. [19] machen einen Vorschlag für eine Dokumentation pro Release, doch wie aktuelle Entscheidungen festgehalten werden erwähnen sie nicht. Van Heesch u.a [20] haben einen für diesen Anwendungszweck vielversprechenderen Ansatz. Sie stellen ein Framework zur Dokumentation von Architekturentscheidungen innerhalb einer vorgegebenen Tabelle vor. Alle wichtigen Felder in solch einer Entscheidungsdokumentation sind in Tabelle II zu sehen. Die Beschreibungen sollten so formuliert werden, dass allen Stakeholder und Teammitglieder diese schnell und einfach verstehen können. Dieser Ansatz hat den Vorteil, dass er durch die vordefinierten Felder relativ kurz und übersichtlich gehalten werden kann.

D. Wie? - Wie kann eine technisch Unterstützung aussehen?

Eine technische Unterstützung zur Modellierung, Visualisierung und Dokumentation ist hilfreich. Die Ergebnisse der Meetings können dadurch direkt festgehalten werden. Gleichzeitig kann die Software schrittweise durch die Planung

leiten. Entscheidungen können dadurch zu einem späteren Zeitpunkt einfacher nachvollzogen werden.

Bei Meetings mit technischer Unterstützung ist sowohl die Visualisierung, wie auch die Interaktion mit der Technik relevant. Bragdon u.a. [21] behandeln hauptsächlich die Interaktion mit einem digitalen Whiteboard über Touch und Gesten. Deren technischer Aufbau sieht ein digitales Whiteboard für Toucheingaben, sowie zwei Kinnect Kameras vor. Eine für Gesten der sitzenden Teilnehmer und eine für Gestern, die direkt vor dem Whiteboard getätigt werden. Sie bieten außerdem die Möglichkeit mit jedem digitalen Gerät interagieren zu können. Die Idee war explizit für Entwickler-Meetings konzipiert. Der Aufbau soll hierbei etwas vereinfacht werden um die Realisierung zu vereinfachen und die Anzahl der Störquellen zu vermindern. Die Interaktion über Touch mit dem Whiteboard soll erhalten bleiben. Die Gestenerkennung vor dem Whiteboard wird weggelassen und nur für Teilnehmer am Meetingtisch angewendet. Die Interaktion über weitere Geräte fällt ebenfalls weg.

Eine Interaktionsmöglichkeit für Teilnehmer, die nicht vor dem Whiteboard stehen ist hilfreich, um z.B. einen virtuellen Zeiger zu realisieren, um zu verdeutlichen über welchen Teil auf dem Bildschirm aktuell geredet wird und die Möglichkeit diesen zu selektieren. Bragdon hat ein paar nützliche Design Prinzipien für die Gesten vorgestellt, die auch zum Teil hier nützlich sind. Ein Punkt davon ist die Möglichkeit, für jeden Teilnehmer, mit der technischen Installation zu interagieren. Die Gesten sollen so gewählt werden, dass sie irrelevant zur aktuellen Anzeige immer die gleiche Bedeutung haben und dadurch einfacher zu erlernen und merken sind. [21] Es ist wichtig, dass das System auf einem Rechner läuft, der nicht als Notebook, an einem Platz steht. Dies hat den Vorteil, das nicht die Person mit dem Notebook automatisch die Leitung des Meetings übernimmt und dadurch die Hauptinteraktion bestimmt.

Neben der Interaktion ist die Visualisierung ein wichtiges Kriterium für die Verwendbarkeit der Anwendung. Zur Vereinfachung wird in dieser Arbeit nur die Darstellung der Architekturplanung berücksichtigt, nicht die der vorherigen Sprint Planning Aktionen. Für die Darstellung soll eine einfach verständliche Diagrammart gewählt werden. Ein Diagrammtyp der nützlich ist, ist das Deploymentdiagramm. In einem Depolymentdiagramm werden die Hauptbestandteile physikalischen Maschinen zugeordnet. Die Hauptbestandteile werden auch Artefakte genannt. Sie können zum Beispiel die Resultate nach dem Kompilieren darstellen. Die Kommunikation wird über Kanäle dargestellt. Die Kommunikationskanäle werden inklusive Protokoll angegeben. Um die Aufteilung der Anwendung in Komponenten zu visualisieren sind Komponentendiagramme geeignet. Die Komponenten werden mit ihren Beziehungen untereinander dargestellt. Bei dieser Art von Diagrammen kann jede Komponente wieder aus anderen Komponenten bestehen. Dadurch ist es möglich nur einen groben Überblick zu bekommen, ohne die Existenz unterliegender Komponenten kennen oder modellieren zu müssen.

UML Diagramme haben den Vorteil, dass sie auch ohne Vorkenntnisse einfach zu verstehen sind. Dadurch ist es für nicht technisch versierte Stakeholder einfacher den Aufbau zu verstehen. Bei der Modellierung sollte eher auf Verständnis, statt vollständig korrekte Syntax geachtet werden.

Die Diagramme sollen vollständig selbst modelliert werden. Eine automatische Generierung soll nicht geschehen, da diese Diagramme dadurch oft sehr unübersichtlich sind und Informationen modelliert werden, die für die aktuelle Bearbeitung irrelevant sind.

E. Architektur-Review

Getroffene Architekturentscheidungen sollten schnellstmöglich evaluiert werden. Wenn Entscheidungen nicht korrekt sind müssen diese meist durch Refaktorisierung angepasst werden. Dies kann unter Umständen sehr teuer und zeitaufwendig sein. Je später die Fehlentscheidung entdeckt wird, desto teurer und aufwendiger wird es diese zu korrigieren. Eine Möglichkeit eine Architektur zu überprüfen ist ein *Pattern-basiertes Architektur Review* (PBAR).

Das PBAR ist auf agile Vorgänge, mit kurzen Zeiträumen, häufigen Änderungen und kleinen Teams angepasst. Qualitätsansprüche, wie z.B. Performance, betreffen die gesamte Anwendung und müssen deshalb wiederholt getestet werden. Die Reviewer sollten nicht Teil des Teams sein, aber Fachwissen zu Architekturen, Architektur-Pattern, Qualitäts-Merkmalen und ein Grundwissen der Fachdomäne besitzen. Am Review-Meeting nehmen alle Entwickler und interessierte Stakeholder teil. Vor dem Meeting sollten die Reviewer alle Architektur Dokumente und Anforderung studiert haben. Innerhalb des Meetings werden anfänglich die wichtigsten Qualitäts-Merkmale besprochen. Anschließend wird die allgemeine Systemarchitektur besprochen und auf einem Whiteboard skizziert. Anschließend identifizieren die Reviewer die darin enthaltenen Pattern. Im nächsten Schritt prüfen die Teilnehmer, ob die identifizierten Qualitäts-Merkmale sich mit jedem Pattern vereinbaren lassen. Im letzten Schritt werden die Qualitäts-Merkmale, die Probleme bereiten, nicht verwendete Pattern, die helfen können, mögliche Konflikte zwischen Pattern, identifiziert und diskutiert. Für die identifizierten Probleme werden anschließend Lösungsstrategien erarbeitet. Nach dem Meeting sollte eine Zusammenfassung über die Ergebnisse von den Reviewern erstellt werden. [12]

Wenn dieses Meeting regelmäßig durchgeführt wird, z.B. nach jedem Sprint können Probleme schnell entdeckt werden. Die regelmäßige Durchführung bringt ebenfalls den Vorteil, dass die benötigte Dauer sinkt. Die Änderungen müssen nur noch im Detail betrachtet werden, die gesamte Anwendung nur noch im Allgemeinen.

IV. Ziele für das Projekt

Die in dieser Arbeit vorgestellte Möglichkeit der Architekturplanung bei Scrum soll im Rahmen des Projektes detaillierter ausgearbeitet werden. Die einzelnen Punkte werden genauer auf ihre Schwachstellen untersucht. Für die Schwachstellen sollen mögliche Lösungsansätze erarbeitet oder Alternativen gesucht werden. Der Vorgang soll die Komponenten für die Architekturplanung, die Dokumentation und die Evaluierung enthalten. Am Ende des Projektes soll ein Prototyp als technische Unterstützung entstehen. Die Anwendung soll auf Basis der Jira Agile[1] Anwendung von Atlassian entstehen. Der Prototyp wird als Plug-in entwickelt und als Grundlage für eine praktische Evaluation innerhalb eines Unternehmens dienen.

[1] https://www.atlassian.com/software/jira/agile

V. Arbeitsplan

Am Anfang des Projektes soll ein genauer Vorgang erarbeitet werden. Dabei sollen die einzelnen Schritte genauer spezifiziert und untersucht werden, wie die Interaktion mit dem System gut ablaufen kann. Da die resultierende Anwendung das gesamte Sprint Planning Meeting abbilden soll, sind auch die nicht direkt mit der Architekturplanung in Verbindung stehenden Aktivitäten für die Umsetzung relevant. Dazu zählt unter anderem auch die Auswahl der Aufgaben und das zugehörige Schätzen der Aufwände.

Im zweiten Schritt wird sich mit der Jira Agile Anwendung genauer Beschäftigt, um zu analysieren, welche Funktionalitäten von Haus aus unterstützt werden und welche noch mit dem Plug-in entwickelt werden müssen. In diesem Abschnitt werden bereits vorhandene Plug-ins analysiert und bewertet. Außerdem wird untersucht, wie die Pluginentwicklung durchgeführt werden kann.

Anschließend sollen aus dem Vorgang die notwendigen Anforderungen für eine softwaregestützte Lösung erarbeitet werden. Diese Anforderungen werden schrittweise umgesetzt. Alternativen zum Whiteboard, wie zum Beispiel ein interaktiver Tisch, sollen als alternative Installation untersucht werden. Ein Teil davon ist es die *Human Computer Interaction* (HCI) zur Bedienung des Systems. Die genauen Interaktionsmögichkeiten werden entwickelt und auf praktische Tauglichkeit untersucht. Für die entfernte Interaktion wird ein Set von allgemein gültigen Gesten erarbeitet, die unterstützt werden sollen.

Sobald ein funktionsfähiger Prototyp entstanden ist, soll dieser praktisch evaluiert werden. Es soll dadurch die Anwendung und der dadurch abgebildete Prozess geprüft werden. Die Evaluation soll im Rahmen eines praktischen Einsatzes innerhalb einer Entwicklungsfirma stattfinden. Dadurch soll geprüft werden, ob die Anwendung praktisch einsetzbar ist und weitere Optimierungsmöglichkeiten aufdecken. Aktionen, die in einem täglichen Einsatz störend oder überflüssig sind sollen identifiziert werden. Die Testphase soll ebenfalls aufdecken, ob zusätzliche Aktionen und Funktionalitäten notwendig sind. Die zusätzlichen Funktionalitäten können als Grundlage für zukünftige Entwicklungen und Erweiterungen verwendet werden.

Jedes Projekt wird unterschiedlich abgewickelt. Dies kann verschiedene Gründe haben, eventuell weil es sich um unterschiedliche Teammitglieder handelt, die Projektziele unterschiedlich sind oder die Firmen, in denen die Arbeit durchgeführt wird unterschiedlich sind. Für die Unterschiede muss herausgefunden werden, wie weit sich diese auf den Vorgang auswirken und ob die Software individuell angepasst werden muss.

VI. Risiken

Es besteht die Gefahr, dass der resultierende Vorgang zu komplex wird. Dabei kann sowohl der Vorgang selbst Probleme bereiten, wie auch die umgesetzte Anwendung.

Bei einem zu komplexen Vorgang wird dieser schwer zu verstehen und unter Umständen sehr Zeit und Kostenintensiv in der Ausführung. Dies kann zur Folge haben, dass er nur unvollständig bearbeitet werden kann und wichtige Bestandteile, wie z.B. die Dokumentation oder die regelmäßigen Reviews ausgelassen werden.

Die Anwendung leitet nur durch die Architektur Modellierung, die Architektur-Evaluation wird dabei nicht abgebildet. Deshalb kann es schnell passieren, dass die Architektur Reviews ausgelassen werden.

Ein weiteres Risiko ist die technische Umsetzung, wenn diese für unterschiedliche Projektteams angepasst werden muss, ist es notwendig, die Anwendung sehr detailliert konfigurierbar aufzubauen. Dies würde die Implementierung erheblich aufwendiger machen. Bei einer zu starren Umsetzung besteht die Gefahr, dass diese vom Nutzerteam nicht verwendet wird.

Die Bedienung kann ebenfalls Probleme bereiten. Wenn die Gesten nicht richtig erkannt werden, kommt es schnell zu Fehleingaben oder die Teilnehmer des Meetings werden abgelenkt, weil zeitweise unerwartete Dinge auf dem Bildschirm passieren. Vor der Entwicklung muss geprüft werden, ob es Probleme bei der Erkennung mit vielen Personen im Raum gibt und wie die maximale und minimale Entfernung dieser von der Kamera Aussehen darf.

Für die Evaluation muss eine spezielle technische Ausstattung mit digitalem Whiteboard und einer möglichen Gestenerkennung aus Entfernung vorhanden sein. Für die Installation muss ein entsprechend ausgestatteter Raum zur Verfügung gestellt werden. Da Firmen diese Ausstattung in den meisten Fällen anschaffen müssen, ist dies mit zusätzlichen Kosten und weiterem Aufwand verbunden.

VII. Schnittstellen zu anderen Projekten

Das Projekt wird gemeinsam mit den Kommilitonen Alexander Kaack und Erwin Lang gestartet. Alle Themen konzentrieren sich um das agilen Umfeld. Im Allgemeinen versuchen alle Themenbereiche die Zusammenarbeit des Projektteams und der Stakeholder zu optimieren. Trotz des gemeinsamen zentralen Themas ist eine genaue Abgrenzung der Themengebiete möglich.

Alexander Kaack konzentriert sich auf die Stakeholder Awareness. Im genauen bedeutet dies, die verbesserte Einbeziehung der Stakeholder und die Bereitstellung eines optimierten Informationsflusses zwischen Team und Stakeholdern. Dies hat den Hintergrund, dass diese unter anderem einen Überblick über die Entwicklungsdauer und Kosten bekommen und Schwachstellen schneller identifizieren können. Eine dauerhafte Stakeholder Awareness hat den Vorteil, dass Probleme schneller erkannt und behoben werden können.

Erwin Lang beschäftigt sich mit der Retrospektive von Scrum. In dieser wird nach jedem Sprint die Arbeitsweise diskutiert (Das Wie, nicht das Was). Dabei geht das darum, gute und nicht so gute Ereignisse des vergangenen Sprints zu diskutieren. Für negative Einflüsse werden Vorschläge zur Verbesserung gesammelt. Im Endeffekt, soll den Teilnehmern deutlicher werden, welche Probleme im vergangen Sprint aufgekommen sind und wie diese verhindert werden können.

In dem hier besprochenen Teil des Projektes geht es um die aktive Planung einer Softwarearchitektur im Rahmen eines agilen Vorgehens. Die bewusste Planung soll das Erreichen nicht funktionaler Ziele, wie Skalierbarkeit vereinfachen, ohne dass aufwändige Anpassungen nach der Entwicklung notwendig sind. Diese soll vorzugsweise im Rahmen des Sprint Plannings angestoßen werden.

REFERENZEN

[1] P. Kruchten, “Software Architecture and Agile Software Development —An Oxymoron?” 2009. [Online]. Available: http://www.zuehlke.com/fileadmin/pdf/events/vortraege/LAT-Agile_Presentation_1.pdf

[2] G. Booch, “The Accidental Architecture,” IEEE Software, vol. 23, no. 3, May 2006, pp. 9–11. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1628932

[3] Version One, “8th Annual State of Agile,” Tech. Rep., 2014. [Online]. Available: http://www.versionone.com/pdf/2013-state-of-agile-survey.pdf

[4] M. Fowler and J. Highsmith, “The agile manifesto,” Software Development, vol. 9, 2001, pp. 28–35. [Online]. Available: http://www.pmp-projects.org/Agile-Manifesto.pdf

[5] M. A. Babar, A. W. Brown, and I. Mistrik, Agile Software Architecture - Aligning Agile Processes and Software Architectures. Elsevier, 2014.

[6] F. Buschmann, “Gardening Your Architecture, Part 1: Refactoring,” IEEE Software, vol. 28, no. 4, Jul. 2011, pp. 92–94. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5929528

[7] M. Di Stefano, “Code Refactoring vs Architecture Refactoring — Refactoring Ideas,” 2013. [Online]. Available: http://www.refactoringideas.com/code-refactoring-versus-architecture-refactoring/

[8] S. W. Ambler, “Agile Model Driven Development (AMDD),” Xootic Magazine, 2007, pp. 13–21. [Online]. Available: http://www.agilemodeling.com/essays/amdd.htm

[9] R. Matinnejad, “Agile Model Driven Development: An Intelligent Compromise,” in 2011 Ninth International Conference on Software Engineering Research, Management and Applications. IEEE, Aug. 2011, pp. 197–202. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6065641

[10] Y. Zhang and S. Patel, “Agile model-driven development in practice,” IEEE Software, vol. 28, 2011, pp. 84–91. [Online]. Available: http://www.profs.iffca.edu.br/~rosana/Pos-gradua\%E7\%E3o/artigosMDEMDDMDAsugest\%F5es/8-AgileModel-DrivenDevelopmentinPractice.pdfhttp://dl.acm.org/citation.cfm?id=1957377.1957535\&coll=DL\&dl=GUIDE\&CFID=603902315\&CFTOKEN=83843853

[11] F. Bushmann, R. Meunier, and H. Rohnert, “Pattern-oriented software architecture: A system of patterns,” John Wiley&Sons, vol. 1, 1996, p. 476.

[12] N. Harrison and P. Avgeriou, “Pattern-Based Architecture Reviews,” IEEE Software, vol. 28, no. 6, Nov. 2011, pp. 66–71. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5661759

[13] Microsoft Developer Network, “Model-View-Controller,” 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/ff649643.aspx

[14] K. Schwaber and J. Sutherland, “The scrum guide,” Scrum. org, October, 2013. [Online]. Available: http://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-US.pdf\#zoom=100

[15] J. F. Hoorn, R. Farenhorst, P. Lago, and H. Van Vliet, “The lonesome architect,” in Journal of Systems and Software, vol. 84, 2011, pp. 1424–1435. [Online]. Available: http://dl.acm.org/citation.cfm?id=2009817

[16] R. Malan and D. Bredemeyer, “Less is more with minimalist architecture,” IT Professional, vol. 4, 2002.

[17] P. Kruchten, “What do software architects really do?” Journal of Systems and Software, vol. 81, 2008, pp. 2413–2416.

[18] C. Miyachi, “Agile software architecture,” ACM SIGSOFT Software Engineering Notes, vol. 36, no. 2, Mar. 2011, p. 1. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1943371.1943388

[19] I. Hadar, S. Sherman, E. Hadar, and J. J. Harrison, “Less is more: Architecture documentation for agile development,” in 2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2013 - Proceedings, 2013, pp. 121–124. [Online]. Available: http://ieeexplore.ieee.org/xpl/login.jsp?tp=\&arnumber=6614746\&url=http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6614746

[20] U. Van Heesch, P. Avgeriou, and R. Hilliard, “A documentation framework for architecture decisions,” Journal of Systems and Software, vol. 85, 2012, pp. 795–820. [Online]. Available: http://dl.acm.org/citation.cfm?id=2148467

[21] A. Bragdon and R. DeLine, “Code space: touch+ air gesture hybrid interactions for supporting developer meetings,” in Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces, 2011, pp. 212–221. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2076354.2076393$\backslash$nhttp://dl.acm.org/citation.cfm?id=2076393