mehr zum thema:
www.agilearchitect.org
www.agilemodeling.com/

THESE II:

ENTKOPPELTE SYSTEME SIND ÄNDERBARE SYSTEME

die autoren

Martin Lippert
(E-Mail: martin.lippert@it-agile.de)

Stefan Roock
(E-Mail: stefan.roock@it-agile.de)

Wir beschreiben hier, wie ein langfristig leicht änderbares System in einem agilen Projektkontext entwickelt werden kann. Dabei spielen Verständlichkeit, Redundanzfreiheit und Entkopplung eine zentrale Rolle. Passende Architekturstile sorgen dafür, dass das System im Ganzen konsistent bleibt.

Die Grenzen der Vorhersagbarkeit

Auch agile Projekte leiden mitunter daran, dass die Kosten für neue Features mit wachsender Systemgröße stark steigen: ein Symptom von Strukturdefiziten in Architektur und Code. Diese Defizite treten dann gehäuft auf, wenn sich nicht geplante Änderungen am System ergeben. Aber gerade diese unvorhergesehenen Änderungen sind ein Prinzip der agilen Softwareentwicklung. Sie erlauben es uns, im Laufe der Entwicklung zu lernen und letztendlich aus fachlicher Sicht deutlich bessere und passendere Systeme zu realisieren. Wie lösen wir dieses Dilemma auf?

Eine Lösung könnte darin bestehen, auch bei einer agilen Vorgehensweise weiter in die Zukunft zu schauen als nur bis zur nächsten Iteration. Schließlich können wir doch aus unserer Erfahrung und einer ungefähren Projektplanung vorhersehen, was in der nächsten Iteration wahrscheinlich kommen wird. Das erscheint zunächst sinnvoll, um weniger Änderungsaufwand für die folgenden Iterationen zu produzieren.

Wenn so tatsächlich geringere Aufwände entstehen, ist das natürlich ein sinnvolles Vorgehen. Allerdings sollte man überlegen, ob man damit möglicherweise nur an den Symptomen herumdoktert. Meistens ist einer von zwei Gründen ursächlich für eine steigende Aufwandskurve in agilen Projekten:

- Die während der Explorationsphase erstellte Architekturvision ist nicht effektiv oder gar nicht vorhanden.
- Die Entwickler beherrschen den inkrementellen Entwurf (vgl. [Bec04]) nicht und stehen daher immer wieder vor größeren Umbaumaßnahmen.

Im Folgenden gehen wir vor allem der Frage nach, was für einen inkrementellen Entwurf notwendig und zielführend ist.

Letztlich wird eine mögliche Kostenexplosion durch das Vorhersehen von Änderungen auf Ebene der Softwarearchitektur nur verzögert. Selbst wenn ich heute schon zwei oder drei Iterationen weiter in die Zukunft schaue und meine Software auf Basis dieser Spekulation gestalte, kann in der darauffolgenden Iteration eine neue Anforderung meine bisherigen Gedanken über den Haufen werfen und auch mein mehrere Iterationen in die Zukunft geschautes Design zu einem Fall für ein Refactoring machen. Um auf diese Art und Weise eine Kostenexplosion vollständig zu vermeiden, wäre es erforderlich, alle Anforderungen im Voraus exakt zu kennen. Und damit wären wir wieder in der Vergangenheit angekommen: beim Wasserfallmodell.

Der Knackpunkt ist also, nicht zu versuchen, Änderungen zu verhindern oder durch geschicktes Vorhersehen zu umgehen – es hat sich immer wieder gezeigt, dass wir diese Änderungen nicht wirklich vorhersehen oder verhindern können, wenn wir ein lebendiges und weiter entwickelbares System erstellen wollen. Ziel muss es sein, Änderungen so einfach wie möglich durchführen zu können.

Wir gehen also den gleichen Weg wie die testgetriebene Entwicklung (*Test Driven Development, TDD*, vgl. [Wes05]): Wir realisieren nur das, was gerade gebraucht wird und zu den Anforderungen passt – und antizipieren absichtlich *keine* zukünftigen Anforderungen, die wir im Entwurf „schon einmal berücksichtigen". Mit diesem Vorgehen erreicht TDD, dass sich Entwürfe im Kleinen einfach und ohne große Aufwand ändern lassen. Ähnliches benötigen wir auf der Ebene der Architektur: Der Schlüssel zum Erfolg einer inkrementellen oder evolutionären Architektur liegt darin, die Softwarearchitektur leicht änderbar zu gestalten und nicht die Zukunft zu antizipieren.

Flache Aufwandskurve nachhaltig herstellen

Eine langfristig flache Aufwandskurve existiert genau dann, wenn der vorhandene Code die Integration der jeweils anstehenden neuen Anforderungen auf einfache Art und Weise erlaubt. Da man nicht alle noch kommenden Änderungen sinnvoll vorhersehen kann, bedeutet die Integration neuer Features meistens, den existierenden Code zunächst anzupassen. Diese vorbereitenden Umbauten nennt man *Refactoring*: Ein Refactoring ist die Umgestaltung existierenden Codes, ohne sein Verhalten zu ändern (vgl. auch [Fow99]). Für eine flache Aufwandskurve müssen diese Refactorings mit geringem Aufwand durchführbar sein – das ist dann der Fall, wenn sich der existierende Code leicht ändern lässt. Daraus können wir das übergeordnete Qualitätskriterium für inkrementellen Entwurf ableiten:

Qualitätskriterium für inkrementellen Entwurf: Der existierende Code ist zu jeder Zeit leicht änderbar.

In der Vergangenheit hat sich eine Reihe von Kriterien herauskristallisiert, die dabei

helfen, Code leicht änderbar zu gestalten: Code ist leicht änderbar, wenn er *leicht verständlich, redundanzfrei* und *entkoppelt* ist.

Wenn der Code leicht verständlich ist, lassen sich die Stellen, die angepasst werden müssen, leicht identifizieren.

Wenn ein Code viele Redundanzen (Doppelungen) enthält, müssen bei Anpassungen potenziell viele Stellen im Code geändert werden. Hat man beispielsweise in einem Verkaufssystem den Mehrwertsteuersatz (z. B. 16 %) überall einfach hingeschrieben, muss man bei einer Änderung des Mehrwertsteuersatzes (z. B. auf 19 %) alle Vorkommen des Mehrsteuersatzes 16 % finden und ändern. Dabei reicht es nicht, einfach alle Vorkommen von „16 %" automatisch zu suchen und zu ersetzen. Möglicherweise bedeutet das Vorkommen von „0.16" im Quellcode nicht an jeder Stelle den Mehrwertsteuersatz. Dabei ist eine solche Änderung einer einfachen Konstante noch ein leichtes Unterfangen, weil die potenziell zu ändernden Stellen verhältnismäßig einfach aufzufinden sind. Viel schwieriger wird es, wenn Algorithmen (z. B. für den Umgang mit Skonto und Rabatten) dupliziert werden. Die Dubletten entwickeln sich in der Regel über die Zeit leicht auseinander, sodass man mit einer automatisierten Suchfunktion nicht alle Dubletten aufspüren kann. Für den inkrementellen Entwurf gilt also immer noch das, was in der Softwareentwicklung immer schon galt: Redundanzen sind des Teufels. Stattdessen verfolgt man das *DRY*-Prinzip (*Don't Repeat Yourself*).

Allerdings reicht Redundanzfreiheit allein nicht aus. Schließlich kann auch ein redundanzfreies System eng gekoppelt sein. Eng gekoppelte Systeme haben die unangenehme Eigenschaft, dass Änderungen an einer Stelle im Code häufig *Fernwirkungen* (Seiteneffekte) haben. Also muss man auch bei einem eng gekoppelten System oft viele Stellen im Code ändern, um *eine* Anpassung vorzunehmen.

Inkrementeller Entwurf mit testgetriebener Entwicklung

Auf der Ebene einzelner oder weniger zusammenhängender Klassen existieren etablierte Techniken, die helfen, leicht verständliche, redundanzfreie und entkoppelte Entwürfe zu realisieren: TDD ist eine zentrale Technik, die sich in vielen Projekten bewährt hat.

Bei TDD wird nach einem strengen Ablaufplan vorgegangen:

- Red
- Green
- Refactor

Zuerst wird ein Unit-Test geschrieben, zu dem es noch gar keinen Produktivcode gibt. Folglich schlägt der Test fehl (*Red*).

Anschließend wird solange Produktivcode geschrieben, bis der Test erfolgreich durchläuft (*Green*).

Anschließend werden Test und Produktivcode in eine saubere Form gebracht (*Refactor*). Sauber bedeutet hier wieder: verständlich, redundanzfrei und entkoppelt. Dabei wird als wichtiges Feedback-Instrument die Form der Tests verwendet. Tests, die schwer verständlich, unübersichtlich und schwer änderbar sind, gelten als Indiz dafür, dass der Produktivcode schlecht strukturiert ist. Der Produktivcode muss refaktorisiert werden, bis die Tests übersichtlich und leicht änderbar sind. Insbesondere zu stark gekoppelter Produktivcode führt zu unübersichtlichen Tests, die mitunter sogar Infrastruktur-Komponenten wie Datenbanken zum Ablauf benötigen.

Bei den so initiierten Refactorings spielen Entwurfsmuster (vgl. [Gam95]) eine wichtige Rolle. Sie definieren die *Best Practices*, mit denen insbesondere entkoppelte Entwürfe hergestellt werden können. Dabei haben sich Entwurfsmuster herauskristallisiert, die den aktuellen Anforderungen an Entwürfe Rechnung tragen. So gilt das *Singleton*-Muster inzwischen als „Anti-Pattern", also als schlechtes Beispiel, und wurde längst durch *Dependency-Injection* ersetzt.

Theoretisch könnte TDD als Technik ausreichen, um auch große Systeme zu strukturieren. Wir haben aber die Erfahrung gemacht, dass die mit TDD erreichten Strukturen auf einer Makro-Ebene nicht immer ausreichend entkoppelt sind. Außerdem muss man – insbesondere bei weniger erfahrenen Entwicklern und Teams – damit rechnen, dass TDD nicht durchgängig verwendet wird.

In der Praxis gelingt es den meisten Entwicklern, Methoden und Klassen inkrementell zu entwickeln (z. B. mit Hilfe von TDD). Deutlich weniger Entwickler können Komponenten oder ganze Architekturen inkrementell entwickeln. **Abbildung 1** visualisiert unsere diesbezüglichen Erfahrungen.

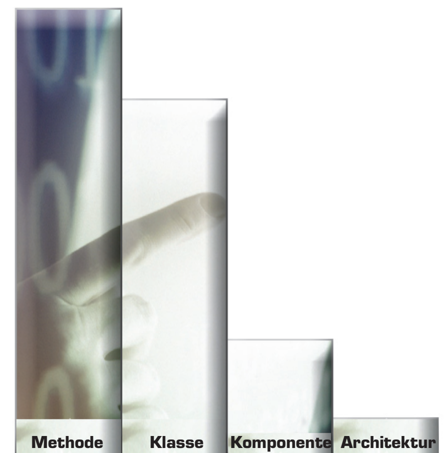

Abb. 1: *Welcher Entwickleranteil beherrscht inkrementellen Entwurf auf welcher Ebene?*

Architekturstil als Rahmen

Die Entwickler benötigen eine Hilfestellung, um Komponenten und Architekturen inkrementell zu entwerfen. Nach unserer Erfahrung kann man tatsächlich einfache Regeln definieren, die – diszipliniert angewendet – das Verklumpen des Systems effektiv verhindern. Einen konkreten Satz dieser Regeln nennen wir einen *Architekturstil*.

Im Gegensatz zu einer konkreten Architektur definiert ein Architekturstil keine konkreten Elemente eines Systems. Vielmehr werden zu Projektbeginn die grundsätzlichen Kategorien und Zusammenhänge zwischen diesen Kategorien einmal definiert – eine Art Architektur-*Spike*, wie wir ihn aus dem *eXtreme Programming (XP)* kennen. Die konkrete Architektur des Systems entsteht im Projekt, wenn der Architekturstil durch die Umsetzung konkreter fachlicher Anforderungen an das System mit Leben gefüllt wird und wenn konkrete Komponenten nach den Regeln des Architekturstils geschnitten und umgesetzt werden. Architekturdiskussionen und Arbeiten an der Architektur werden damit zum integralen Bestandteil der Arbeit an fachlichen Anforderungen und sind keine separaten Tätigkeiten, die speziell eingeplant werden oder für die dediziert Zeit reserviert werden muss.

Im Gegenteil: Die Entwickler sind dafür verantwortlich, jeden Tag eine gute und saubere Architektur im System zu hinterlassen. Geschieht das nicht, beginnen viele Teams damit, neue Anforderungen „irgendwie" in das System „hineinzupressen" und stellen nach einer Zeit fest, dass dadurch größere Arbeiten an der Ar-

chitektur notwendig werden. Letztendlich handelt es sich dabei um Aufräumarbeiten, die durch eben diese Nachlässigkeiten entstanden sind. Teams kommen dann häufig in die Verlegenheit, diese Arbeiten an der Architektur separat einplanen zu müssen und sie vielleicht sogar dem *Product Owner* in die Hand zu drücken, damit dieser ihnen Zeit dafür einräumt – ein hervorragendes Symptom dafür, dass sich das Team nicht konsequent an den Architekturstil gehalten hat und die saubere Struktur der Software über einen längeren Zeitraum vernachlässigt hat.

Solche Situationen können in jedem Team auftreten und sollten explizit gemacht werden. Es darf aber nicht vergessen werden, dass diese größeren Arbeiten an der Architektur ein Symptom sind. Tritt ein solcher Fall einmal auf, muss explizit hinterfragt werden, wie es zu dieser Situation kommen konnte, welche Versäumnisse vorliegen und wie diese in Zukunft verhindert werden können – anstatt die Arbeit mit diesen Symptomen zu einem festen Prozessbestandteil zu machen, ohne die eigentlichen Ursachen zu beseitigen.

Welcher Stil ist der Richtige?

Allerdings ist nicht jeder Architekturstil für den inkrementellen Entwurf geeignet, weil nicht jeder Architekturstil eine gute Entkopplung herstellt. Einem ganz einfachen Architekturstil folgen Systeme nach der so genannten *Drei-Schichten-Architektur*. Diese legt fest, dass Oberfläche, Fachlogik und Datenhaltung zu trennen sind und die Oberfläche nicht direkt auf die Daten zugreift. Leider gibt sie keine Regeln an, wo und wie zu entkoppeln ist. Die Drei-Schichten-Architektur reicht daher als Architekturstil für agile Projekte nicht aus. Ein anderes Beispiel für einen Architekturstil ist *Quasar* (vgl. Kasten „Quasar" auf S. 28 sowie [Sie04]). Obwohl Quasar nicht vor dem Hintergrund des inkrementellen Entwurfs definiert wurde, eignet es sich trotzdem nach unserer Erfahrung sehr gut als Rahmen für den inkrementellen Entwurf. Quasar erzwingt nämlich die Entkopplung „an dem richtigen Stellen": in erster Linie zwischen Technologien und Fachlichkeit.

Andere Architekturstile

Quasar ist aber nur *ein* möglicher Architekturstil. Man kann sich auch selbst einen passenden Architekturstil definieren oder andere existierende Stile adaptieren. Aus unserer Sicht müssen die im Architekturstil definierten Regeln aber auf jeden Fall Entkopplung erzwingen. Nur so können Änderungen an einem System lokal begrenzt werden.

Zusätzlich führt ein klar definierter Architekturstil in der Regel dazu, dass Entwürfe einfacher verständlich sind, weil sie durch den Architekturstil vorstrukturiert werden. Wenn man den Architekturstil im Kopf hat, lassen sich konkrete Architekturen deutlich einfacher verstehen und die Struktur des Systems lässt sich einfacher erkennen. Eine Garantie für eine leichte Verständlichkeit liefern Architekturstile natürlich nicht. Auch mit einem klar definierten Architekturstil lassen sich schwer verständliche Entwürfe produzieren. Aber sie helfen dabei, leicht verständliche Architekturen zu gestalten.

Die Forderung nach Redundanzfreiheit kann bei einem Architekturstil ebenfalls berücksichtigt werden – oder auch nicht. Betrachten wir beispielsweise noch einmal die Drei-Schichten-Architekturen, liefert uns diese zunächst einmal keine Anleitung, wie Redundanzen vermieden werden können, und tatsächlich treten in solchen Systemen schnell Redundanzen auf. Quasar macht die Eliminierung von Redundanzen ebenfalls nicht explizit zum Thema, erleichtert es aber durch Kategorien und erlaubte Beziehungen zwischen diesen Kategorien, dass Redundanzen schneller sichtbar werden. Letztendlich sind natürlich die Entwickler dafür verantwortlich, diese Redundanzen zu entfernen – Quasar garantiert schließlich keine Redundanzfreiheit.

Hilfsmittel für Architekturstile

Um in einem Projekt dauerhaft inkrementelle Architekturen zu leben, müssen die Regeln des gewählten Architekturstils konsequent eingehalten werden. Je häufiger diese Regeln verletzt werden, desto weniger entfalten sie ihre Vorteile (flache Aufwandskurve). Und tatsächlich existieren Werkzeuge, die dabei helfen können, Architekturstile einzuhalten, wenn nicht gar zu überwachen. Die Vielfalt dieser Werkzeuge erstreckt sich von typischen Codeanalyse-Tools (z. B. dem Open-Source-Werkzeug „JDepend" oder „Sotograph" und „SonarJ" von der Firma hello2morrow), über technische Modularisierungshilfsmittel wie „OSGi" (vgl. www.osgi.org), bis hin zu modellgetriebenen Ansätzen, bei denen die Regeln des Architekturstils in den Meta-Modellen definiert werden. Welches dieser Hilfsmittel nützlich und praktikabel ist, entscheidet das konkrete Projekt. Eines dieser Hilfsmittel einzusetzen, hat sich aber auf jeden Fall als sehr hilfreich erwiesen.

Nach unserer Erfahrung ist es sehr nützlich, wenn die Werkzeuge Verletzungen an den definierten Regeln möglichst schnell als Feedback an den Entwickler zurückgeben. Im Idealfall sollte dies direkt beim Speichern in der Entwicklungsumgebung passieren, spätestens aber durch den *Continuous-Integration*-Server – und sichtbar für alle Teammitglieder.

Modellierung zu Projektbeginn

Agile Methoden werfen den klassischen Methoden – zu Recht – „großen Entwurf vorab" (*Big Design Upfront, BDUF*) vor, bei dem zu früh zu viel modelliert wird. Im Projekt stellt sich dann immer wieder heraus, dass die Modelle nicht tragfähig sind. Das bedeutet aber nicht, dass jede Vorab-Modellierung schlecht ist. Wenn Vorab-Modellierung in kurzer Zeit möglich ist, stabile Ergebnisse liefert und relevantes Wissen generiert, ist daran nichts auszusetzen.

Prinzipiell tut man natürlich gut daran, die Dinge früh zu modellieren, die sich später nur sehr schwer wieder ändern lassen. Und solche Dinge gibt es auch in agilen Projekten – dazu gehören z. B. die Systemarchitektur und der Architekturstil.

Die Definition des Architekturstils sollte daher im Projekt definitiv sehr früh erfolgen. Häufig gibt eine erste skizzenhafte Modellierung der Komponenten im Rahmen des Architekturstils dem Team Sicherheit, dass und wie der Architekturstil funktioniert.

In unseren Projekten hat es sich bewährt, ein erstes Fachmodell und einige Kernkomponenten vor Beginn der Programmierung zu modellieren. Dazu haben wir erfolgreich farbige Haftnotizen auf Flipcharts verwendet. Die Flipcharts mit den Modellen blieben im Teamraum an der Wand hängen. Für das Team waren das wichtige Orientierungshilfen, sodass die Entwickler die Diagramme aus eigenem Antrieb aktualisiert haben.

Brauchen wir Softwarearchitekten?

Wir plädieren dafür, die Entscheidung über Architektur und Architekturstil im Sinne der Selbststeuerung dem Team zu überlassen. Insbesondere Teams, die wenig Erfahrungen mit inkrementellen Architek-

turen haben, ist es sinnvoll, einen Experten zur Verfügung zu stellen. Dieser hat allerdings nicht die Aufgabe, sich selbst eine Architektur auszudenken, geschweige denn, Vorgaben für das Team zu machen. Stattdessen muss er dem Team sein Wissen zur Verfügung stellen und gegebenenfalls den Entscheidungsprozess moderieren. Über die Zeit sollte der Experte sich selbst überflüssig machen, indem er sein Wissen und seine Erfahrungen an das Team weitergibt. Wichtig ist dabei aber, dass sich jeder im Team für die Architektur des Systems verantwortlich fühlt und dass jeder sensibel dafür ist, Architekturprobleme zu erkennen und zu beseitigen.

Die Rolle des Softwarearchitekten im Team ist aus unserer Sicht also nicht notwendig.

Fazit

Inkrementeller Entwurf ist eine große Herausforderung für jedes Projekt, egal ob agil oder klassisch. Damit eine langfristig flache Aufwandskurve realisiert werden kann, muss der Code *verständlich, redundanzfrei* und *entkoppelt* sein. Architekturstile definieren die grundlegenden Regeln für das Erstellen der konkreten Architektur während des Projekts. Quasar (vgl. Kasten auf S. 28) ist ein Architekturstil, der besonders viel Wert auf Entkopplung legt und damit den inkrementellen Entwurf sehr gut unterstützt. Das haben wir in den letzten Jahren in mehreren Projekten eindrucksvoll erleben dürfen.

Mit dem Stand der Technik bezüglich Architekturstil, Entwurfsmustern und Technologien ist der Umfang dessen, was sich später nur schwer ändern lässt, immer weiter gesunken. So ist es heute tatsächlich mit überschaubar wenigen Vorab-Festlegungen möglich, in der Folge mit einer flachen Aufwandskurve agil zu entwickeln. Eine explizite Vorbereitung auf einzelne Anforderungen, die in der Zukunft möglicherweise kommen werden, ist in den meisten Softwareprojekten nicht notwendig. Architekturdiskussionen und Arbeiten an der Architektur werden damit zum integralen Bestandteil der Arbeit an fachlichen Anforderungen und sind keine separaten Tätigkeiten.

Der vorgestellte Ansatz liefert keine direkte Antwort darauf, wie mehrere Teams zu synchronisieren sind, die auf demselben Produkt arbeiten oder wie mit einem gemeinsam genutzten Framework umzugehen ist. Dazu sind in der Regel zusätzliche organisatorische Maßnahmen notwendig, auf die wir im Rahmen dieses Artikels nicht eingehen konnten. ■

Literatur

[Bec04] K. Beck, eXtreme Programming Explained. Embrace Change (2. Aufl.), Addison-Wesley Longman, 2004

[Gam95] E. Gamma, R. Helm, R.E. Johnson, J. Vlissides, Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley Longman, 1995

[Fow99] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman, 1999

[Sie04] J. Siedersleben, Moderne Softwarearchitektur: umsichtig planen, robust bauen mit Quasar, dpunkt.verlag, 2004

[Wes05] F. Westphal, Testgetriebene Entwicklung mit JUnit & FIT: Wie Software änderbar bleibt, dpunkt.verlag, 2005