Thomas Rümmler
(thomas.ruemmler@aitgmbh.de)
ist Senior Software Consultant und Projektleiter bei der AIT GmbH & Co. KG. Er begleitet Unternehmen bei der Softwareentwicklung über den gesamten Application Lifecycle hinweg. Schwerpunktthemen seiner Arbeit sind .NET Architekturberatung und Cloud Computing sowie Projektplanung, -controlling & Projektmanagement mit dem Team Foundation Server.

Besser spät als früh: Architekturentscheidungen auf dem Prüfstand

In den meisten Softwareprojekten wird angestrebt, die Architektur möglichst vollständig vorab zu spezifizieren. Doch eine einmal gefällte fundamentale Architekturentscheidung kann ein Korsett sein, aus dem man sich später nur noch mit großer Anstrengung oder gar nicht mehr befreien kann. Alternative Ansätze versuchen daher, architektonische Entscheidungen ganz bewusst zu verzögern und möglichst spät im Verlauf der Entwicklung zu fällen. Die Prämisse hierfür lautet: Je später ich eine Entscheidung treffe, desto mehr Informationen stehen mir zur Verfügung, um sie auf eine solide Basis zu stellen. Obwohl solche Ideen bereits wiederholt proklamiert wurden und auch schon in der Praxis ihre Vorteile bewiesen haben, stoßen sie trotzdem noch häufig auf breiten Widerstand. In diesem Artikel wird gezeigt, was wirklich hinter dieser Idee steckt und für welche Teile eines Softwareprojektes späte Entscheidungen besonders geeignet sind. Des Weiteren wird erläutert, für welche Architekturentscheidungen größerer Aufwand betrieben werden sollte und für welche ein hoher Aufwand keinen signifikanten Wertbeitrag leistet.

Motivation

Hinter jeder Softwareentwicklung steckt eine bestimmte Softwarearchitektur. Die Entscheidung dafür mag bewusst oder unbewusst getroffen sein. Wenn beispielsweise jemand mit einem „Hallo-Welt-Programm“ erste Gehversuche in der Programmierung unternimmt, wird dieses kleine Stück Software bereits auf einer gewissen Architektur aufgebaut. In diesem Falle werden die Architekturentscheidungen wahrscheinlich unbewusst getroffen.

Ganz anders sieht das jedoch bei Industrielösungen aus. Hier werden die Architekturentscheidungen ganz bewusst getroffen, um Ziele, wie Nachhaltigkeit, Wartbarkeit, Skalierbarkeit usw. zu erreichen. Doch wann trifft man die Entscheidungen für die Softwarearchitektur? Was wird dabei eigentlich entschieden?

Die zwei Vorgehensweisen BUFD (Big Upfront Design) und YAGNI (You Ain’t Gonna Need It) stehen für zwei komplett gegensätzliche Ansätze (vgl. [CaB]). BUFD steht für enormen Aufwand zu Beginn des Projektes, obwohl Anforderungen noch entstehen. YAGNI hingegen repräsentiert das komplette Gegenteil. Extreme führen selten zum Ziel. Die Kombination beider Ansätze und selektiver Einsatz dieser unterschiedlichen Bereiche versprechen aber einen deutlich verschlankten und agilen Entwicklungsprozess.

Softwarearchitektur

Die Verwendung von gewissen Mustern in der Softwareentwicklung ist eine Form der Wiederverwendung und damit Effizienzsteigerung. Andererseits ist es aber auch eine Form der Festlegung auf bestimmte Vorgehensweisen: „Architekturprinzipien und daraus abgeleitete Richtlinien sind ein zentrales Instrument, um transparente und nachhaltige Entscheidungen im Architekturbereich herbeizuführen“ [Mau10].

Es gibt jedoch Architekturentscheidungen, die nahezu unwiderruflich sind und andere, die änderbar oder kombinierbar sind. Unwiderruflich bezieht sich dabei darauf, dass Änderungen im Bereich dieser Entscheidungen wirtschaftlich nicht mehr sinnvoll sind, weil der Aufwand nicht mehr in einem sinvollen Verhältnis zum Nutzen steht.

Architekturmuster

Unter Architekturmustern versteht man den Grobentwurf eines Systems. Die OPEN GROUP (siehe [OPG]) definiert Architekturmuster wie folgt:

„Ein Architekturmuster beschreibt die grundlegende Struktur bzw. das Schema für ein Softwaresystem. Es stellt eine Menge an vordefinierten Subsystemen zur Verfügung und definiert deren Zuständigkeiten. Außerdem bezieht ein Architekturmuster auch Regeln und Hinweise für die Organisation der Beziehungen zwischen den Subsystemen mit ein“ (deutsche Übersetzung, originaler Wortlaut siehe [OPG12]).

Es geht bei Architekturmustern also um die fundamentalen und zentralen Entscheidungen. Beispiele dafür sind:

- Schichtenarchitektur,
- Datenstrom und Filter,
- MVC (Model-View-Controller), MVVM (Model-View-ViewModel), ...

Wenn die Entscheidung beispielsweise einmal auf eine eng gekoppelte Schichtenarchitektur gefallen ist, so bedeutet es nach Beginn der Entwicklung einen enormen Aufwand, auf ein lose gekoppeltes System umzusteigen.

Designmuster

Im Gegensatz zu den Architekturmustern stellen Designmuster das Feinkonzept dar. Es geht dabei um Vorgehensweisen zur Lösung eines immer wieder auftretenden Problems. Diese Art der Muster wird durch die OPEN GROUP wie folgt erklärt:

„Ein Designmuser liefert ein Schema zur Verfeinerung der Subsysteme oder Komponenten eines Softwaresystems bzw. der Beziehung zwischen ihnen. Es beschreibt die üblicherweise wiederkehrende Struktur kommunizierender Komponenten, die ein allgemeines Entwurfsproblem innerhalb eines bestimmten Kontextes löst“ (deutsche Übersetzung, originaler Wortlaut siehe [OPG12]).

Designmuster schließen sich nicht gegenseitig aus, sondern können sich gegenseitig ergänzend verwendet werden. Eine einmal getroffene Entscheidung für ein Designmuster kann also leichter geändert werden, als eine Entscheidung für ein Architekturmuster. Beispiele für Designmuster sind:

- Singleton,
- Prototype,
- Composite,
- Observer.

Einflussfaktoren

Wie Architekturentscheidungen im Detail getroffen werden sollten, lässt sich nicht pauschal empfehlen. Dies hängt maßgeblich von den Zielen ab, die man mit der Softwareentwicklung verfolgt. Es gilt, zunächst die Ziele zu definieren und daraus die Architektur abzuleiten. Nachfolgende Beispiele zeigen verschiedene Architekturansätze, die von unterschiedlichen Zielen getrieben sind.

Event-driven architecture

Bei der Event-driven architecture (ereignisgetriebener Architektur) stehen Ereignisse im Zentrum des Geschehens. Die Reaktion auf Ereignisse und deren Verarbeitung ist das Hauptziel dieses Ansatzes. Daraus abgeleitet ergibt sich das Prinzip des CEP (Complex Event Processing, komplexe Ereignisverarbeitung). Diese Technologie dient der Verarbeitung großer Mengen von Ereignissen in Echtzeit (siehe [BaD10]).

Design to cost

Bei Design to cost (kostengetriebener Entwurf) bekommen die Kosten, die zur Laufzeit entstehen, besondere Beachtung. Insbesondere im Zeitalter von Everything-as-a-service, also der IT-Dienste, die mit einem Preisschild versehen sind, bekommt dieser Einflussfaktor wieder mehr Gewicht. Wenn z. B. eine Anwendung auf Basis einer Cloud-Plattform entwickelt wird, die verschiedene Speicherdienste zu unterschiedlichen Abrechnungsmodellen anbietet, muss der Architekt sorgfältiger denn je die Persistenztechnologie auswählen (relationales Datenbanksystem, reiner Binärspeicher zum Ablegen großer Datenmengen ohne Aggregatsfunktionen usw.).

Service-oriented architecture

Serviceorientierte Architekturen zielen auf die Erstellung wiederverwendbarer, abgeschlossener Dienste. Im Idealfall kann man diese koppeln, um Geschäftsprozesse abzubilden und Enterprise-Lösungen zu orchestrieren (siehe [Ros08]).

Risk-driven architecture

Um Fehlverhalten zu vermeiden, gibt es verschiedene Technologien. Risk-driven architecture (risikogetriebene Architektur) konzentriert sich darauf, architekturelle Lösungen für die Aspekte des Systems zu spezifizieren, die einem besonders hohen Risiko ausgesetzt sind. Zur Risikovermeidung werden im Software Engineering-Prozess Verfahren wie Domänenmodellierung, Sicherheitsanalysen und Datenkapselung eingesetzt [FaG10].

Die Qual der Wahl

Unter der Annahme, man hätte unbegrenzt Zeit zur Verfügung, würde man wohl alle Einflussfaktoren berücksichtigen, um ein qualitativ maximales Ergebnis zu erzielen. Doch in der Praxis ist dies nicht erforderlich. Man sollte nach der optimalen Lösung streben. Die schwierige Frage, wo das Optimum eigentlich liegt, ist anwendungsfallabhängig und lässt sich wiederum aus den Zielen und Rahmenbedingungen ableiten.

Eine Software, die nur einmal verwendet wird, z. B. für eine Migration von Daten, erzielt ihr Optimum mit geringerem Aufwand als ein Consumer Product, welches millionenfach verkauft wird.

Beispiel MVVM (Model-View-ViewModel)

Neben der potenziellen Nutzerzahl lässt sich aber auch noch nach dem Teilbereich

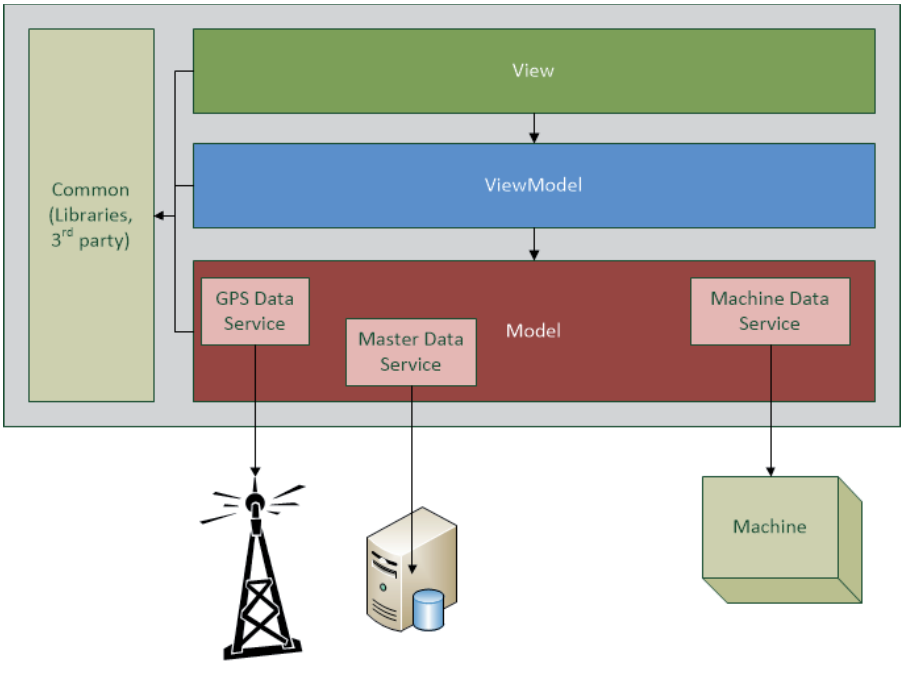

Abb. 1: MVVM Design-Beispiel

der Software unterscheiden. Um dies zu verdeutlichen wird eine Architektur im MV-VM-Muster verwendet (vgl. Abbildung 1).

Wenn man sich für ein Architekturmuster wie MVVM entscheidet, dann verfolgt man damit ebenfalls bestimmte Ziele. Diese können sein:

- Seperation of concern – Aspekte eines Problems werden in Teilprobleme mit eigenen Zuständigkeiten zerlegt (siehe [Vog08]),
- Austauschbarkeit einzelner Schichten,
- Technologieunabhängigkeit der Schichten,
- Kommunikation über wohldefinierte Schnittstellen.

Weiterführende Informationen zu MVVM sind unter anderem in der MSDN zu finden [Mic].

In der Beispielarchitektur aus Abbildung 1 werden verschiedene Datenquellen angebunden. Diese Anbindung ist in der *Model-Schicht* komplett gekapselt. Dazu werden in der logischen Ebene *Model* die verschiedenen Datenzugriffsdienste (hier: GPS Data Service, Master Data Service und Machine Data Service) entwickelt.

Das *ViewModel* ist die intermediäre Schicht zwischen *Model* und *View*. Diese ruft bei Bedarf Methoden des *Models* auf und übernimmt die Umwandlung der Businessobjekte, die das *Model* zurückliefert in eine einfachere Struktur an Eigenschaften, die an die Oberfläche gebunden werden können.

Die *View* – als oberste Schicht dieser Architektur – repräsentiert den Teil der Anwendung, mit der der Anwender interagiert. Meist ist dies eine grafische Benutzeroberfläche.

Unter Berücksichtigung der knappen Resssource Zeit muss die Frage gestellt werden, ob die Architektur aller drei vorher dargestellten Schichten des MVVM-Musters den gleichen Wert darstellt. Eines der oben genannten Ziele ist die Austauschbarkeit von Schichten. Genau an dieser Stelle ist der Schlüssel zu suchen.

Wenn man sich vorstellt, die Beispielanwendung aus Abbildung 1 weiterzuentwickeln, dann ist ein typischer Anwendungsfall die Darstellung der Inhalte in einem neuen User Interface. Diese Bewegung hat die Softwareindustrie in den letzten Jahren immer wieder beschäftigt und auch bei den maschinennahen Systemen sind mittlerweile moderne Benutzeroberflächen eingezogen.

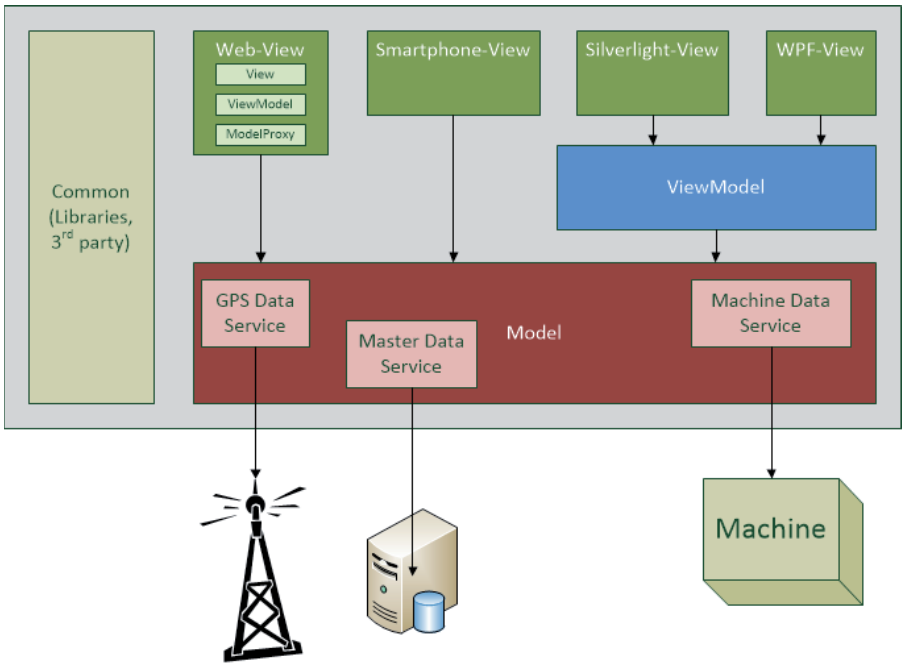

Abb. 2: Weiterentwicklung der Beispielanwendung

Derartige Renovierungsmaßnahmen zielen darauf ab, eine bessere User Experience (Nutzbarkeit, wörtlich: Benutzererfahrung) zu erzeugen. Es geht nicht darum, die Basisfunktionalität maßgeblich zu verändern. Dies wird durch die Trennung der Aufgaben in den einzelnen Schichten vereinfacht.

Eine mögliche Erweiterung der Beispielanwendung ist in Abbildung 2 dargestellt. Es fällt auf, dass der Grad der Veränderung in den Schichten von unten nach oben zunimmt (siehe Abbildung 2).

Im *Model* haben sich gar keine Veränderungen ergeben. Dies beinhaltet nach wie vor die Geschäftslogik und stellt Businessobjekte bereit.

Die Smarphone-View kann eine Anwendung sein, die in einer anderen Technologie entwickelt ist und das *Model* remote als Services nutzt. Die Web-View ist in dem Beispiel ein leichtgewichtiges Frontend, welches lediglich von HTML, CSS und JavaScript lebt. Die in der Abbildung dargestellten weiteren drei Schichten innerhalb der Web-View können durch den Einsatz eines entsprechenden Frameworks, wie z. B. dem JavaScript Framework Knockout [Kno12] erreicht werden. Das *ViewModel* wird deshalb nun von zwei und nicht mehr von allen User Interfaces verwendet.

Bei der *View* sind die größten Veränderungen zu erkennen. Die ursprüngliche Schicht ist durch vier verschiedene Ansichten komplett ersetzt worden. Diese wiederum greifen mehr oder weniger auf die darunterliegenden Schichten zu.

> **Merke:** Je weiter unten die Schicht, desto belastbarer muss deren Architektur sein.

Dabei geht es nicht um die Details der Implementierung. Maßgeblich sind die Schnittstellen über die eine Komponente ihre Funktionalität zur Verfügung stellt. Dies lässt sich am Beispiel von Persistenztechnologien verdeutlichen.

Die konkreten Schnittstellen eines Speicherdienstes im *Model* müssen frühzeitig bekannt und stabil sein, da die darüberliegenden Schichten darauf aufbauen. Die konkrete Auswahl der eigentlichen Technologie zur Datenspeicherung kann hingegen sehr spät im Projekt festgelegt werden, da es sich dabei „nur noch“ um die Implementierung handelt.

In dem beschriebenen Szenario ermöglicht dieses Vorgehen schnelles Vorankommen in der Entwicklung der eigentlichen Geschäftslogik, ohne sich damit aufhalten zu müssen eine geeignete Speichertechnik auszuwählen. Diese Auswahl ist später im Projekt unter Umständen deutlich einfacher, wenn weitere Rahmenbedingungen bekannt sind.

Anlehnung an die Lean-Prinzipien

Die zweite entscheidende Frage ist, wann die grundlegenden Entscheidungen getroffen werden müssen. Hier kann man sich

ein wenig an den Lean-Prinzipien anlehnen. Die Grundideen stammen aus der Fertigung, speziell aus dem Bereich der Automobilbranche. Die beiden Autoren Mary und Tom Poppendieck haben sich jedoch verdient gemacht, diese Ansätze auf die Softwareentwicklung zu übertragen [Pop03]. Teilweise streben diese Prinzipien in die gleiche Richtung wie YAGNI (You ain't gonna need it).

Zwei von den Prinzipien sind an dieser Stelle besonders hilfreich.

Eliminierung von Verschwendung (Eliminate Waste)
Dieser Grundsatz zielt darauf ab, gegen Verschwendung vorzugehen. Verschwendung kann in der Softwareentwicklung ganz unterschiedliche Ursachen haben. Eine falsch getroffene Entscheidung ist Verschwendung. Source Code, der verworfen werden muss, ist Verschwendung. Dokumente, die keinen Nutzen bringen, sind Verschwendung. Features, die keine Verbesserung erzielen, sind Verschwendung.

Es geht also darum, den Nutzen zu erhöhen, indem man sich auf Wertschöpfendes konzentriert. Dazu zählen auch Entscheidungen zum richtigen Zeitpunkt. Dies führt zu einem weiteren Prinzip, nämlich Entscheidungen spät zu treffen.

Entscheidungen so spät wie möglich treffen (Decide as late as possible)
Die auf den ersten Blick vielleicht etwas seltsam anmutende Aussage hat durchaus ihre Berechtigung. Das Requirements Engineering in einem Projekt kann noch so ausgefeilt sein, es wird kein Softwareprojekt geben, in dem es während der Durchführung nicht zu Änderungen kommt. Dies bedeutet, dass sich während der Entwicklung Informationen verändern bzw. neue hinzukommen.

> **Merke:** Je später man eine Entscheidung trifft, desto bessere Informationen hat man zur Verfügung.

An einem Beispiel außerhalb der Softwareentwicklung lässt sich dieses Prinzip wunderbar verdeutlichen: beim Hausbau. Vor Baubeginn müssen natürlich die grundlegende Architektur, der umbaute Raum, die Fensterpositionen usw. festgelegt werden. Es ist jedoch sinnvoll, die Farbe der Fliesen erst später zu bestimmen. Diese sollte auf andere Artefakte, z. B. die Farbe der Küchenmöbel, abgestimmt sein. Diese Information wiederum hat nicht jeder Bauherr zu Beginn parat. Die frühzeitige Entscheidung für eine bestimmte Fliese wäre also Verschwendung in Form einer sich noch einmal ändernden Entscheidung gewesen.

In diesem Vergleich zählen die Wahl der Programmiersprache sowie die Festlegung des Architekturmusters (SOA, MVVM, ...) zur grundlegenden Architektur. Damit wird das Fundament für die Entwicklung gelegt. Auch die Auswahl der Oberflächentechnologie ist zu diesem Zeitpunkt wichtig, um frühzeitig Prototypen liefern zu können. Die Details der Oberfläche hingegen entsprechen eher der Fliesenauswahl und können später erfolgen.

Fazit

Die vorher dargestellten Prinzipien sollten nicht blind verwendet werden. Die Kombination aus späten Entscheidungen verbunden mit dem Bewusstsein, dass weiter „oben" liegende Schichten eher ausgetauscht werden als weiter „unten" liegende, stellt einen guten Ausgangspunkt dar.

Basierend auf diesen Kenntnissen kann man frühzeitig im Projekt die Architektur des *Models* fokussieren. Je weiter man sich jedoch dem User Interface annähert, desto besser ist es, späte Entscheidungen zu treffen und dabei auch stets die Wahrscheinlichkeit der späteren Austauschbarkeit zu berücksichtigen.

Es ist also kein „Schwarz-Weiß-Denken" sinnvoll. Weder frühe noch späte Entscheidungen sind der Schlüssel zu einer erfolgreichen Architektur. Die Wahrheit liegt – wie so oft – in den verschiedenen Abstufungen der Graubereiche dazwischen. ■

Literatur

[CaB] J. O. Coplien and G. Bjørnvig, Lean Architecture: for Agile Software Development.

[Mau10] D. P. Maurer, "Effiziente Architekturentscheidungen durch Architekturprinzipien," Wirtschaftsinformatik & Management, vol. 2010–02, pp. 46–51, 2010.

[OPG12] Open Group, "Architectural Patterns," 2012. [Online]. Available: http://pubs.opengroup.org/architecture/togaf7-doc/arch/p4/patterns/patterns.htm. [Accessed: 15-Oct-2012].

[BaD10] R. Bruns and J. Dunkel, Event-Driven Architecture. Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, p. 241.

[Ros08] M. Rosen, B. Lublinsky, K. T. Smith, and M. J. Balcer, Applied SOA: service-oriented architecture and design strategies. 2008.

[FaG10] G. Fairbanks and D. Garlan, Just enough software architecture: a risk-driven approach. 2010.

[Vog08] O. Vogel, I. Arnold, A. Chughtai, E. Ihler, T. Kehrer, U. Mehlig, and U. Zdun, Software-Architektur: Grundlagen-Konzepte-Praxis. 2008.

[Mic] Microsoft, "MVVM QuickStart." [Online]. Available: http://msdn.microsoft.com/en-us/library/gg430869(v=pandp.40).aspx. [Accessed: 16-Oct-2012].

[Kno12] "Knockout," 2012. [Online]. Available: http://knockoutjs.com/.

[Pop03] M. Poppendieck and T. Poppendieck, Lean Software Development: An Agile Toolkit. 2003.

[OPG] http://www.opengroup.org/