Whitepaper

Flexible Architekturen
(Stand: August 2009)

The Big Ball of Mud

„*The most frequently deployed software architecture: the Big Ball of Mud*", so haben Foote und Yoder das Ergebnis von 30 Jahren Software-Engineering zusammengefasst [1].

Größere Software-Systeme bestehen häufig aus verklumpten Strukturen. Strukturen, die bis zur Unkenntlichkeit erodiert sind, mit verworrener Kommunikation und Informationsverteilung über weit entfernte Elemente des Systems. Ein Klumpen voller Duplikationen und mit ungeregeltem Wachstum.
Wie kommt es dazu? Kein Softwareentwickler erstellt Code mit der Absicht, ein undurchdringliches Strukturdickicht zu erzeugen. Neben offensichtlichen Gründen wie Unerfahrenheit und Zeitdruck gibt es eine Reihe weiterer Gründe, von denen hier nur einige genannt seien:

it-agile GmbH
Paul-Stritter-Weg 5
22297 Hamburg
Fon +49 (0)40 88 173-300
Fax +49 (0)40 88 173-333

http://www.it-agile.de
Mail: info@it-agile.de

Handelsregister Hamburg
HRB 92261

it-agile

- Rotten Neighborhood Theory: Sind bereits kleine Verklumpungen im System, gesellen sich leicht weitere schlechte Strukturen dazu, ohne aufzufallen
- Prototypen, die nicht sofort weggeworfen werden, nachdem sie ihren Demonstrationszweck erfüllt haben, rutschen leicht mit ihrer Quick&Dirty-Implementierung in den Produktivcode.
- Peter Principle Programming: An das Berufskarrieren-Peter-Prinzip angelehnt gilt für Software: Die Komplexität einer Software steigt schnell bis zu genau einen Schritt hinter der Stufe, die einem Entwickler gerade noch verständlich ist [2].
- Conways Law: Eine Software ist immer so komplex wie die Strukturen der Organisation, für die sie entwickelt wird. [3]

Warum flexible Architekturen?

Warum ist ein derart erodiertes System so schlimm? Weil es die Wartung, Pflege und Weiterentwicklung eines Systems sehr teuer macht. Jede Änderung wird erschwert durch die Unverständlichkeit der Strukturen und die hohe, schwer durchschaubare Kopplung der Systemteile. Besonders für die inkrementelle Softwareentwicklung ist es Voraussetzung, dass Änderungen zu späterer Zeit günstig durchgeführt werden können, damit spätere Inkremente nicht deutlich teurer werden als die Inkremente der Anfangszeit.
Und wenn man den Begriff „inkrementell" etwas lockerer fasst, dann fallen sogar die klassisch per Wasserfall-Entwicklung erstellten Systeme darunter, da nahezu jedes System nach der Erstauslieferung korrigiert und erweitert wird, und damit also auch neue Inkremente erstellt werden.

Prinzipien einer flexiblen Architektur

Wie kann die beschriebene System-Erosion vermieden werden?
Für die Änderbarkeit von Software sind Verständlichkeit und Flexibilität die obersten Prinzipien. Die Kriterien eines flexiblen Systems sind unter anderem:

- Unabhängige Bestandteile (Modularisierung)
- Leicht kombinier-, erweiter- und änderbar
- Übersichtliche Grundstrukturen
- Wiederverwendbarkeit
- Austauschbarkeit

In den Anfangsjahren der Softwareentwicklung wurde nach dem Code&Fix-Paradigma entwickelt. Kleine Systeme erforderten keine umfangreichen Engineering-Praktiken. Je größer die Systeme wurden und je schlimmer der unverständliche Spaghetticode wurde, umso offensichtlicher wurde die Notwendigkeit von softwaretechnischen Strukturierungsleitbildern. Seit der Softwarekriese 1968 sind eine Vielzahl von Paradigmen und Mustern entstanden, die bei der Erstellung verständlicher und flexibler, erweiterbarer Software helfen, von denen einige hier erwähnt werden sollen:

- Komponentenbildung und Modularisierung: Software sollte in Komponenten mit klar getrennten inhaltlichen Aufgaben (separation of concerns), Schnittstellen und definierten Benutzungsbeziehungen strukturiert werden, um Verständlichkeit und Wartbarkeit zu gewährleisten.
- Lose Kopplung und hohe Kohäsion: Eine Komponente sollte so definiert sein, dass sie keine Interna anderer Komponenten kennt und nur locker an die öffentlichen Schnittstellen anderer Komponenten gebunden ist. [4]
  Eine Komponente sollte zusätzlich so gefasst sein, dass ihre Bestandteile alle einem klar umrissenen Zweck dienen und keine Sammlung verschiedener Aufgaben und Konzepte ist (hohe Kohäsion) [5].
- Von der prozeduralen Programmierung bis zur Objektorientierung wurden Mittel geschaffen, die für Entkopplung auf Sprachebene sorgen.

Entwicklung „solider" Software

Zusätzlich zu diesen Leitbildern wurde eine Reihe von Prinzipien entwickelt, die bei der Erstellung der Klassen und Module helfen. Robert Martin hat einen weit akzeptierten Satz an Prinzipien zusammengefasst, die richtig angewandt zu einer flexiblen Softwarearchitektur führen. Am wichtigsten sind nach Martin die SOLID-Prinzipien, die hier kurz vorgestellt werden. [6],[7]

- *Single-Responsibility-Principle:*
  Jede Klasse hat genau eine Aufgabe. Nur wenn sich an dieser klar umrissenen Aufgabe etwas ändert, dann muss auch die Klasse angepasst werden. Richtig umgesetzt führt das zu kleineren Klassen mit geringen Abhängigkeiten zu anderen Elementen.
- *Open-Closed-Principle:*
  Eine Entwurfseinheit sollte offen für Erweiterungen, aber geschlossen für Veränderungen an ihrer Grundstruktur sein. Damit ist gemeint, dass das Verhalten von Entwurfseinheiten geändert werden kann, ohne dass die Entwurfseinheit modifiziert werden muss. [8] Beispielsweise kann in einer abstrakten Oberklasse ein Algorithmus implementiert sein, der in Subklassen durch entsprechende Template-Methoden um spezifisches Verhalten erweitert wird. Ein weiteres Beispiel ist die Verwendung von Strategie-Objekten, mit denen das Verhalten anderer Objekte dynamisch modifiziert und erweitert werden kann.
  Man erhält dadurch einen flexiblen Entwurf. Dynamische Teile werden getrennt von der statischen Infrastruktur.
- *Liskov-Substition-Principle:*
  Ein Objekt einer Klasse muss überall einsetzbar sein, wo ein Objekt einer Oberklasse erwartet wird. Dass bedeutet, dass Objekte von Unterklassen sich grundsätzlich so verhalten müssen, wie in der Oberklasse definiert. Insbesondere dürfen keine Methoden wegdefiniert werden (in Java können Methoden beispielsweise „wegdefiniert" werden durch das Werfen von UnsupportedOperationExceptions), keine Vorbedingungen verschärft und keine Nachbedingungen aufgeweicht werden. Klienten des Grundtyps, also der Oberklasse, sollen kein Wissen darüber haben, welche Spezialisierungen der

Oberklasse existieren. Insbesondere sollen „instanceof"-Subtyp-Prüfungen vermieden werden, die die Klienten unflexibel machen und eng an die Vererbungshierarchie binden.

- *Interface Segregation Principle:*
  Schnittstellen sollen möglichst klein sein und die Operationen einer Schnittstelle sollen eine hohe Kohäsion aufweisen. Klassen mit vielen öffentlichen Methoden sind zu groß und sollten ihre Schnittstelle in verschiedene Interfaces auftrennen. Klienten sind dann nicht mehr gezwungen, die große, breite Schnittstelle zu verwenden, sondern können sich auf einen spezifischen zusammenhängenden Ausschnitt konzentrieren.
- *Dependency Inversion Principle:*
  Klassen sollen immer von Abstraktionen abhängen und nicht von konkreten Implementierungen. Jede Klasse, die Zugriff auf ein Objekt braucht, dass sie nicht selbst erzeugen kann oder will, lässt sich das Objekt von außen setzen. In der Klasse wird nur definiert, welche Objekte welcher Typen benötigt werden, die Erzeugung und Bereitstellung der benötigten Objekte wird von der Infrastruktur gewährleistet.
  In Abbildung 1 ist der grundsätzliche Perspektivenwechsel verdeutlicht. In der klassischen 3-Schichten-Architektur kennt die Geschäftslogikschicht die konkrete technische Datenbankschicht. In der rechts dargestellten Komponentenstruktur wird von der konkreten Persistierung abstrahiert, die Abhängigkeitsrichtung hat sich gedreht.

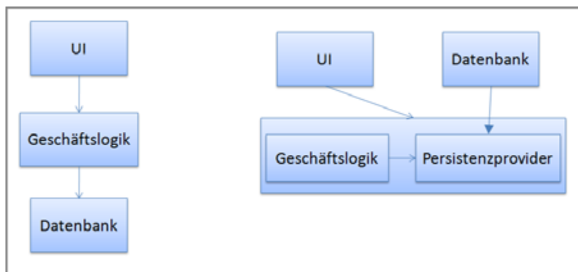

Abbildung 1: 3-Schichten-Architektur versus DIP

Werden die SOLID-Prinzipien beim Entwurf konsequent eingehalten, so zeichnet sich die entwickelte Architektur durch hohe Entkopplung und Flexibilität aus.

Einhaltung der SOLID-Prinzipien

Wenn also nun ein Satz an Prinzipien definiert ist, der hilft, den Big Ball of Mud zu verhindern, dann stellt sich die Frage, wie die Einhaltung der Prinzipien erreicht und überwacht werden kann.
Zunächst einmal müssen natürlich die Prinzipien und vor allem ihre praktische Anwendung den Entwicklern klar sein. Die Erfahrung zeigt, dass es nach der Einübung nicht reicht, nur an das gute Gewissen und die Entwicklerehre zu appellieren. In Zeiten von Termindruck sind Rückfälle in alte Muster vorpro-grammiert. Es entstehen kleinere Verklumpungen, und daraus resultieren dann schnell größere faule Strukturen (Rotten-Neighbourhood-Theory, s.o.).
Zum einen gibt es die Möglichkeit der manuellen Kontrolle:

- durch Code Reviews: regelmäßige Code-Begutachtungen in der Gruppe helfen, bereits kleinere strukturelle Unschönheiten früh aufzudecken.
- über Pair Programming: Das Programmieren in Paaren verhindert häufig schon die Entstehung kleinerer Verklumpungen, da permanent ein Review-Augenpaar bei der Entwicklung anwesend ist.

Es gibt aber auch die Möglichkeit, die Flexibilität der Architektur permanent automatisiert kontrollieren zu lassen. Als Beispiele seien hier Checkstyle, ein kostenloses Plugin für Eclipse [9], und XDepend [10], ein kommerzielles Architekturanalystool genannt.
Diese Tools machen manuelle Architekturreviews nicht überflüssig, helfen aber bei der Einhaltung einiger Prinzipien:

- Einhaltung des Single-Responsibility-Principles:
  Checkstyle misst während der Entwicklung permanent den Wert der „Fan-Out-Complexity“, das ein Maß für die Kopplung einer Klasse an andere Klassen ist. Hat eine Klasse zu viele Beziehungen zu anderen Klassen, dann ist das ein Indikator dafür, dass die Klasse zu viele Zuständigkeiten hat und damit das SRP verletzt.

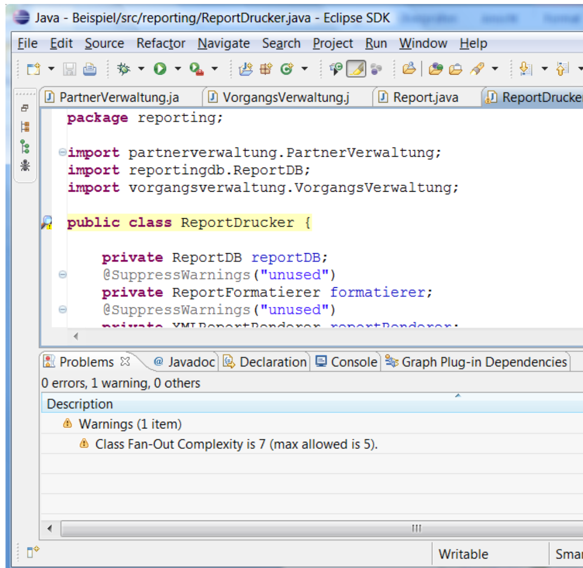

Abbildung 2: Fan-Out-Complexity-Überschreitung

- Einhaltung des Interface-Segregation-Principles:
  Das XDepend-Architekturtool überprüft und visualisiert, ob es Interfaces gibt, die eine definierte Obergrenze von Operationen übersteigt. Wird der Wert überschritten, so ist das ein Indikator dafür, dass die Schnittstelle zu breit definiert ist und besser in sinnvolle, kohärente Bestandteile unterteilt werden sollte.

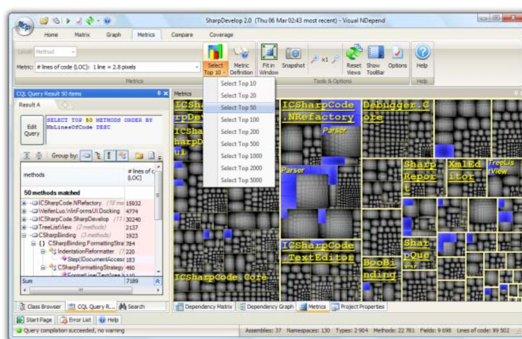

Abbildung 3: XDepend mit markierten Klassen

- Einhaltung des Dependency-Inversion-Principles:
  Checkstyle ermittelt bei jedem Speichern den Wert der Abstract-Data-Coupling-Metrik. Wird eine Obergrenze überschritten, wird eine Warnung oder ein Fehler erzeugt. Vereinfacht gesagt misst die ADC-Metrik die Anzahl verschiedener Konstruktor-Aufrufe einer Klasse. Gibt es zu viele dieser „new"-Aufrufe, dann ist das ein Indikator für eine zu enge Kopplung.

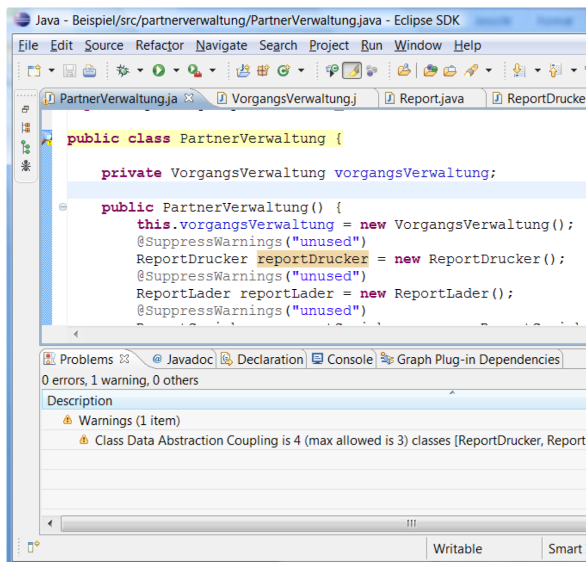

Abbildung 4: CDA als Indikator für DIP-Verletzung

Für immerhin drei der fünf SOLID-Prinzipien gibt es automatisierbare Überprüfungsunterstützungen. Und für Checkstyle gibt es sogar die Möglichkeit, die Überprüfung der Metriken in den Pre-Checkin-Prozess von Subversion einzubinden [11]. Damit lässt sich dann sogar komplett verhindern, dass Code im Repository landet, der gegen die Metriken und damit die Prinzipien verstößt. Und damit werden dann schon Little Balls of Mud, die Keimzellen eines Big Balls of Mud, im Ansatz verhindert. Die Voraussetzungen für eine langlebige, flexible, wartbare Architektur sind damit geschaffen.

Zusammenfassung

Eine verständliche und flexible Architektur ist die Voraussetzung für die Wart- und Änderbarkeit eines Softwaresystems. Neben den Grundmechanismen von Modularisierung und Komponentenbildung gibt es eine Reihe von Prinzipien, deren Einhaltung zu entkoppelten Strukturen führt. Die SOLID-Prinzipien bilden eine gute Basis, an der man sich bei der Entwicklung orientieren kann. Die Einhaltung der SOLID-Prinzipien lässt sich zum Teil automatisiert überwachen. Die dazu benötigten Tools können gut in den Entwicklungsprozess eingebunden werden und verhindern schon im Ansatz die Entstehung verklumpter Strukturen.

Referenzen

[1] Brian Foote and Joseph Yoder, “Big Ball of Mud”, Fourth Conference on Patterns Languages of Programs, Monticello, Illinois, September 1997

[2] http://c2.com/cgi/wiki?PeterPrincipleProgramming

[3] Fred Brooks, "The Mythical Man-Month"

[4] http://en.wikipedia.org/wiki/Coupling_(computer_science)

[5] http://en.wikipedia.org/wiki/Cohesion_(computer_science)

[6] Robert C. Martin, “Agile Software Development”, Prentice Hall

[7] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[8] Bertrand Meyer, “Object-Oriented Software Construction”, Prentice Hall

[9] http://eclipse-cs.sourceforge.net/

[10] http://www.xdepend.com/, noch im RC-Status

[11] http://www.javaworld.com/javaworld/jw-11-2008/jw-11-checkstyle2.html?page=3

Unser Angebot zu flexiblen Architekturen

Wir bieten Schulungen und Beratung zu verschiedenen Architektur-Themen, auch inhouse. Besuchen Sie unsere Webseite http://www.it-agile.de oder treten Sie mit uns in Kontakt: info@it-agile.de