Modularisierung in Legacy-Projekten - API Determination

Matthias Gutheil
matthias.gutheil@itemis.de
itemis AG

Zusammenfassung

In vielen Legacy-Projekten, die über mehrere Jahre oder gar Jahrzehnte entwickelt worden sind, wurde über die Projektdauer zu wenig Wert auf die Architektur gelegt. Wir diskutieren, wie es in Software-Projekten zu einer Architektur-Erosion kommen konnte und wie man diese hätte verhindern können.

Besonderen Augenmerk legen wir auf das Thema Modularisierung, und hier besonders auf das nachträgliche Herausarbeiten einer Schnittstelle (API-Determination).

1 Einleitung

In vielen langlebigen Software-Projekten, in denen zu wenig Aufmerksamkeit auf die Architektur gelegt wurde, kommt es zu einem sogenannten Big Ball of Mud (BBoM) [2]. Dies bezeichnet die Tatsache, dass sehr wenig Struktur in der Software zu erkennen ist. In den seltensten Fällen ist solch ein System gut getestet, was Änderungen und Weiterentwicklung erschwert.

Wie Food und Yoder treffend sagen: "If you think good architecture is expensive, try bad architecture." Wenn man frühzeitig in Software-Projekten auf die Architektur achtet und wenig technische Schulden eingeht, lohnt sich dies schon mittelfristig. Dies trifft besonders auf langdauernde Projekte zu, bei denen in mehreren Teams an verschiedenen Teilen des Systems gearbeitet wird. Nur in Ausnahmefällen, z.B. wenn Time to Market sehr hohe Priorität hat, kann man technische Schulden bewusst anhäufen. Es sollte in diesem Fall aber kurz- bis mittelfristig entweder neu entwickelt werden oder die technischen Schulden beseitigt werden. Im Folgenden reden wir von Architektur-Erosion statt Big Ball of Mud. Dies beschreibt mehr den schleichenden Prozess bei dem mehr und mehr technische Schulden aufgebaut werden.

2 Was sind die Gründe von Architektur-Erosion?

Warum kommt es nun in vielen langlaufenden Projekten zu einer Architektur-Erosion? Der allerwichtigste Punkt ist unserer Erfahrung nach, dass bei vielen Entwicklern oder ganzen Teams das Bewusstsein für Architektur nicht besteht. Besteht dieses Bewusstsein ist die Voraussetzung geschaffen, dass mehr und mehr auf die Architektur geachtet wird. Ist beispielsweise die hexagonale Architektur bekannt, so werden die Entwickler fachlichen Code nicht mit technischem Code vermischen. Im Folgenden betrachten wir einige Punkte, die die Architektur-Erosion ermöglichen.

2.1 Die Module besitzen keine sauberen Schnittstellen

Dies führt dazu, dass auf verschiedene Art und Weise auf bzw. in die Module zugegriffen wird. Programmcode der von außen sichtbar ist, wird auch genutzt. Dies konnten wir in vielen langlaufenden Projekten beobachten. So werden Utility-Klassen wie z.B. Formatter, Konvertierungen, Validierungen oder Stringfunktionen von Modulen genutzt, die die aufrufenden Module rein fachlich nicht benötigen.

Dadurch, dass die Module keine sauberen Schnittstellen besitzen, existiert auch keine API / Interfaces gegen die entwickelt wird. Dies führt zu hoher Kopplung, was die konkrete Abhängigkeit der Module erhöht. Dadurch wird das Testen erschwert, da nur gegen die echten Module getestet werden kann. Würden Interfaces genutzt, so könnte man einfacher mit Mock-Objekten testen.

Der saubere Modulschnitt ist wohl das wichtigste Thema bei der Architekturarbeit. Dies erkennt man auch an Themen wie Domain Driven Design oder der Microservice-Architektur, welche beide einen sauberen Modulschnitt begünstigen.

2.2 Vermischung von fachlichem und technischem Code

In vielen Projekten mussten wir feststellen, dass fachlicher Code nicht frei von technischem Code ist. Beispiele sind das Nutzen von Spring oder Hibernate in den Fachklassen. Dies erschwert es, die Fachklassen in einem anderen Kontext zu nutzen, seien es nur Unit-Tests die ohne Spring oder Hibernate auskommen könnten.

Technischer Code in Fachklassen erschwert auch einen Technologiewechsel, so wird es z.B. bei einer selbst geschriebenen Datenhaltung schwer das Datenhaltungsframework zu wechseln.

2.3 Definieren und Validieren der Software-Architektur

Oft wird in Projekten zu Beginn des Projektes die Architektur definiert. Immerhin existiert in solchen Projekten eine Architekturvorgabe. Nur leider wird in fast

allen Projekten später nicht mehr überprüft ob die Architekturvorgabe eingehalten wurde. So kann es dann passieren, dass Fachklassen eine Abhängigkeit zur UI haben oder Module voneinander abhängen, die eigentlich keine Abhängigkeit haben sollten.

Weiterhin erschweren Praktiken wie das Verwenden von Reflection das automatische Erkennen von Architekturverletzungen.

3 API-Determination

Wie geht man nun vor, wenn man ein Modul vorfindet, welches über die Jahre zu groß geworden ist und man einen funktionalen Bestandteil herauslösen möchte? In vielen Projekten ist solch ein größeres Refactoring nur in bestimmen Zeiträumen möglich. Hinzu kommt noch, dass eine Änderung fast alle Kollegen betrifft und man die Änderung nur als Ganzes durchführen kann. Aus diesem Grunde muss die Änderung lokal vorbeireitet werden.

Die Herausforderung besteht nun darin, die API so minimal zu gestalten wie möglich. Die Ist-API ist oft viel zu groß und besteht aus allen öffentlich zugänglich und auch verwendeten Klassen und Methoden. In der Praxis haben sich folgende Schritte bewährt.

3.1 API verkleinern

Zuerst sollte die Ist-API verkleinert werden um eine geringere Menge an Klassen und Methoden zu erhalten, auf die von außen zugegriffen wird. Hier helfen Werkzeuge wie Structure 101 [3] oder das jdt-codemining [4] Eclipse Plug-In.

1. Klassen und Methoden löschen, die nicht mehr benutzt werden.
2. Sichtbarkeiten von Klassen und Methoden minimieren.
3. Zugriffe auf Implementierungsklassen in API Zugriffe umwandeln.
4. Klassen verschieben, die nicht in das Paket gehören.

3.2 Fachliche Interfaces identifizieren

Der nächste Schritt besteht darin, fachliche Interfaces zu identifizieren und die Implementierung von der API zu trennen. Dies ermöglichst es später reine API-Module zu verwenden oder die Zugriffe auf die API-Pakete zu verifizieren.

1. Unterschiedliche Funktionalitäten des Moduls/-Paketes erkennen.
2. API und Implementierungen in eigene Pakete verschieben.

3.3 API absichern

Nun haben wir Schnittstellen identifiziert und die API von der Implementierung getrennt. Momentan müssen noch einige Implementierungsklassen public sein, sonst könnten sie nicht erzeugt werden. Nun muss verhindert werden, dass neue Zugriffe auf die Implementierungsklassen entstehen. Hier verwenden wir ArchUnit [1], ein sehr mächtiges Werkzeug. Damit ist es z.B. möglich, Zugriffe auf Pakete nur von bestimmen Paketen zu erlauben.

Die Architekturregeln liegen dann als Unit-Tests vor und können lokal als auch im Build ausgeführt werden.

4 Verhindern von Architektur-Erosion

Wie verhindert man nun eine Architektur-Erosion? Wir erachten die folgende Punkte als die Wichtigsten:

- Architektur diskutieren, definieren, kommunizieren und validieren.
- Jedes Modul wird nur über seine API angesprochen. Hier kann Dependency Injection (z.B. Spring) verwendet werden.
- Pair Programming und Reviews dienen zum Know-How Austausch auch in Richtung Architektur.
- Sinnvolle Metriken sollten erhoben werden um z.B. bei einer bestimmten Modul/Paketgröße Alarm zu schlagen.

Die obigen Punkte dienen sowohl dazu das Know-How der Mitarbeiter bzgl. Architektur zu verbessern, als auch technische Möglichkeiten zu verwenden um die gewünschte Architektur einzuhalten.

5 Fazit

Wir sind uns noch unsicher, warum in vielen Projekten nicht ausreichend auf die Architektur geachtet wurde. Lag es an dem mangelnden Wissen der Projektmitarbeiter oder an den fehlenden Werkzeugen zum Überprüfen auf Architekturverletzungen? Oder wurde zu sehr auf das Implementieren weiterer Features wert gelegt und erst dann auf die Architektur geachtet, als dies unabdingbar war. So können nicht funktionale Anforderungen wie z.B. neue UI-Technologien, Umstieg von Fat-Client ins Web, Performanceprobleme oder gar eine sich stark verlangsamende Entwicklung Grund sein um die technischen Schulden zu beseitigen.

Literatur

[1] *ArchUnit Webseite*. `https://www.archunit.org/`. Besucht: 2019-05-12.

[2] *Big Ball of Mud*. `http://www.laputan.org/mud/mud.html#BigBallOfMud`. Besucht: 2019-05-12.

[3] *Structure 101 Webseite*. `https://structure101.com/`. Besucht: 2019-05-12.

[4] *jdt-codemining Webseite*. `https://github.com/angelozerr/jdt-codemining`. Besucht: 2019-05-12.