Automatisierte Architekturanalyse

Marc Breit
Department Informatik
Hochschule für Angewandte Wissenschaften
Hamburg, Germany
marc.breit@haw-hamburg.de

***Abstract*—Obwohl das Aufdecken von Code Smells automatisiert bereits gut funktioniert, ist das automatisierte Aufdecken von architekturelle Mängel noch Gegenstand aktueller Forschung. Hier sollen einige der aktuellen Forschungsstände dargelegt werden.**

Although the detection of Code Smells already works well in an automated way, the automated detection of archtiectural flaws is still subject of current research. Some of the current state of research will be presented here.

***Index Terms*—Arcitectural Flaws, Architectural Smells, Automatisierte Architekturanalyse**

I. Einleitung

Softwarelösungen sind in der Regel schon bei ihrer Erstellung komplexe Gebilde. Über ihre Lebensdauer steigt diese Komplexität mit jedem weiteren Feature und jedem weiteren Bugfix, meist unbeachtet durch die Entwickler, weiter an. Das Verletzten von Codeconventionen und Architekturregeln kann z.B. aus Zeitgründen oder aber auch aus Unwissenheit über die jeweiligen Regeln erfolgen. Am Ende steht häufig ein unwartbares System. Der etablierte Begriff für diese Art von technischen Fehlentscheidungen ist *Technische Schulden* [6]. Im Detail kann hier noch weiter unterteilt werden. Code-Smells, also technische Schulden, die sich mit Implementationsdetails beschäftigen, werden hier nicht weiter betrachtet. Architekturelle Mängeln bzw. Struktur-Smells [8, S.14] steht hingegen im Vordergrund.

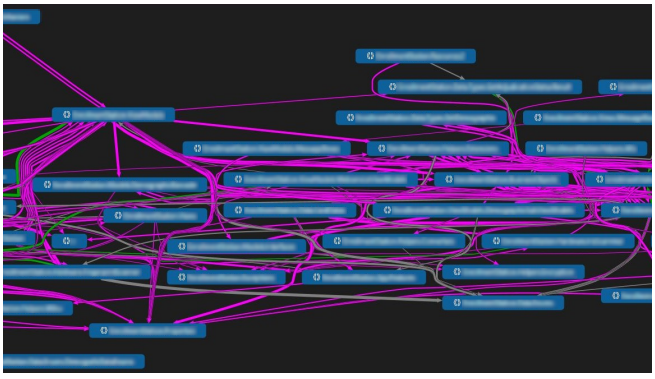

Fig. 1. Beispiel für hohe Kopplung in einer C# Applikation

Figure 1 zeigt eine über mehrere Jahre und unter häufigem Entwicklerwechsel gewachsene C# Applikation, die in dieser Zeit nie einer Refaktorisierung unterzogen wurde. Ursprüngliche Architekturentscheidungen wurden mit jedem weiteren Release aufgeweicht und neue, teils widersprüchliche, hinzugefügt.
Warum das Fehlen von Refaktorisierungsphasen zu einem solchen Zustand führen kann ist in Figure 2 zu erkennen. [8, Vgl. S.5f] Der ideale Bereich befindet sich für ein Softwareprojekt dort, wo die technische Schuld gering ist. Fehlerbehebung und das Erweitern des Softwareproduktes lassen sich sowohl aus zeitlich als auch aus finanzieller Sicht gut planen. Allerdings steigt, je nach Komplexität der Änderungen, bei jeder Bearbeitung des Quellcodes die technische Schuld an. Kümmert man sich nicht kontinuierlich um diese Mängel werden die Fortschritte, die pro Release umgesetzt werden können, kleiner und der Aufwand für die Wartung des Systems steigt. Ab einem gewissen Punkt mach es dann keinen Sinn mehr die Software zu refaktorisieren und es kommt aus ökonomischen Gesichtspunkten nur noch eine Neu-Implementierung in Frage.

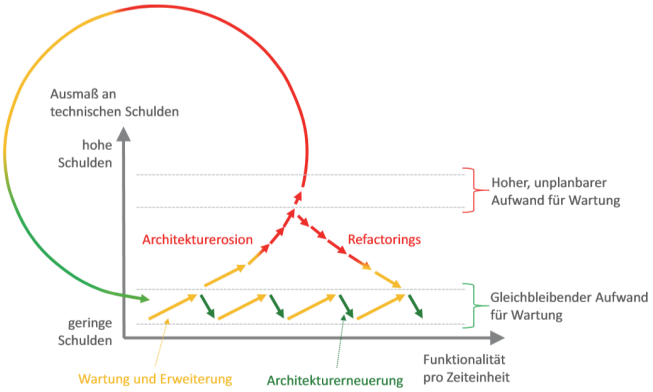

Fig. 2. Architekturerosion [8, S.5]

II. Architekturanlayse

Generell kann man Softwarearchitektur durch verschiedene Ansätze bewerten [9]. Während sich quantitative Verfahren vor allem auf die Bewertung von Implementierungsdetails durch den Einsatz diverser Metriken beziehen, beziehen sich qualitative Ansätze auf die Bewertung von Qualitätsanforderungen. Ein Ansatz zur qualitativen Archtekturanalyse ist die *Architecture Tradeoff Analysis Method* (ATAM). Hierbei wird basierend auf Geschäftzielen Qualitätsmerkmale definiert, die

die Architektur für bestimmte Szenarien erfüllen muss. Qantitative Ansätze zur Bewertung beinhalten meist Metriken, die z.B. die Kopplung der einzelnen Software Einheiten messen oder aber auch die zyklomatische Komplexität. [9, Vgl. S.305ff]

A. Toolgestützte manuelle Analyse

Um eine umfangreiche Analyse eines Softwaresystems durchzuführen untersucht Lilienthal [8] die Soll- und die Ist- Architektur. Die Soll-Architektur stellt dabei die geplante Architektur da und die Ist-Architektur die wirklich implementierte. Figure 3 zeigt das weitere Vorgehen. Nachdem toolunterstützt der Sourcecode in eine Struktur gebracht wurde, die ein weiteres Untersuchen möglich macht, wird darauf ein Mapping der Soll-Architektur durchgeführt. Durch den Vergleich der beiden Artefakte können nun Unterschiede und mögliche architekturelle Mängel in der Codebasis erkannt werden [8, S. 21ff].

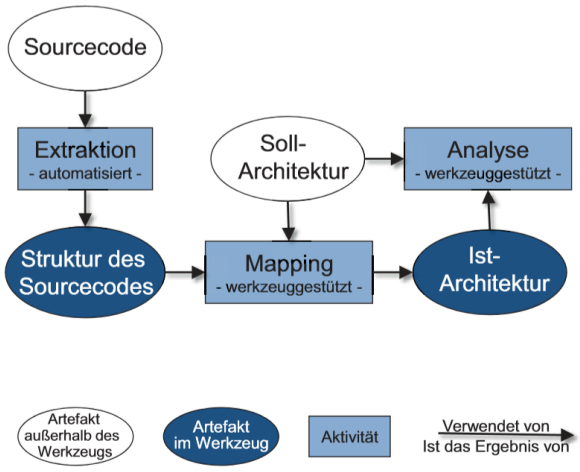

Fig. 3. Beispiel für das Vorgehen bei einer manuellen Architekturanalyse [8, S.23]

Figure 4 zeigt ein Beispiel mit dem kommerziellen Tool *Sotograph* [1]. Der Quellcode wird in einer Baumstruktur dargestellt und mit den Informationen der Soll-Architektur ergänzt. So kann man direkt die einzelnen Schichten des Systems, absteigend ergänzt mit Abhängigkeitsinformationen, untersuchen. Mit diesem Tool lassen sich Verletzungen des Schichtenmodells sehr gut erkennen. Verbindungen von oben nach unten sind erlaubt und grün dargestellt, von unten nach oben (Schichtenübergreifend) verboten und rot dargestellt.

B. Automatische Analyse

Quantitative Ansätze zur Bewertung beinhalten Metriken, die z.B. die Kopplung der einzelnen Software Einheiten messen oder aber auch die zyklomatische Komplexität. [9, Vgl. S.305ff]. Statische Code Analyse und diverse Metriken werden bereits seit vielen Jahren in Entwicklungsumgebungen und Buildservern eingesetzt um Code Smells zu erkennen. In Abschnitt 3 werden nun einige Ansätze betrachtet, die

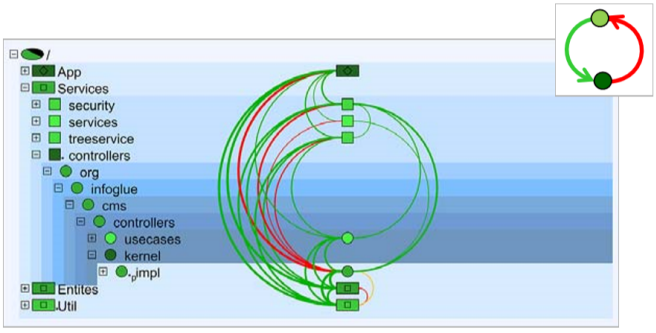

Fig. 4. Sotograph [8, S.43]

versuchen aus diesen Metriken architekturelle Mängel zu identifizieren.

III. Aktuelle Forschung

A. Software Architecture Health Monitor und Design Rules Spaces

1) Design Rule Spaces: Basierend auf Baldwin und Clark Design Rule Theory [3] wurde von Cai et al. das Design Rule Spaces Modell entwickelt [5] . Ein Design Rule Space ist dabei z.B. durch einen fachlichen Zusammenhang oder ein verwendetes Entwurfsmuster definiert. Sie merken an, das bisherige Forschungsansätze Software Einheiten immer isoliert betrachten und dadurch viele verschiedene Sichten entstehen, die alle für sich Sinn machen. Ihr Ansatz geht allerdings darüber hinaus, in dem sie zulassen, dass eine Software Einheit in mehreren Clustern aktiv sein kann. Um im Weiteren eine Relation zwischen mangelhafter Architektur und Fehleranfälligkeit herzustellen, betrachten sie auch die Änderungshistorie. Diese kann bei der Vorhersage von zukünftigen Fehlerquellen von Nutzen sein. Aus dieser Erkenntnis leiten sie ab, dass Fehler häufig nicht vollständig entfernt werden. Um strukturelle Zusammenhänge zwischen den einzelnen Quellcodedateien analysieren zu können tragen sie die Abhängigkeiten in eine Matrix ein um im weiteren daraus wieder Cluster, die DR Spaces, zu bilden.

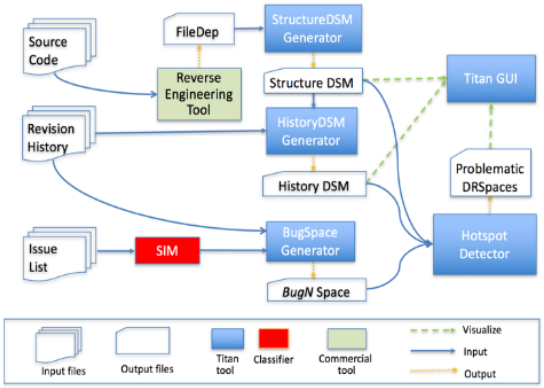

Fig. 5. Titan Tool Chain [4]

2) Software Architecture Health Monitor: Figure 5 zeigt das entwickelte Tool Set ”Titan Tool Chain” [4]. Als Eingangsquellen dienen neben dem Quellcode und der Revisionshistorie auch die Informationen aus dem Ticketsystem. Um die strukturellen Abhängigkeiten der einzelnen Quellcodedateien aufzulisten wird zunächst eine Design Structure Matrix (DSM) erstellt. Auch von der Revisionshistorie wird eine DSM erstellt, allerdings ist hier nicht der strukturelle Aufbau, sondern ob Dateien zusammen eingecheckt wurden relevant. Um DRSpaces abzuleiten werden nun die beiden Matrizen mit Hilfe des Design Rule Algorithm (DRH) gruppiert. Um mit diesem Werkzeug die ”Gesundheit” der Softwarelösung zu bewerten, werden neben den Architectural Roots, die sich aus der Analyse der DRSpaces ergeben, weitere Ansätze verfolgt. Architekturelle Mängel wie häufige Schnittstellenänderungen, Zyklen zwischen Modulen und Paketen, falsche Ableitung oder Modularitätsverletzungen werden ebenso beobachtet, wie das Decoupling Level. Am Ende soll dieser Ansatz als Entscheidungsgrundlage für eventuelle Refactoring Maßnahmen dienen.

B. Automatic Detection Of Instability Architectural Smells

Fontana et al. [7] haben ebenfalls ein Werkzeug entwickelt, um architekturelle Mängel in Java Projekten zu identifizieren. Sie konzentrieren sich dabei auf 3 Kernprobleme:

- Existieren Abhängigkeiten zu instabilen Komponenten
- Existieren Hub-artige Abhängigkeiten bzw. zentrale Knotenpunkte, die viele Abhängigkeiten zu anderen Komponenten haben
- Existieren zyklische Abhängigkeiten zwischen Komponenten

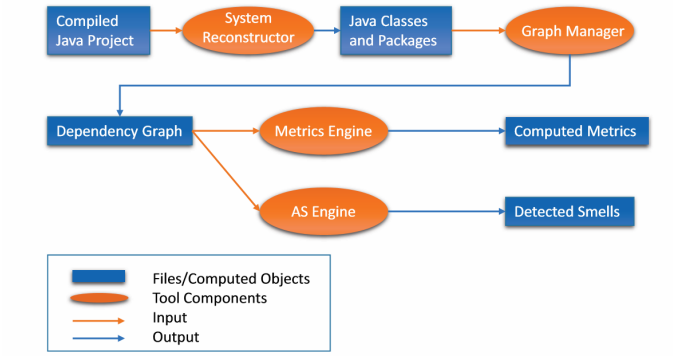

Fig. 6. Arcan Tool [7]

In Figure 6 ist der Workflow ihres Tools zu sehen [7]. Der vorgelagerte Schritt ist das dekompilieren des Java Projekts. Im Weiteren werden aus der gewonnen Klassenstruktur die Abhängigkeiten der einzelnen Klassen/Dateien herausgefiltert. Interessant hierbei ist, dass sie zur Repräsentation der Abhängigkeiten (im Gegensatz zu Cai et al.[5]) einen Abhängigkeitsgraphen gewählt haben. Das gibt ihnen die Möglichkeit zusätzliche Informationen zur Art der strukturellen Abhängigkeit zu speichern.

QUARTZ UNSTABLE DEPENDENCY RESULTS

Package	Arcan	inFusion	Bad dep. %	Filtered
org.quartz.core	yes	no	<30%	no
org.quartz.utils.counter.sampled	yes	no	35%	yes
org.quartz.ee.jta	yes	no	32%	yes
org.quartz.impl	yes	no	<30%	no
org.quartz.utils	yes	yes	100%	yes
org.quartz.impl.jdbcjobstore.oracle	yes	no	<30%	no
org.quartz.utils.counter	yes	no	83%	yes
org.quartz	yes	no	60%	yes

HUB-LIKE RESULTS

Derby	Jedit	Junit	Maven	Quartz	Spring	Struts
1	7	1	3	0	2	1

Fig. 7. Ergebnisse des Arcan Tools [7]

Figure 8 zeigt den schematischen Aufbau ihres Abhängikeitsgraphen. Für die Untersuchung der ”Instabilen Abhängigkeiten” werden für alle als Paket deklarierten Knoten im Graphen ”Instabilitäts-Metriken” berechnet. Je mehr Abhängigkeiten zu instabilen Komponenten bestehen, desto stärker ist der architekturelle Mangel. Die Ergebnisse, die das Arcan Tool liefert, wurden von ihnen u.a. mit dem Tool inFusion verglichen. Figure 7 (links) zeigt ihre Ergebnisse für Apache Quartz. Da sich die Werte stark unterschieden, haben sie einen Filter nachgelagert und verschiedene Schwellwerte angewendet um ihre Ergebnisse zu verfeinern.

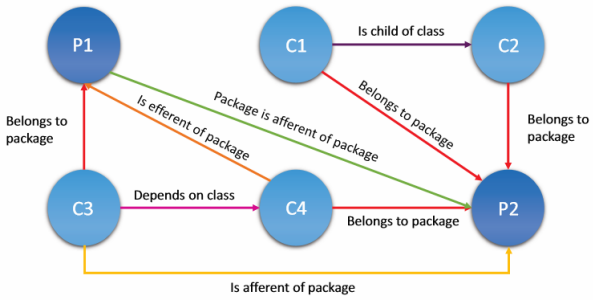

Fig. 8. Dependency Graph [7]

Um große fehleranfällige Knotenpunkte zu finden, werden im Abhängigkeitsgraph alle Klassen sowie ihre ein- und ausgehenden Verbindungen betrachtet [7]. Liegt bei einer Klasse der Wert dieser Abhängigkeiten über dem, der über alle Klassen im Projekt gebildet wurde, wird im Nachgang untersucht, ob das Verhältnis von eingehenden und ausgehenden Abhängigkeiten nicht den Schwellwert übersteigt (25% der Gesamtzahl aller Abhängigkeiten der Klasse). Sie merken an, dass sich diese Ergebnisse mit anderen Tools nicht vergleichen lassen, da andere Tools diese Metrik nicht verwenden. Sie verweisen in ihrem Paper selbst darauf, dass man hier nicht unbedingt von einem architekturellen Mangel ausgehen muss. Stattdessen könnte es sich um eine bewusste architekturelle Entscheidung gehandelt haben um den Kontrollfluss besser kontrollieren zu können. Figure 7 (rechts) zeigt die Ergebnisse des Arcan Tools für 7 verschiedene Software Projekte. Um sowohl auf Paket-Ebene als auch auf Klassen-Ebene Zyklen zu erkennen verwenden Fontana et al. Depth First Search. Der Vergleich ihres Tools mit Hotspot Detector zeigt, dass Resultate sehr davon abhängen wie der Schwellwert der Zykluserkennung gewählt wird. Fontana et

al. planen weiterführend, ähnlich wie in der Titan Tool Chain, die Revisionsinformationen mit einzubeziehen, um auch hier architekturelle Mängel aufdecken zu können.

IV. Fazit und Ausblick

Wie bereits erwähnt, ist die automatisierte Erkennung von Architekturmängeln noch Gegenstand aktueller Forschung. Dabei kommt es vor allem auf die verwendeten Metriken an. Im Bezug auf verteilte Softwaresysteme, wie z.B. Microservice-Architekturen, wäre es interessant zu klären, ob sich die erwähnten Ansätze auch hier anwenden lassen können. Ein weiteres Feld wären formale Verifikationsmethoden. Altoyan und Perry [2] schreiben in ihrem Paper z.B., dass manuellen Methoden nicht mehr ausreichen, sobald Softwaresysteme komplexer werden. Eine präzise Methode wäre aus ihrer Sicht wünschenswert, um Architekturen bewerten und verifizieren zu können.

References

[1] URL: https://www.hello2morrow.com/products/sotograph (visited on 02/28/2019).

[2] Najd Altoyan and Dewayne E. Perry. "Towards a Well-formed Software Architecture Analysis". In: *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*. ECSA '17. Canterbury, United Kingdom: ACM, 2017, pp. 173–179. ISBN: 978-1-4503-5217-8. DOI: 10.1145/3129790.3129813. URL: http://doi.acm.org/10.1145/3129790.3129813.

[3] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity Volume 1*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0262024667.

[4] Yuanfang Cai and Rick Kazman. "Software Architecture Health Monitor". In: *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*. BRIDGE '16. Austin, Texas: ACM, 2016, pp. 18–21. ISBN: 978-1-4503-4153-0. DOI: 10.1145/2896935.2896940. URL: http://doi.acm.org/10.1145/2896935.2896940.

[5] Y. Cai et al. "Design Rule Spaces: A New Model for Representing and Analyzing Software Architecture". In: *IEEE Transactions on Software Engineering* (2018), pp. 1–1. ISSN: 0098-5589. DOI: 10.1109/TSE.2018.2797899.

[6] Ward Cunningham. "The WyCash Portfolio Management System". In: *SIGPLAN OOPS Mess.* 4.2 (Dec. 1992), pp. 29–30. ISSN: 1055-6400. DOI: 10.1145/157710.157715. URL: http://doi.acm.org/10.1145/157710.157715.

[7] F. A. Fontana et al. "Automatic Detection of Instability Architectural Smells". In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Oct. 2016, pp. 433–437. DOI: 10.1109/ICSME.2016.33.

[8] C. Lilienthal. *Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen*. 2., überarbeitete und erweiterte Auflage. Dpunkt Verlag (Heidelberg). ISBN: 978-3-86490-494-3.

[9] Gernot Starke. *Effektive Softwarearchitekturen: Ein praktischer Leitfaden*. 8., überarbeitete Auflage. Carl Hanser Verlag (München). ISBN: 978-3-86490-494-3.