Dr. Carola Lilienthal
(cl@wps.de)
ist Geschäftsführerin der WPS - Workplace Solutions GmbH. Sie analysiert seit 2003 regelmäßig die Zukunftsfähigkeit von Softwarearchitekturen und spricht auf Konferenzen über dieses Thema. 2015 hat sie ihre Erfahrungen aus über zweihundert Analysen in dem Buch „Langlebige Softwarearchitekturen – Technische Schulden analysieren, begrenzen und abbauen" zusammengefasst.

Langlebige und erweiterbare Architekturen ohne Schulden

Fast jedes Softwaresystem wird mit guten Vorsätzen aber unter schwierigen Bedingungen entwickelt. Die Entwickler haben mit Deadlines zu kämpfen, die sie zwingen, Hacks zu programmieren; es gibt unterschiedliche Qualifikationen im Entwicklungsteam, die zu Code-Anteilen mit unterschiedlichen Qualitäten führen; und zusätzlich muss das Team sich auch noch um alten Code kümmern, der unordentlich und zu einem großen Knäuel verwoben ist. Die Wartung wandelt sich mit der Zeit immer mehr von strukturierter hin zu defensiver Programmierung. Die Entwickler fangen an, Code zu schreiben, von dem sie wissen, dass er aus Architektursicht schlecht ist. Der Code wird immer komplexer und das Team häuft technische Schulden an. Die Kosten für die Wartung steigen und Erweiterungen führen zunehmend zu mehr Seiteneffekten. Das Ergebnis ist eine schlechte Architektur, die die Entwicklungskosten in die Höhe treibt. Um dieser Abwärtsspirale auf Dauer entgegenwirken zu können, brauchen wir eine qualitativ hochwertige und flexible Architektur mit möglichst wenig technischen Schulden. Sind die technischen Schulden gering, dann finden sich die Wartungsentwickler gut im System zurecht. Sie können schnell und einfach Bugs fixen und haben keine Probleme, kostengünstig Erweiterungen zu machen. Wie kommen wir in dieses gelobte Land der Architekturen mit reduzierten technischen Schulden?

Technische Schulden

Der Begriff „Technische Schulden" wurde 1992 von Ward Cunningham geprägt [Cun92]. Technische Schulden entstehen, wenn bewusst oder unbewusst falsche oder suboptimale technische Entscheidungen getroffen werden. Diese falschen oder suboptimalen Entscheidungen führen zu einem späteren Zeitpunkt zu Mehraufwand, der Wartung und Erweiterung teurer macht. Zu dem Zeitpunkt der falschen oder suboptimalen Entscheidung hat man also technische Schulden aufgenommen, die man mit ihren Zinsen irgendwann abbezahlen muss, wenn man nicht überschuldet enden will.

In der Literatur werden verschiedene Arten und Varianten von technischen Schulden aufgeführt. In diesem Artikel stehen die technischen Schulden im Fokus, die Softwareentwickler bei ihrer Arbeit ausbremsen:

- **Implementationsschulden:** Im Quellcode (engl. Source Code) finden sich sogenannte Code-Smells, wie lange Methoden, leere Catch-Blöcke usw. Implementationsschulden sind heute weitgehend automatisiert mit einer Vielzahl von Tools im Quellcode zu finden. Jedes Entwicklungsteam sollte diese Schulden in seiner täglichen Arbeit sukzessive beheben, ohne dass extra Budget dafür erforderlich ist.
- **Design- und Architekturschulden:** Das Design der Klassen, Pakete, Subsysteme, Schichten und Module und die Abhängigkeiten zwischen ihnen sind uneinheitlich, komplex und passen nicht mit der geplanten Architektur zusammen. Diese Schulden sind durch einfaches Zählen und Messen nicht zu ermitteln und bedürfen einer umfassenden Analyse, die im Verlauf dieses Artikels vorgestellt wird.

Andere Problemfelder, die man auch als Schulden von Softwareprojekten betrachten kann, wie fehlende Dokumentation, geringe Testabdeckung, schlechte Usability oder ungenügende Hardware, bleiben hier außen vor.

Entstehen von technischen Schulden

Wurde zu Beginn eines Softwareentwicklungsprojekts eine qualitativ hochwertige Architektur entworfen, dann kann man davon ausgehen, dass das Softwaresystem sich am Anfang gut warten lässt. In diesem Anfangsstadium befindet sich das Softwaresystem in dem Korridor geringer technischer Schulden mit gleichbleibendem Aufwand für die Wartung (siehe Abbildung 1).

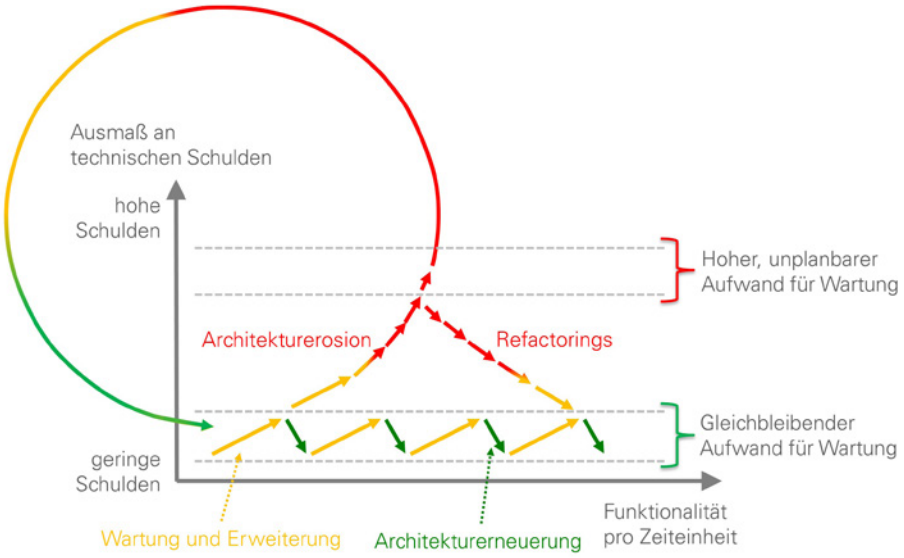

Abb. 1: Entwicklung und Effekt von technischen Schulden

Erweitert man das System mehr und mehr, so entstehen zwangsläufig technische Schulden (gelbe Pfeile in Abbildung 1 im Korridor gleichbleibender Aufwand). Softwareentwicklung ist ein ständiger Lernprozess, bei dem der erste Wurf einer Lösung selten der endgültige ist. Die Überarbeitung der Architektur (Architekturerneuerung, grüne Pfeile in Abbildung 1) muss in regelmäßigen Abständen durchgeführt werden. So entsteht eine stetige Folge von Erweiterung und Refactoring. Kann ein Team diesen Zyklus von Erweiterung und Refactoring dauerhaft verfolgen, so wird das System im Korridor geringer technischer Schulden bleiben (gelbe und grüne Pfeile in Abbildung 1 im Korridor „Gleichbleibender Aufwand für Wartung“).

Darf das Entwicklungsteam die technischen Schulden nicht kontinuierlich reduzieren, so setzt im Laufe der Zeit zwangsläufig Architekturerosion ein (gelbe und rote aufsteigende Pfeile in Abbildung 1). Sind erst einmal technische Schulden angehäuft, dann werden Wartung und Erweiterung der Software immer teurer, Folgefehler sind immer schwerer nachvollziehbar, bis zu dem Punkt, an dem jede Änderung zu einer schmerzhaften Anstrengung wird.

Abbildung 1 macht diesen langsamen Verfall dadurch deutlich, dass die roten Pfeile immer kürzer werden. Pro Zeiteinheit kann man bei steigenden Schulden immer weniger Funktionalität umsetzen.

Um erst gar nicht in dieses Dilemma zu kommen, brauchen wir einerseits das Wissen im Entwicklungsteam darüber, was eine qualitativ hochwertige und flexible Architektur ausmacht, und andererseits Methoden bei der Entwicklung, die zum Erhalten dieser Architektur beitragen.

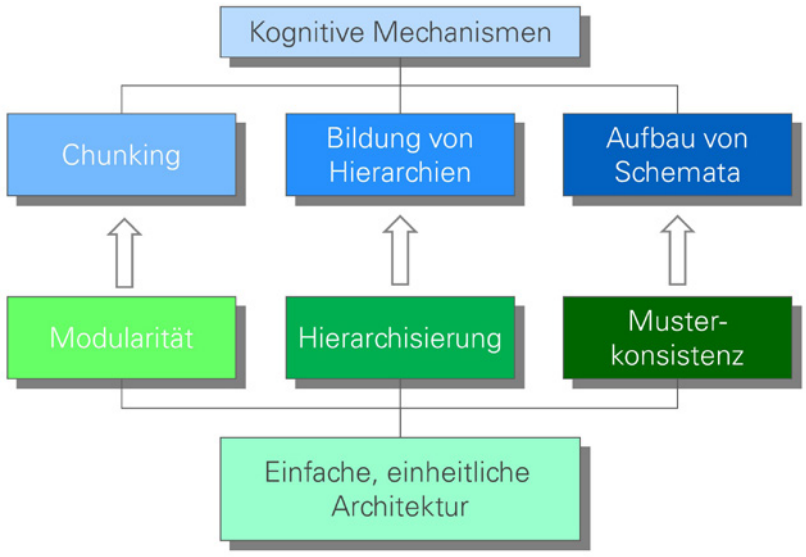

Abb. 2: Kognitive Mechanismen und ihr Abbild in Architektur

Kognitive Psychologie als Basis von langlebiger und erweiterbarer Architektur

Das menschliche Gehirn hat sich im Laufe der Evolution einige beeindruckende Mechanismen angeeignet, die uns beim Umgang mit komplexen Strukturen helfen. Diese Mechanismen gilt es in Softwaresystemen zu nutzen, damit Wartung und Erweiterung schnell und ohne viele Fehler von der Hand gehen. Das Ziel ist dabei, dass wir unsere Softwaresysteme auch mit sich verändernden Entwicklungsteams lange bei gleichbleibender Qualität weiterentwickeln können.

Die drei Mechanismen, die unser Gehirn für komplexe Strukturen entwickelt hat, sind: Chunking (dt. „Bündelung“), Bildung von Hierarchien und Aufbau von Schemata (siehe Abbildung 2). Diese Mechanismen haben direkte Abbilder in Kriterien für die Architektur.

Sind dem Entwicklungsteam diese Mechanismen und ihre Umsetzung in der Architektur klar, so ist eine wichtige Grundlage für eine hochwertige Architektur gelegt. Weitere Details zur Umsetzung von Modularität, Hierarchisierung und Musterkonsistenz finden Sie in [Lil15].

Mob Architecting als Methode für Architekturqualität

Um die Architektur auf diesem hohen Niveau zu halten, muss das Entwicklungsteam kontinuierlich an der Verbesserung der Architektur arbeiten. Ein gutes Mittel, um diese Arbeit an der Architektur zu unterstützen, ist Mob Architecting. Mit Mob Architecting lässt sich direkt am Quellcode überprüfen, inwieweit die geplante Architektur im Quellcode tatsächlich umgesetzt worden ist (siehe Abbildung 3) und wie hoch der Verschuldungsgrad der Software ist.

Die Soll-Architektur ist der Plan für die Architektur, der auf Papier oder in den Köpfen des Architekten und der Entwickler existiert. Für solche Soll-Ist-Vergleiche stehen heute eine Reihe guter Tools zur Verfügung: Lattix, Sotograph/SotoArc, Sonargraph, Structure101 u. v. m.

In Abbildung 4 sieht man den Ablauf eines Mob Architecting. Mob Architecting wird von einem Piloten gemeinsam mit allen Architekten und Entwicklern des Systems in einem Workshop durchgeführt. Zu Beginn des Workshops wird der Quellcode des Systems mit dem Analysewerkzeug geparst (1) und so die Ist-Architektur erfasst. Auf die Ist-Architektur werden nun verschie-

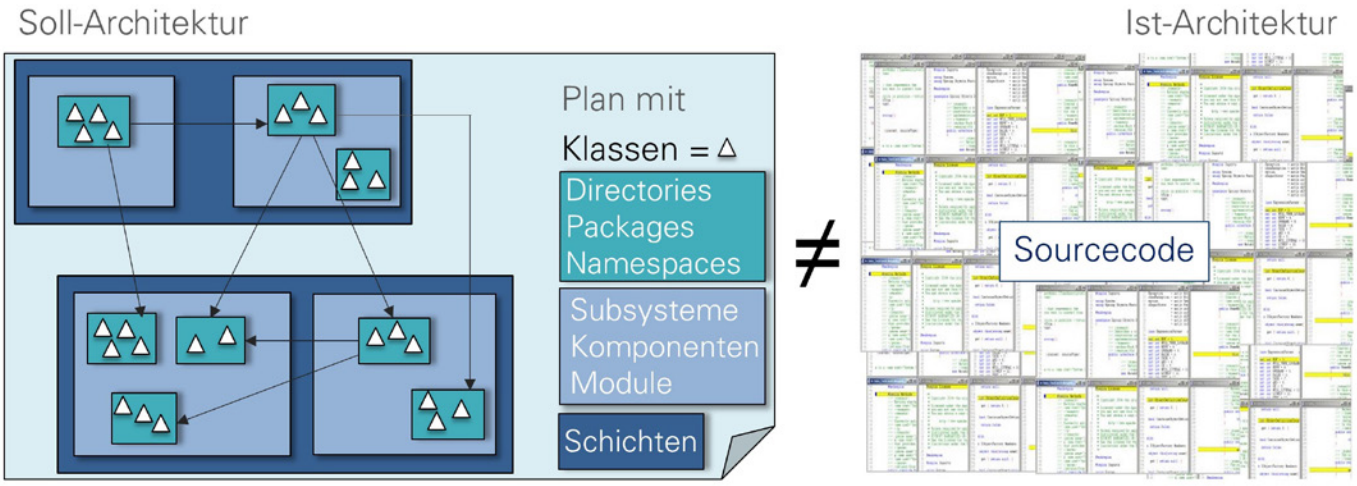

Abb. 3: Soll-Ist-Vergleich der Architektur

dene Sichten auf die Soll-Architektur modelliert, sodass der Vergleich von Soll und Ist möglich wird (2).

Dabei werden technische Schulden sichtbar und der Pilot macht sich gemeinsam mit dem Entwicklungsteam auf die Suche nach einfachen Lösungen, wie die Ist-Architektur durch ein Refactoring an die Soll-Architektur angeglichen werden kann (3). Oder aber der Pilot und das Entwicklungsteam stellen in der Diskussion fest, dass die im Quellcode gewählte Lösung besser ist als der ursprüngliche Plan.

Manchmal ist aber auch weder die Soll-Architektur noch die abweichende Ist-Architektur die beste Lösung und Pilot und Entwicklungsteam müssen gemeinsam ein neues Zielbild für die Architektur entwerfen.

Insbesondere, wenn ein System vor einer größeren Erweiterung steht, tritt dieser Fall ein. Die geplante Architektur war für die anstehende Erweiterung nicht ausgelegt, sodass eine grundlegende Weiterentwicklung der Architektur notwendig wird.

Dabei werden Fragen beantworten, wie: Ist die vorhandene Architektur flexibel genug für die Erweiterung? Muss die Kopplung an einigen Stellen reduziert werden, damit eine Umstrukturierung möglich wird? Ist der Schnitt der Module und

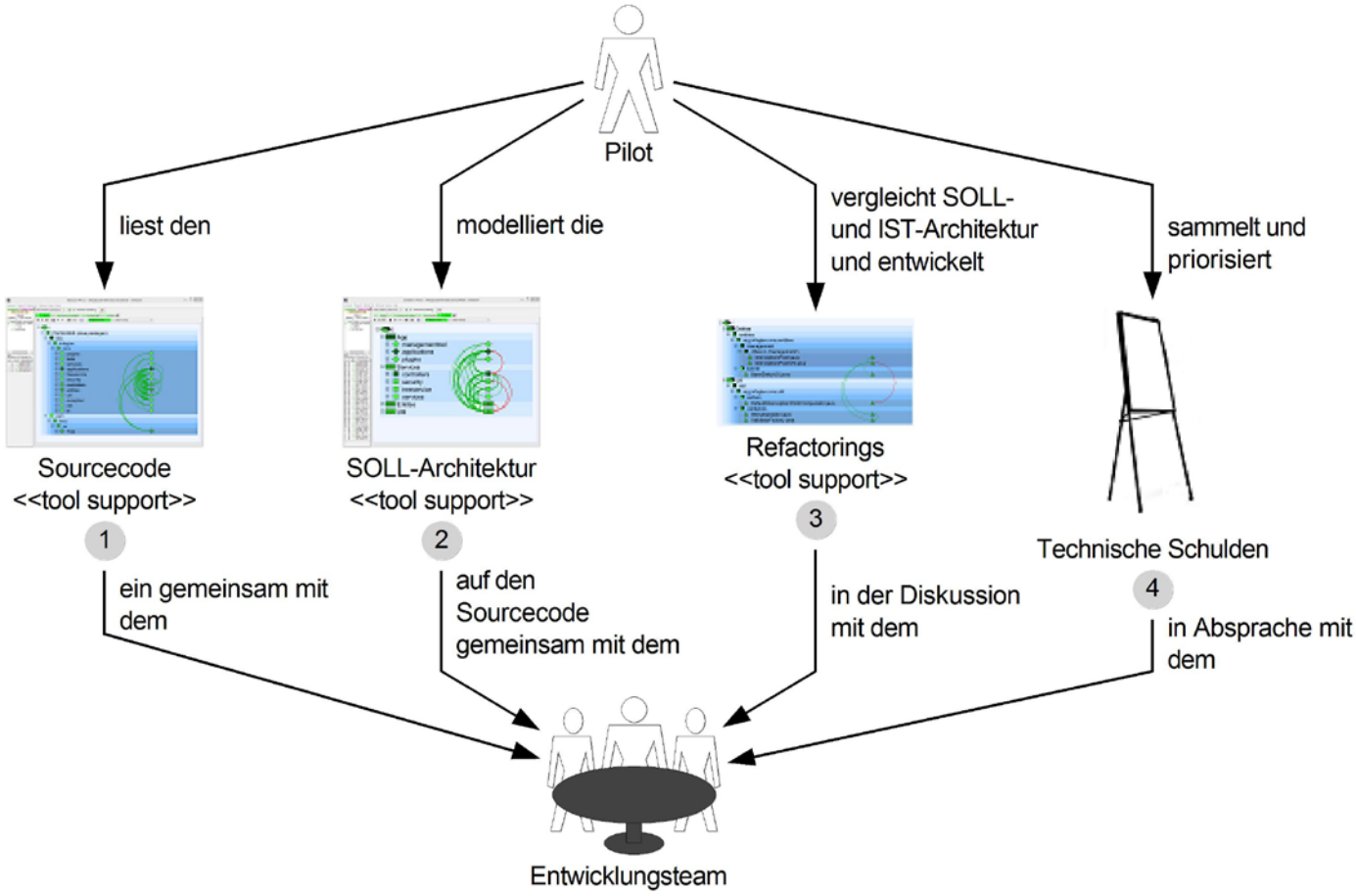

Abb. 4: Mob Architecting mit einem Entwicklungsteam

Schichten richtig gewählt, um die neuen Module mit der neuen Funktionalität konsistent zu integrieren?

Hat man ein Softwaresystem übernommen, das man nicht selbst mitentwickelt hat, dann lohnt sich auf jeden Fall ein Workshop mit einem tiefgehenden Mob Architecting, um das System überhaupt erst einmal kennenzulernen. Ist die Quellcode-Basis dem Entwicklungsteam unbekannt, so können auf diese Weise die ursprünglich geplanten Strukturen überhaupt erst sichtbar gemacht werden.

Im Laufe eines solchen Mob Architectings sammelt der Pilot mit dem Entwicklungsteam technische Schulden und mögliche Refactorings (4). Gemeinsam werden die Refactorings priorisiert und ihre Umsetzung geplant.

Zusammenfassung

In diesem Artikel habe ich sehr kurz und knapp beschrieben, was ein Entwicklungsteam wissen und können muss, um seine Architektur in einem guten Zustand zu erhalten. Einerseits muss das Entwicklungsteam wissen, was eine modulare, hierarchische und musterkonsistente Architektur ausmacht. Diese Grundkonzepte werden für das jeweilige System in einen Plan – die Soll-Architektur – und schließlich in den Quellcode, also die Ist-Architektur gegossen. Im Laufe der Entwicklung und Wartung braucht das Entwicklungsteam die Möglichkeit, Mob Architecting einzusetzen, um Abweichungen von Soll- und Ist-Architektur zu überprüfen und ggf. Refactorings auszuarbeiten. Werden diese beiden Aspekte beherzigt und dem Team die notwendigen Mittel zur Verfügung gestellt, wird jedes Softwaresystem langlebig und erweiterbar bleiben. ■

Literatur

[Cun92] Ward Cunningham: The WyCash Portfolio Management System. Experience Report, OOPSLA ,92, 1992.

[Lil15] Carola Lilienthal: Langlebige Softwarearchitekturen – Technische Schulden analysieren, begrenzen und abbauen, dpunkt Verlag 2015.