CONTINUOUS INTEGRATION FÜR SOFTWAREARCHITEKTUREN

In modernen Continuous-Integration-Ansätzen spielt Softwarearchitektur bisher keine Rolle. Neue und leicht verfügbarer Werkzeuge könnten das ändern.

| von ANDREAS RAQUET

Softwareentwicklung ist nach wie vor eine äußerst herausfordernde Disziplin. Entwickler bauen – allen Analogien zur Industrialisierung zum Trotz – selbst größte IT-Systeme manuell aus einzelnen Codezeilen auf. Die Softwaretechnik unternimmt erhebliche Anstrengungen, das sich daraus ergebende Fehlerpotenzial einzudämmen, damit robuste und verlässliche Software entstehen kann. Noch in den 1990er-Jahren wurde eine von einem Programmierer neu entwickelte Softwarekomponente erst in Integrationstests einer tieferen Qualitätssicherung unterzogen. Heute löst jeder Commit eine ganze Kette von automatischen Überprüfungsschritten einer Continuous-Integration-Pipeline aus. Diese schließt einen erheblichen Teil möglicher Fehlerquellen frühzeitig aus – angefangen von der Kompilierung des Gesamtsystems über die Durchführung und Überdeckungsmessung der Unit-Tests bis hin zur Prüfung von Codierungsstandards und der Vermeidung schlecht strukturierter Codes (code smells). Das Ergebnis: Klassische Integrationstests gibt es kaum noch, und wenn doch, liefern sie kaum interessante Ergebnisse.

FUNKTIONIERT DAS AUCH FÜR ARCHITEKTUR?

Völlig außen vor ist bei diesem Prozess leider die Softwarearchitektur. Eine automatische Prüfung der Einhaltung der Softwarearchitektur im Programmcode findet schlicht nicht statt. An ihre Stelle treten gelegentliche manuelle Code-Reviews mit mangelhaftem Deckungsgrad und eingeschränkter Zuverlässigkeit.

Dabei ist das Problemfeld bezüglich der Softwarearchitektur exakt das gleiche wie bei fachlichen Anforderungen oder der Qualität von Quellcode: Vorgaben werden auf abstrakter Ebene formuliert und dann durch viele Entwickler manuell in Quellcode übertragen. Die Einhaltung der Vorgaben hängt zunächst einmal von der Erfahrung und der Umsicht der Entwickler ab. Und so finden sich selbst in nagelneuen IT-Systemen mit Sicherheit bereits Architekturverstöße. Und im Laufe der Lebenszeit eines IT-Systems verschlimmert sich die Situation nur weiter – das IT-System altert.

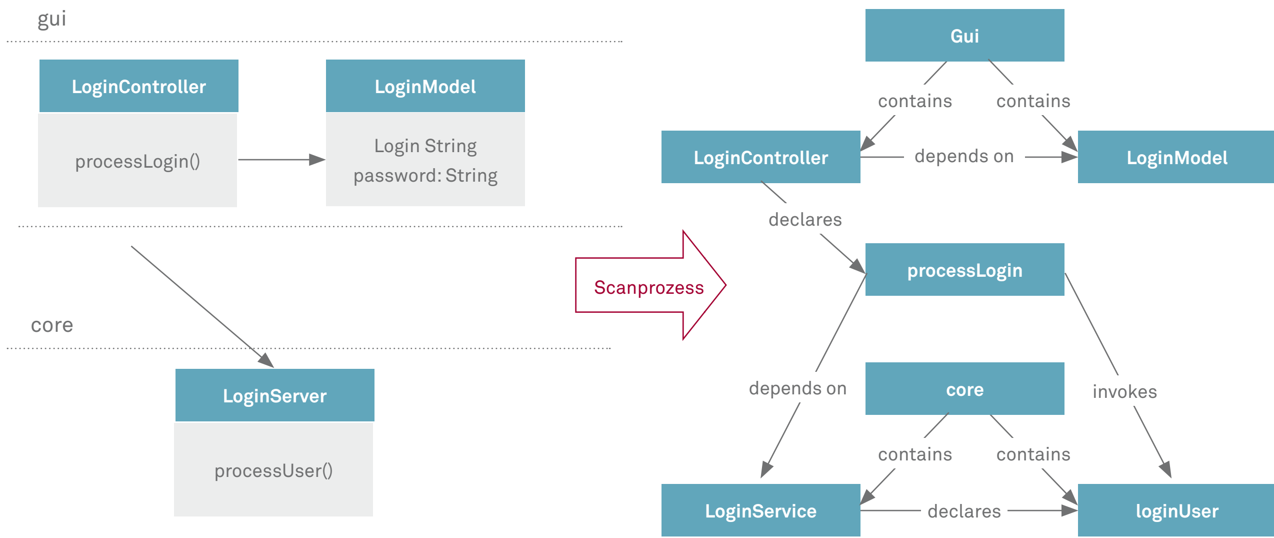

Abbildung 1: Einlesen der Codestruktur in einen Graphen

Technische Werkzeuge für eine Überprüfung der Architektur gibt es bereits seit einigen Jahren. Allerdings sind diese recht teuer und nicht primär für die Einbindung in einen Continuous-Integration-Prozess gedacht. Dieser Artikel betrachtet daher einen alternativen Ansatz, der im Wesentlichen auf den Open-Source-Werkzeugen jQAssistant[1] und Neo4J[2] basiert.

KANN MAN ARCHITEKTUR ÜBERHAUPT AUTOMATISIERT PRÜFEN?

Vor der Betrachtung, wie Architekturüberprüfung mittels Werkzeugen unterstützt und automatisiert werden kann, muss zunächst geklärt werden, ob und unter welchen Umständen das überhaupt möglich ist. Die wichtigste Voraussetzung dafür ist, dass die Architektur operationalisierbar ist. Das heißt, es muss eine klare Vorschrift geben, wie die abstrakten Konzepte der Architektur – Schichten, Bausteine, Services – auf die konkreten Artefakte der Programmierung abzubilden sind – in Java also beispielsweise in Packages, Klassen und Methoden. Ohne diese Vorschriften können Entwickler zwar etwas in die Architekturbilder hineininterpretieren. Das wird bei unterschiedlichen Entwicklern aber vermutlich ganz unterschiedlich ausfallen und kann schon gar nicht automatisiert geprüft werden. Die meisten Architekturen verfügen jedoch über eine solche Vorschrift oder können leicht darum ergänzt werden. Schließlich verfolgen Architekturen konkrete Ziele für das entstehende IT-System. Diese können nur erreicht werden, wenn die Architektur sich auch im Programmcode des IT-Systems wiederfindet. Dieses „Wiederfinden" der Architektur im Programmcode ist allerdings eine der größten Herausforderungen bei der Architekturüberwachung. Dazu später mehr.

DIE MACHT DER GRAPHEN

Die konkrete Überprüfung der Architekturkonformität steht zunächst vor dem Problem, dass das „Soll" – nämlich die Architektur – und das „Ist" – nämlich der Programmcode – auf unterschiedlichen Abstraktionsebenen und in unterschiedlichen „Sprachen" vorliegen: Architektur wird primär mittels verschiedener grafischer Notationen und natürlicher Sprache kommuniziert, Programmcode in einer oder mehreren Programmiersprachen, teilweise auch in Markup-Sprachen wie XML. Für einen Soll-Ist-Vergleich ist es wichtig, beides, also Architektur und Programmcode, in dieselbe Sprache zu übersetzen. Dazu bieten sich Graphen an:

1. Graphen sind die „Lingua Franca" der Architektur. Praktisch jede Notation für Softwarearchitekturen basiert auf Graphen, sei es die UML, Archimate oder einfach „Kästchen und Pfeile".
2. Aus dem Compilerbau ist außerdem bekannt, dass sich Programmcode sehr gut als Graph darstellen lässt, nämlich als Syntaxbaum.

1 https://jqassistant.org
2 https://neo4j.com/

Mittels dieser beiden Ansätze lassen sich Architektur (Soll-Architektur) und Programmcode (Ist-Architektur) in Graphen übersetzen und auf ein gemeinsames Abstraktionsniveau transformieren.

Sind die Soll-Architektur und die Ist-Architektur in dieselbe Sprache transformiert und auf dasselbe Abstraktionsniveau gebracht, muss nur noch verglichen werden, ob beide konsistent sind, und die Ergebnisse so aufbereitet werden, dass sie durch das Entwicklerteam verstanden und sinnvoll weiterverarbeitet werden können.

UMSETZUNG MIT MODERNEN WERKZEUGEN

Auch wenn das Konzept intuitiv nachvollziehbar erscheint, ist die Umsetzung viel schwieriger und aufwendiger, als es der erste Blick vermuten lässt. Zunächst muss der Programmcode des IT-Systems in einen Syntaxbaum transformiert (geparst) werden. Außerdem muss man sich mit dem Erzeugen, der Transformationen und dem Abgleich von Graphen auseinandersetzen. Glücklicherweise gibt es jedoch mittlerweile fertige Open-Source-Werkzeuge, die diese Aufgaben bewältigen:

1. Neo4j ist eine Graph-Datenbank. Doch anders, als der Name Datenbank suggeriert, geht es nicht primär um das sichere Speichern von Graphen, sondern um das effiziente Erzeugen und Analysieren von solchen. Dazu bringt Neo4J die einfach erlernbare Sprache Cypher mit, sowie eine Web-basierte Benutzeroberfläche, die Graphen erzeugen, aber auch transformieren und vergleichen kann.

2. jQAssistant übernimmt im Grunde den Rest des Ansatzes:
   a. Es liest den Java-Programmcode ein und erzeugt einen Syntaxbaum in Neo4j.
   b. Es ermöglicht die Transformation in einen Ist-Architektur-Graphen und den Aufbau des Soll-Architektur-Graphen mittels Cypher-Anweisungen. jQAssistant nennt diese Anweisungen „Konzepte".
   c. Es erlaubt die Prüfung der Ist-Architektur oder den Vergleich von Soll- und Ist-Architektur mit Cypher-Abfragen. jQAssistant nennt diese Abfragen „Regeln".
   d. Es bereitet die Ergebnisse entsprechend auf.

Der gesamte Prozess kann mittels Plugin in den Continuous-Integration-Prozess integriert werden. Regelverletzungen können entweder den Build abbrechen – und damit ähnlich wie ein Kompilierfehler behandelt werden – oder sie liefern nur Warnungen, die dann, ähnlich wie fehlerhafte Unit Tests, durch die Entwickler behandelt werden können.

KANN DAS WIRKLICH SO EINFACH SEIN?

Der Ansatz steht und fällt mit der Formulierung der Konzepte und Regeln. Dabei sollten einige Randbedingungen beachtet werden:

Es ist nicht sinnvoll und auch nicht möglich, die gesamte Architektur in jQAssistant zu modellieren und zu überprüfen:

- Um ein Architekturkonzept zu überwachen, muss es im Quellcode als solches erkennbar sein. Wenn zum Beispiel eine Architekturkomponente „Controller" in der entsprechenden Klasse im Programmcode (versehentlich) nicht entsprechend gekennzeichnet (annotiert) wurde, kann die Klasse nicht als Controller erkannt werden.
- Eine Architektur enthält immer auch einen Teil Feindesign für technische Komponenten. Es lohnt in der Regel nicht, diese Komponenten zu überwachen, da eine Verletzung des Feindesigns, anders als Architekturverletzungen auf gröberen Ebenen, keine Auswirkungen auf das Gesamtsystem hat. Zudem ist eine Abbildung des Feindesigns unverhältnismäßig aufwendig.

Empfehlenswert ist es also, sich zunächst auf die Grobarchitektur zu konzentrieren und Regeln mit hoher normierender Wirkung zu definieren:

1. Die Schichtenarchitektur sollte eingehalten werden.
2. Komponenten sollten nur über ihre Schnittstellen gekoppelt sein.
3. Vertikale Schnitte[3] dürfen keine gemeinsamen Entitäten oder Relationen verwenden.

Es ist nicht zwingend notwendig, eine Soll-Architektur als Graph zu modellieren. Stattdessen genügt es auch, die Konsequenzen aus der Soll-Architektur direkt als Regeln für die Ist-Architektur zu formulieren. Es ist zunächst einfacher, eine Regel der Art „Der Anwendungskern darf nicht von der GUI-Schicht abhängen" zu formulieren, als alle Schichten in der Soll-Architektur zu modellieren und diese generisch mit der Ist-Architektur zu vergleichen. Der Mächtigkeit dieser „Ist-Architektur-Regeln" sind aber Grenzen gesetzt. Auf Dauer ist es daher sinnvoll, auch in die Modellierung der Soll-Architektur einzusteigen.

Um diese beiden Strategien zur Formulierung der Regeln zu unterscheiden, haben sich folgende Begriffe eingebürgert:

3 https://en.wikipedia.org/wiki/Vertical_slice

- Im Blacklisting-Ansatz formulieren Regeln unmittelbar Verbote auf der Ist-Architektur. Jede Regel stellt also ein unerlaubtes Muster dar.
- Im Whitelisting-Ansatz wird zunächst die Soll-Architektur als Gesamtheit der erlaubten Architekturmuster formuliert. Die Regeln vergleichen dann Soll- und Ist-Architektur.

In der Praxis werden auf Dauer beide Ansätze benötigt und in Kombination eingesetzt.

REFERENZARCHITEKTUR HILFT BEI REGELSATZERSTELLUNG

Die Erstellung eines initialen Regelsatzes dauert wenige Tage bis Wochen, abhängig von der verfügbaren Architekturdokumentation. Es ist ratsam, mit einem kleinen Regelsatz zu beginnen und diesen sukzessive zu verfeinern. Fertige Regelsätze, die man einfach als Ausgangsposition nutzen könnte, gibt es aufgrund der Unterschiedlichkeit der Architekturen nicht.

Einen guten Ausweg bietet allerdings der Einsatz einer Referenzarchitektur, also einer standardisierten Architektur, für die bereits ein Regelsatz existiert oder zumindest nur einmalig entwickelt werden muss. Ein Beispiel für eine solche Referenzarchitektur ist der IsyFact-Standard des Bundesverwaltungsamts.[4] Wer eine Referenzarchitektur wie IsyFact nutzt, profitiert nicht nur von den positiven Eigenschaften der Referenzarchitektur selbst, sondern bekommt die zugehörige Konformitätsüberwachung gleichsam dazu geschenkt.

EINSCHRÄNKUNGEN DER WERKZEUGE

Die vielleicht wichtigste Einschränkung des Ansatzes besteht darin, dass jQAssistant derzeit nur Java Bytecode und XML parsen kann. Insbesondere wird kein JavaScript unterstützt, sodass moderne Clients, wie zum Beispiel Single Page Applications (SPAs), damit nicht analysiert werden können. jQAssistant kann zwar über ein Plugin-Konzept erweitert werden, die Bereitstellung eines JavaScript-Plugins dürfte aber nicht ganz trivial sein.

Eine weitere Schwäche besteht in der Beschränkung auf die statische Architektur. jQAssistant liest den Syntaxbaum aus dem Programmcode. Damit lässt sich nicht viel über das Laufzeitverhalten der Anwendung in Erfahrung bringen. Architektur kennt aber auch insbesondere die sogenannte Laufzeitsicht. Diese ließe sich leicht als Soll-Architektur in den Graphen laden, aber die zugehörige Ist-Architektur fehlt. In der Praxis fällt das allerdings wenig ins Gewicht. Die meisten Architekturen konzentrieren sich ohnehin auf die statische Komponentensicht, die sich ausgezeichnet in jQAssistant abdecken lässt.

4 http://www.bva.bund.de/DE/Organisation/Abteilungen/Abteilung_BIT/Leistungen/IT_Produkte/IsyFact/node.html

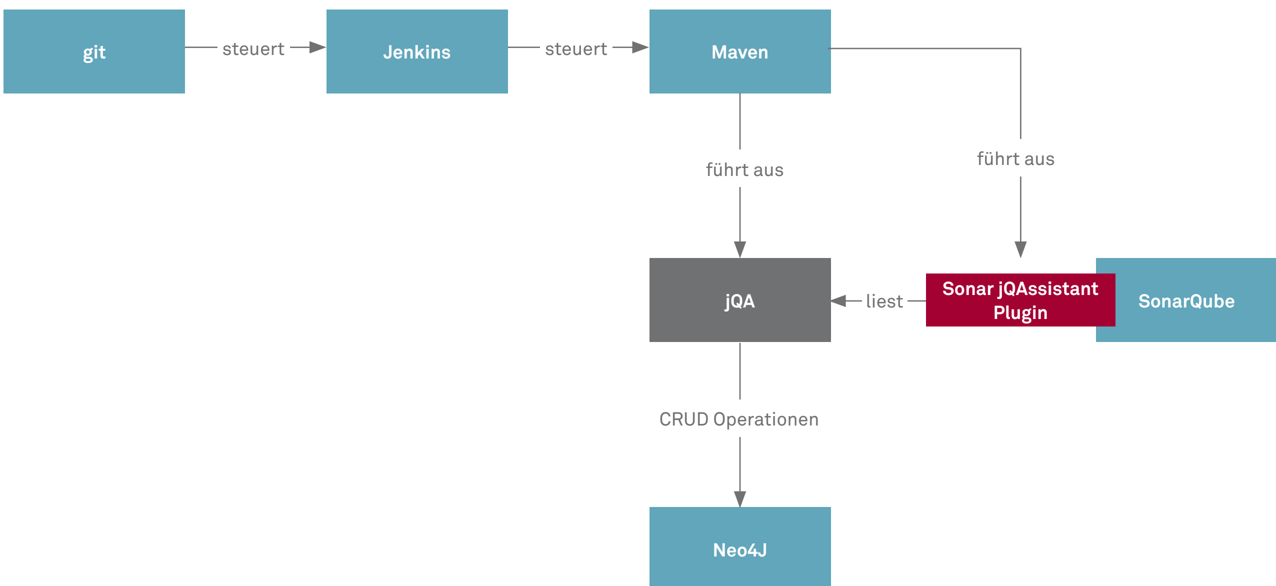

Abbildung2: Aufbau der Continuous-Integration-Tool-Kette mit jQAssistant und Sonar-jQAssistant-Plugin

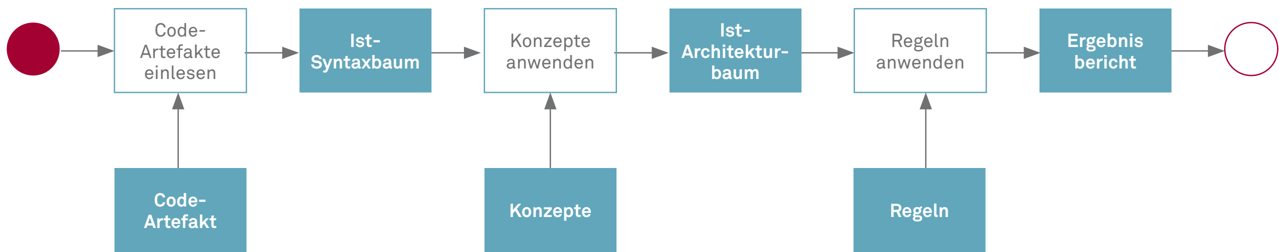

Abbildung 3: Ablauf des Überprüfungsprozesses

VERBESSERTE ANALYSIERBARKEIT DURCH JQA-SONAR-PLUGIN

Bisher klaffte im Continuous-Integration-Ansatz mit jQAssistant noch eine Lücke: Zwar liefert jQAssistant seine Befunde in aufgeräumter Form in einer Webansicht. Die Stakeholder dieser Berichte – Architekten und Qualitätsmanager – arbeiten jedoch nicht oder nur ungerne mit proprietären Formaten. Viel ansprechender wäre es, die Befunde aus der Architekturüberwachung direkt in die Überwachungssysteme zu importieren, die ohnehin für anderen Qualitätsmerkmale verwendet werden.

Hier nimmt das Werkzeug SonarQube mittlerweile eine zentrale Position ein: Es läuft im Continuous-Integration-Prozess mit, sammelt Befunde aus den vorhergehenden Verarbeitungsschritten und führt auf der Grundlage einer umfassenden Regelbasis weitere Codeüberprüfungen durch.

Auch hier sind Architekturüberprüfungen zunächst Fehlanzeige. Ein von msg weiterentwickeltes Plugin baut nun jedoch eine Brücke zwischen jQAssistant und SonarQube: Es sammelt Architekturverstöße aus der Architekturüberwachung und spielt diese als Issues in SonarQube ein. Dort werden die Verstöße zusammen mit allen anderen Befunden aufbereitet und können weiterverarbeitet werden, zum Beispiel, indem ein Ticket zur Bearbeitung geöffnet oder der entsprechende Befund als False Positive markiert wird. Das oben erwähnte jQA-SonarQube-Plugin von msg ist mittlerweile Teil der offiziellen jQAssistant Distribution 1.5.

FAZIT

Ein größeres Softwareentwicklungsprojekt ohne Continuous Integration ist heute kaum mehr denkbar. Die Methoden und Werkzeuge sind weithin bekannt, einfach verfügbar und nutzbar. Und die Wirksamkeit ist unbestritten. Der nächste logische Schritt ist nun, die bisher offene Flanke der Architekturüberwachung in die Continuous-Integration-Pipeline aufzunehmen. Open-Source-Werkzeuge wie jQAssistant und Neo4J machen das heute mit überschaubarem Aufwand möglich. Die Ergebnisse lassen sich mithilfe des jQA-Sonar-Plugin leicht in die bestehende Überwachung mit SonarQube integrieren. Weitere Integrationen können bei Bedarf geschaffen werden. Es ist zu hoffen, dass weitere, vergleichbare Werkzeuge in den Markt drängen und Architekturüberwachung so zu einer Selbstverständlichkeit der modernen Softwaretechnik wird. ●