Sonderdruck aus OBJEKTspektrum

Continuous Integration für Softwarearchitektur

Zeit, die Hausaufgaben zu machen!

In modernen Continuous-Integration-Ansätzen spielt Softwarearchitektur bisher kaum eine Rolle. Neue und leicht verfügbare Werkzeuge könnten das ändern.

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 90er Jahren wurde eine von einem Programmierer neu entwickelte Softwarekomponente erst im Integrationstest 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 – angefangen von der Kompilierung des Gesamtsystems über die Durchführung und Überdeckungsmessung der Unittests bis hin zur Prüfung von Codierungsstandards und Vermeidung schlecht strukturierten Codes (Code Smells) – frühzeitig aus.

Das Ergebnis: Klassische Integrationstests gibt es kaum noch und wenn doch, liefern sie kaum interessante Ergebnisse.

Funktioniert das auch für Architektur?

Weitgehend außen vor ist bei diesem Prozess die Softwarearchitektur. Eine automatische Prüfung der Einhaltung der Softwarearchitektur im Programmcode (**vgl. Abbildung 1**) findet daher in der Regel nicht statt. An ihre Stelle treten manuelle Codereviews, die naturgemäß weder den Deckungsgrad noch die Zuverlässigkeit einer automatisierten Überprüfung erreichen.

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 die Situation sich nur weiter – das IT-System altert.

Kommerzielle Werkzeuge für eine Überprüfung der Architektur gibt es bereits seit einigen Jahren. Für den Einsatz in größeren Entwicklungsteams haben sich diese aber bis heute nicht durchgesetzt (**siehe Kasten 1**).

Dieser Artikel betrachtet daher einen alternativen Ansatz, der auf den Open-Source-Werkzeugen jQAssistant und Neo4J basiert (siehe [JQA] und [Neo4J]).

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 es eine klare Vorschrift gibt, wie die abstrakten Konzepte der Architektur – Schichten, Bausteine, Services – auf die konkreten Artefakte der Programmierung abzu-

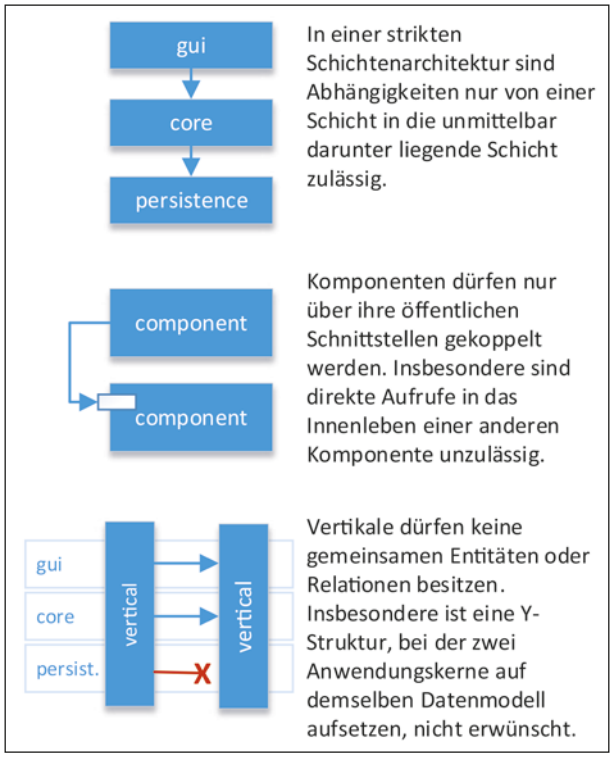

Abb. 1: Grundlegende Architekturregeln

Klassische Überwachungswerkzeuge wie Findbugs, Checkstyle und PMD sind für die Analyse und Überwachung der Softwarearchitektur nicht ausgelegt. Sie können nur rudimentäre Regeln wie die korrekte Package-Hierarchie überprüfen, jedoch weder Zyklen noch sonstige Abhängigkeitsmuster analysieren. Dafür bedarf es spezieller Werkzeuge zur Architekturanalyse.

Kommerzielle Produkte für diesen Zweck gibt es seit mehr als 10 Jahren. Die bekanntesten sind Lattix, Structure101 und SonarGraph. Sie werden häufig in der Analyse von Legacy Code eingesetzt. Die Codestrukturen werden grafisch aufbereitet und können so manuell mit der Soll-Architektur verglichen werden (siehe auch [Lil16]). Die Werkzeuge verfügen durchaus auch über Mechanismen zur Überwachung im Buildprozess. Dennoch sind sie nur selten als integraler Bestandteil der Softwareentwicklung anzutreffen.

Kasten 1: Bestehende Werkzeuge zur Architekturüberwachung

bilden sind, in Java also beispielsweise in Packages, Klassen und Methoden. Ohne diese Vorschriften können Entwickler zwar etwas in die Architekturbilder hineininterpretieren, das wird für unterschiedliche Entwickler aber vermutlich ganz unterschiedlich ausfallen und kann schon gar nicht automatisiert geprüft werden.
Die meisten Architekturen verfügen 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.

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 zunächst wichtig, Architektur und Programmcode in dieselbe Sprache zu übersetzen. Dazu bieten sich Graphen an:

- Graphen sind die „Lingua franca" der Architektur. Praktisch jede Notation für Softwarearchitekturen basiert auf Graphen, sei es UML oder einfach „Kästchen und Pfeile".
- Und aus dem Compilerbau wissen wir, dass sich Programmcode sehr gut als Graph darstellen lässt, nämlich als Syntaxbaum.

Mittels dieser beiden Ansätze lassen sich Architektur (Soll-Architektur) und Programmcode (Ist-Architektur) in Graphen übersetzen und auf ein gemeinsames Abstraktionsniveau transformieren. **Abbildung 2** zeigt beispielhaft, wie eine Graphendarstellung für ein einfaches Codefragment aussehen kann.
Sind die Soll- und die Ist-Architektur in dieselbe Sprache transformiert und auf dasselbe Abstraktionsniveau gebracht, muss nur noch verglichen werden, ob beide konsistent sind. **Abbildung 3** zeigt eine solche Gegenüberstellung von Soll- und Ist-Architektur. Beispielhaft ist der `LoginService` dabei im falschen Package gelandet und greift unerlaubt auf das `LoginModel` zu.

Umsetzung mit modernen Werkzeugen

Auch wenn das Konzept intuitiv nachvollziehbar erscheint, ist die Umsetzung viel schwieriger und aufwendiger, als es sich anhört. Zunächst muss der Programmcode des IT-Systems in einen Syntaxbaum

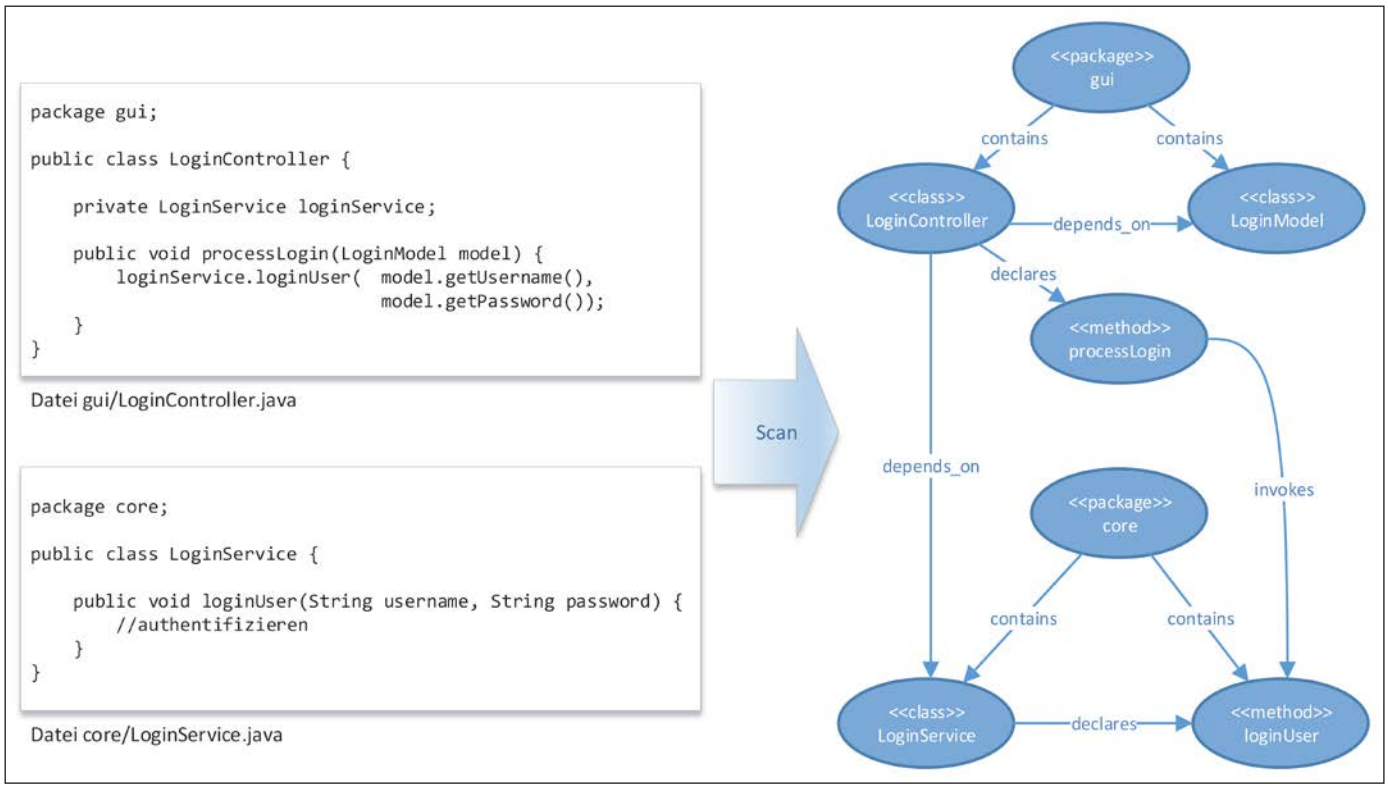

Abb. 2: Transformation von Code in eine Graphendarstellung

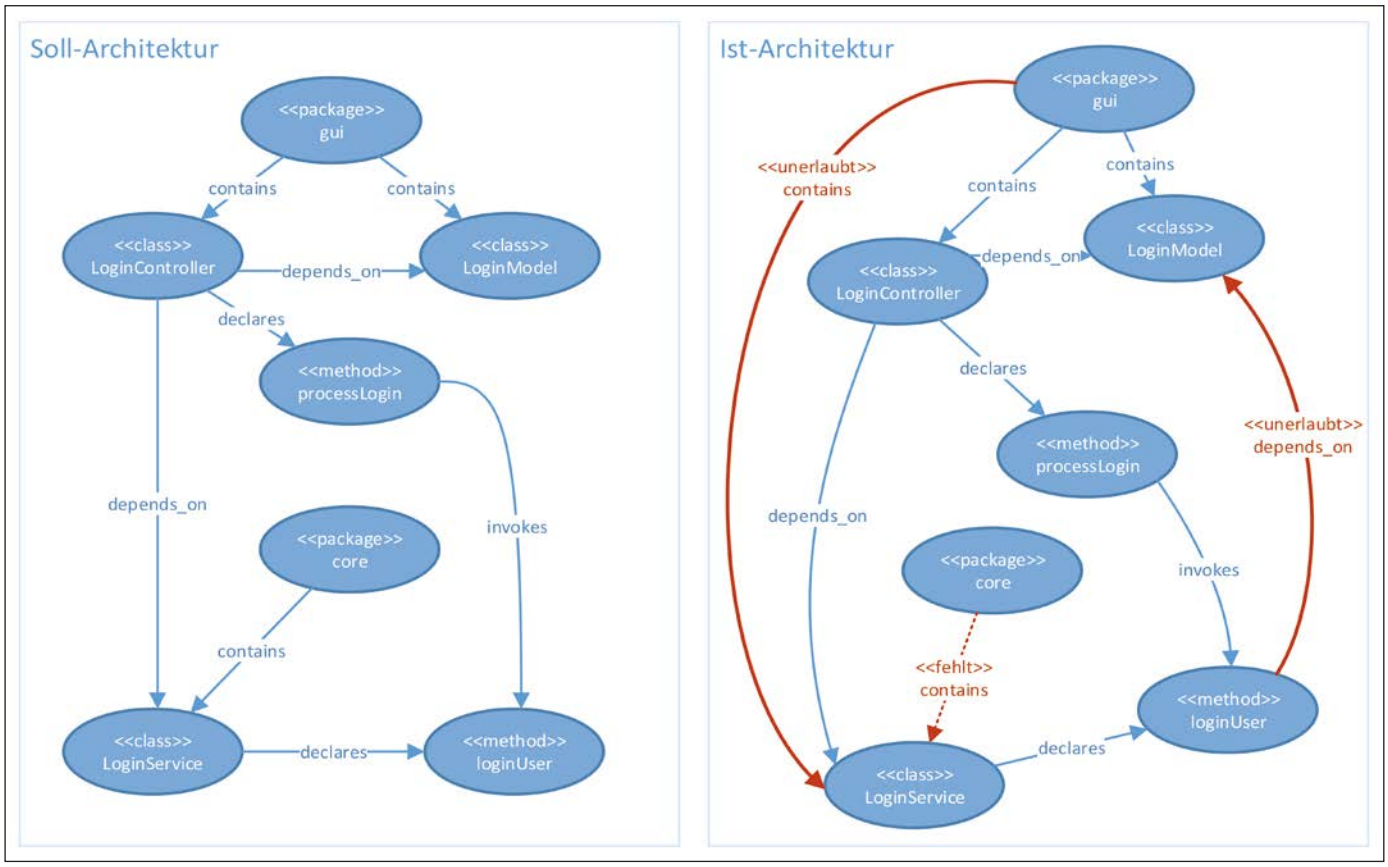

Abb. 3: Vergleich von Soll- und Ist-Architektur

transformiert (geparst) werden. Außerdem muss man sich mit der Erzeugung, der Transformationen und dem Abgleich von Graphen auseinandersetzen. Glücklicherweise gibt es jedoch mittlerweile fertige Open-Source-Werkzeuge, die diese Aufgaben erheblich vereinfachen.

Neo4j ist eine Graphdatenbank. Sie ist darauf spezialisiert, Graphen zu erzeugen und zu analysieren. Dazu bringt Neo4J die einfach erlernbare Sprache *Cypher* mit, sowie eine Web-basierte Benutzeroberfläche, die Graphen visualisieren kann. Damit lassen sich Graphen sehr leicht erzeugen, transformieren und vergleichen. Einen Überblick vermittelt **Kasten 2**.

jQAssistant übernimmt im Grunde den Rest des Ansatzes:

- Es liest den Java-Programmcode ein und erzeugt einen Syntaxbaum in Neo4j.
- Es reichert den Syntaxbaum mit den Konzepten der Architektur an und erzeugt so den Ist- und den Soll-Architekturgraphen. Die dazu verwendeten Cypher-Statements nennt jQAssistant „Concepts“.
- Es prüft die Ist-Architektur beziehungsweise vergleicht diese mit der Soll-Architektur mittels zu erstellender Cypher-Abfragen. jQAssistant nennt diese Abfragen „Constraints“.

Graphen in Neo4j

Cypher visualisiert Knoten und Kanten in Ascii-Art. `( )` bezeichnet einen Knoten, `-->` eine Kante. `( ) --> ( )` stellt zwei Knoten mit einer gerichteten Kante dazwischen dar.

Nützlich werden Knoten und Kanten erst, wenn man sie auch mit Informationen anreichert. Die einfachste Art sind Labels, mit denen man Knoten und Kanten markieren kann. Dazu schreibt man das Label einfach mit vorangestelltem Doppelpunkt in den Knoten oder die Kante.

`CREATE (:package) -[:contains]-> (:class)` erzeugt Knoten mit den Labels „package“ und „class“ und eine gerichtete Kante mit dem Label „contains“. Ganz ähnlich sehen übrigens die Graphenstrukturen aus, die jQAssistant aus dem Code generiert.

`MATCH (:package) -[:contains]-> (:class)` findet alle Knotenpaare mit Labels „package“ beziehungsweise „class“, die durch eine „contains“-Kante verbunden sind.

Natürlich können die Graphenausdrücke auch deutlich komplexer sein und beispielsweise durch transitive Abhängigkeiten und Filterkriterien ergänzt werden. Die folgende Anweisung findet alle Methodenaufrufe zwischen zwei unterschiedlichen Packages:

```
MATCH (p1:package) -[:contains*]->
  (:class) -[:declares]-> (:method) -[:invokes]->
  (:method) <-[:declares]- (:class) <-[:contains*]- (p2:package)
WHERE NOT p1 = p2 RETURN p1.name, p2.name
```

Der * zeigt dabei an, dass auch transitive contains-Beziehungen gefunden werden sollen, also ganze Kantenzüge aus contains-Kanten. Die WHERE-Klausel funktioniert analog zum SQL-Pendant.

Kasten 2: Die Abfragesprache Cypher

- Es bereitet die Ergebnisse in Form eines HTML-Reports auf.

In **Abbildung 4** ist der gesamte Ablauf schematisch dargestellt. Der Überprüfungsvorgang kann über ein Maven-Plug-in in den Buildprozess eingebunden 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 Unittests, durch die Entwickler behandelt werden können.

Kann das wirklich so einfach sein?

Der Ansatz steht und fällt mit der Formulierung der Konzepte und Regeln. Hier sind zunächst einige Randbedingungen zu nennen.
Es ist weder sinnvoll noch 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. Dazu eignen sich beispielsweise die Regeln aus **Abbildung 1**.
Weiter ist es 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:

- 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 (siehe [Isy]), in dessen Umfeld der hier vorgestellte Ansatz entstanden ist. Wer eine Referenzarchitektur wie IsyFact nutzt, profitiert nicht nur von den positiven Eigenschaften der Referenzarchitektur selbst, sondern bekommt die zugehörige Konformitätsüberwachung gleichsam dazu geschenkt.
Ein solcher Regelsatz für eine real existierende Architektur ist natürlich ungleich komplexer, als die hier vorgestellten Beispiele. Wer einen Eindruck gewinnen möchte, kann einen Blick auf den Regelsatz der IsyFact auf GitHub werfen (siehe [IRule]).

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 Single-Page Applications (SPAs) damit nicht analysiert werden können. jQAssistant kann zwar über ein Plug-in-Konzept erweitert werden, die Bereitstellung eines JavaScript-Plug-ins dürfte aber nicht ganz trivial sein.
Eine weitere Schwäche besteht in der Beschränkung auf die statische Architektur. jQAssistant liest den Syntaxbaum aus

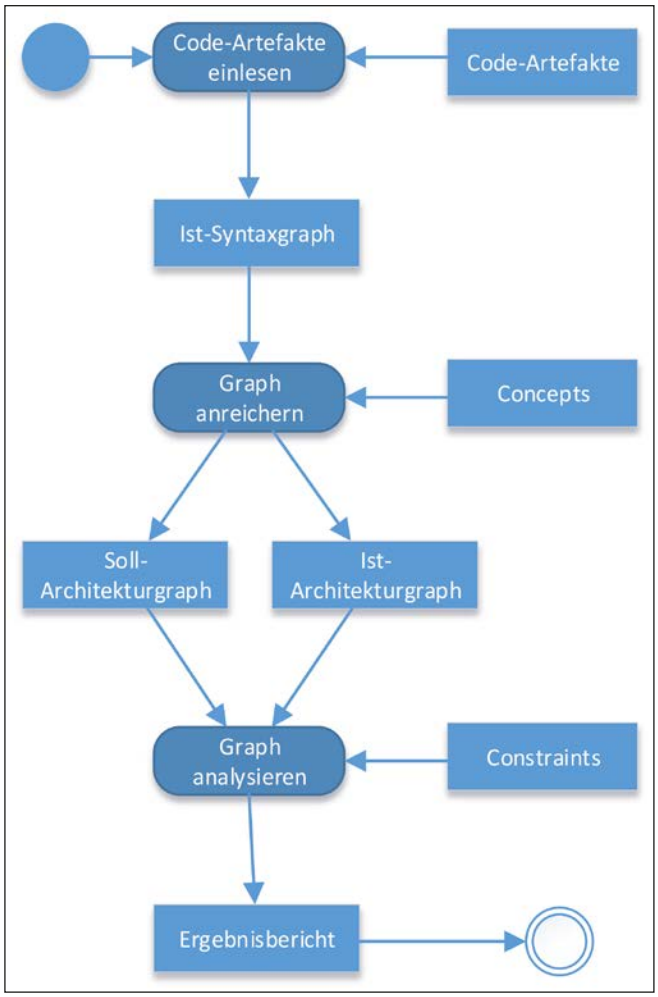

Abb. 4: Ablauf des Überprüfungsprozesses mit jQAssistant

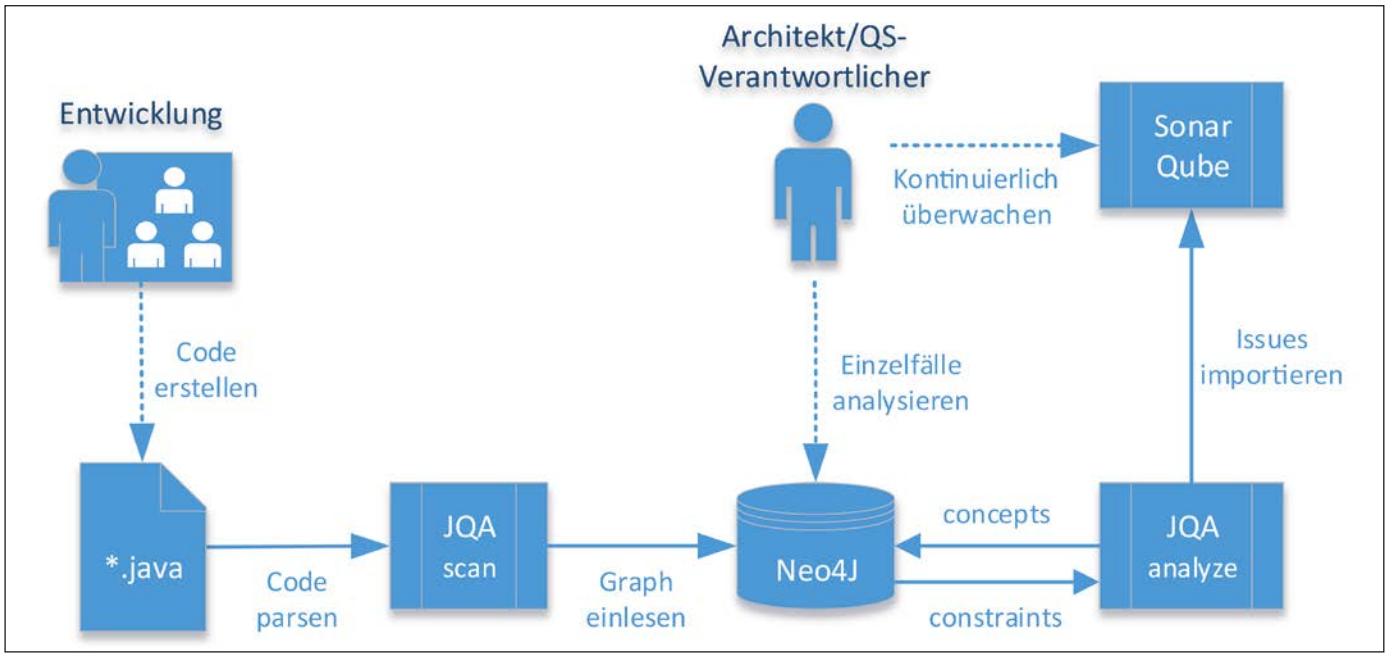

Abb. 5: Integrierte Architekturüberwachung mit jQAssistant und SonarQube

dem Programmcode. Damit lässt sich nicht viel über das Laufzeitverhalten der Anwendung in Erfahrung bringen. Die 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, welche sich ausgezeichnet in jQAssistant abdecken lässt.

Verbesserte Analysierbarkeit durch SonarQube-Integration

Bisher klaffte im Continuous-Integration-Ansatz mit jQAssistant noch eine Lücke: Zwar liefert jQAssistant seine Befunde in einem aufgeräumten Report. Die Stakeholder dieser Berichte – Architekten und Qualitätsmanager – arbeiten jedoch 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.

Ein sehr verbreitetes Überwachungssystem ist *SonarQube*: Es läuft im Continuous-Integration-Prozess mit, sammelt Befunde aus den vorhergehenden Verarbeitungsschritten und führt auf der Grundlage einer umfassenden Regelbasis weitere Codeüberprüfungen durch (**siehe Abbildung 5**).
SonarQube selbst kennt praktisch keine Architekturregeln. Seit Kurzem ist nun ein Sonar-Plug-in als Teil der jQAssistant Distribution verfügbar, das die Ergebnisse aus der Architekturanalyse als Issues in SonarQube übernimmt. 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.

Fazit

Ein größeres Softwareentwicklungsprojekt ohne Continuous Integration ist heute kaum mehr denkbar. Die Methoden und Werkzeuge sind weithin bekannt, einfach verfügbar und nutzbar. 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 leicht in die bestehende Überwachung mit SonarQube integrieren. Weitere Integrationen können bei Bedarf geschaffen werden.
Es steht zu hoffen, dass weitere vergleichbare Werkzeuge in den Markt drängen und Architekturüberwachung so zu einer ähnlichen Selbstverständlichkeit der modernen Softwaretechnik wird, wie Unittesting und statische Codeanalyse das heute sind. ||

Literatur & Links

[IRule] Regelsatz der IsyFact, siehe: **https://github.com/IsyFact/isyfact-jqassistant-plugin**

[Isy] Referenzarchitektur des Bundesverwaltungsamts, siehe: **http://isyfact.de**

[JQA] Homepage von jQAssistant, siehe: **https://jqassistant.org/**

[Lil16] C. Lilienthal, Langlebige Softwarearchitekturen, dpunkt.verlag, 2016

[Neo4J] Homepage von Neo4J, siehe: **https://neo4j.com/**

Der Autor

Andreas Raquet
(andreas.raquet@msg.group)
arbeitet als IT Consultant für die msg systems AG. Als technischer Architekt verantwortet er die Umsetzung von IT-Großprojekten für die öffentliche Verwaltung. Er hat 20 Jahre Erfahrung in der Entwicklung betrieblicher Informationssysteme