Strukturelle Integrität

jQAssistant mit Neo4j – Codeanalyse zur Absicherung von Strukturen in Java-Projekten

Dirk Mahler

Neben technologischen Entwicklungen ist Qualitätssicherung das Schlagwort, welches sich mit Beständigkeit in Publikationen und auf den Agenden von Fachkonferenzen wiederfindet. Dabei werden Ansätze demonstriert, welche auf verschiedensten Ebenen ansetzen: Vorgehensmodell, Anforderungsanalyse, Teststrategie, Build- und Integrationsmanagement oder statische Codeanalyse. Für den letztgenannten Bereich soll eine bestehende Lücke betrachtet und mit einem Lösungsansatz gefüllt werden: die Definition und Validierung projektspezifischer Regeln, zum Beispiel zur Durchsetzung von Architekturregeln und Namenskonventionen. Als technische Basis dafür soll die Graphdatenbank Neo4j dienen.

Problem Komplexität

Am Anfang sollen zunächst einige Zahlen stehen: Das Release 8.1.0.Final des Applikationsservers WildFly aus dem Hause JBoss besteht inklusive Testcode aus ca. 600 Packages, welche 10.800 Class-Dateien beinhalten, 89 Prozent davon repräsentieren Klassen, 8 Prozent Interfaces. Bewegt man sich eine Strukturebene tiefer, gelangt man zu ca. 36.000 Feld- beziehungsweise 77.000 Methodendeklarationen. Der nächste Drilldown-Schritt in die Methodenimplementierungen hinein liefert 140.000 Zugriffe auf Instanz- oder statische Variablen sowie 350.000 Methodenaufrufe.

Derartige Größenordnungen sind nicht ungewöhnlich für aktuelle Java-Projekte. Das gilt nicht nur für technische Frameworks à la WildFly, sondern gerade auch für Anwendungen im Unternehmensumfeld. Die genannten Mengengerüste umfassen dabei noch nicht einmal einzelne Statements wie Operationen, Bedingungen oder Schleifen, sie liegen noch auf der Ebene struktureller Beziehungen zwischen Grundelementen der Sprache Java – also Packages, Klassen, Feldern und Methoden. Die Zahlen vermitteln jedoch einen Eindruck davon, dass allein aufgrund derartiger Mengengerüste offensichtlich auch eine Komplexität jenseits des eigentlichen Programmcodes existiert und in geeigneter Form beherrscht werden muss.

Die Sprache Java vs. Anwendungsstrukturen

An dieser Stelle betritt im Normalfall der Begriff der Anwendungsarchitektur das Spielfeld: Vereinfacht ausgedrückt geht es dabei um die Zerlegung eines komplexen Ganzen in eine Menge beherrschbarer Einheiten und die Definition der Beziehungen zwischen diesen. Es werden Begriffe wie Komponenten, Module, Schichten, Schnittstellen und Abhängigkeiten eingeführt.

Diese besitzen bei der Umsetzung in eine Quellcodestruktur ein gemeinsames Problem: Sie stellen Elemente dar, für die es in der Sprache Java keine unmittelbaren technischen Repräsentationen gibt. Die Folge ist, dass die Umsetzung einer Architektur immer über Umwege erfolgen muss.

Die schwächste Lösung im Sinne der Durchsetzung ist dabei die Dokumentation (z. B. UML), welche an die Entwickler kommuniziert wird – ein Ansatz, der selbst in kleinen Projekten in der Regel nur sehr eingeschränkt funktioniert, da Entwickler kein unmittelbares Feedback über mögliche Verletzungen erhalten. Etwas sicherer ist der Einsatz spezifischer Frameworks und/oder Codegenerierung. Der Preis hierfür ist jedoch unter anderem ein hoher Initial- und Wartungsaufwand sowie ein hohes Maß an Abstraktheit mit einhergehenden Einschränkungen in der Flexibilität.

Ein eher pragmatischer Weg verläuft über die gezielte Nutzung – oder eher einen Missbrauch – des vorhandenen Build-Systems (z. B. Maven). Alle fachlichen beziehungsweise technischen Schnitte (also Module, Schichten, APIs, Implementierungen usw.) werden als Module abgebildet und entsprechende Abhängigkeiten definiert. Unangenehme Seiteneffekte sind dabei unter anderem aber eine große Menge kleiner Module sowie nicht zu vernachlässigende Aufwände zum Ausblenden transitiver Abhängigkeiten.

Es gibt jedoch noch weitere strukturelle Aspekte, welche durch Java-Sprachelemente ebenfalls nicht abgebildet werden können. Sie liegen auf der Ebene zwischen konkreten, ausführbaren Statements und den durch die Architektur definierten Einheiten. Ein Beispiel im Stil einer Java EE-Anwendung soll dies veranschaulichen.

Die Package-Struktur der Beispielanwendung mit dem Root-Package `com.buschmais.demo` und den Modulen

```
...module1
...module1.ui
...module1.ejb
...module1.persistence
...module2
...module2.ui
...module2.ejb
...module2.persistence
```

stellt mögliche fachliche („module1“ und „module2“) und technische („ui“, „ejb“, „persistence“) Schnitte dar.

Erfahrungsgemäß entstehen in der Praxis noch weitere Schnitte unterhalb dieser Ebenen. Es handelt sich dabei um Packages, welche Klassen mit speziellen Eigenschaften oder Zusammengehörigkeiten aufnehmen, zum Beispiel UI-Models und -Controller, JPA-Entitäten, Separierungen zwischen APIs und Implementierungen usw. Den Packages werden damit Rollen zugewiesen, die der Feinstrukturierung einer Architektureinheit (Modul, Schicht usw.) dienen. Damit ergeben sich normalerweise auch Aussagen über erlaubte und verbotene Abhängigkeiten, das heißt, Klassen in einem Implementierungs-Package müssen das jeweilige API kennen, umgekehrt darf dies jedoch nicht der Fall sein.

Im Idealfall gleichen sich derartige Festlegungen zwischen verschiedenen Modulen, um Entwicklern die Orientierung im System zu erleichtern – im Idealfall wird also die Möglichkeit geschaffen, Erwartungshaltungen aufbauen und innerhalb eines Projektes gleichbleibend befriedigen zu können. Eine entsprechende Strukturierung wird hier für ein fachliches Modul demonstriert

```
...module1
...module1.ui
...module1.ui.model
...module1.ui.controller
...module1.ejb.api
...module1.ejb.impl
...module1.persistence.api
...module1.persistence.api.model
...module1.persistence.impl
```

```
1  package com.buschmais.demo.model;
2
3  import javax.persistence.Entity;
4
5  @Entity
6  public class Person {
7
8    public String getName() { ... }
9
10   ...
11 }
```

Abb. 1: Strukturen einer Java-Anwendung als Graph

Ähnliche Fragen tauchen normalerweise auch auf „tieferen“ technischen Ebenen auf, wie der von Klassen: Die Verwendung von Namenskonventionen ermöglicht es, auf den ersten Blick eine grobe Einschätzung ihrer Rolle im System zu treffen, ohne deren Quellcode öffnen, betrachten oder gar verstehen zu müssen. Ein gängiges Beispiel hierfür ist die Verwendung des Suffixes „MDB“ für Message-Driven Beans.

Die Urheberschaft derartiger Definitionen für die Verortung von Elementen und deren Namenskonventionen verbleibt in der Realität aber oftmals im Unklaren: Sie werden eher selten durch die Architektur definiert, sie entstehen vielmehr während der täglichen Entwicklungsarbeit und sind daher oft zufälliger Natur.

Wäre es nicht wünschenswert, an dieser Stelle durch ein Werkzeug unterstützt zu werden, welches in der Lage ist, einzelnen Elementen einer Anwendung individuell oder sogar anhand von Mustern Rollen zuzuweisen und darauf aufbauend die Möglichkeit eröffnet, Regeln zu formulieren und automatisiert zu überprüfen?

Software as a Graph

Die grundlegende Idee besteht nun darin, Strukturelemente und deren Beziehungen untereinander in einer Datenbank zu erfassen. Über Abfragen können dann Elemente identifiziert werden, deren Eigenschaften definierten Konventionen oder Constraints nicht entsprechen – solche Abfragen können als Regeln betrachtet werden.

An dieser Stelle spielt die Wahl einer passenden Datenbank und ihrer Eigenschaften eine entscheidende Rolle – ein relationaler Vertreter wäre sicherlich ungeeignet, wenn die folgenden Anforderungen in Betracht gezogen werden:

- effiziente Abbildung und Traversierung von Beziehungen (z. B. Vererbungshierarchien unbekannter Tiefe),
- flexible Möglichkeiten zur Anreicherung erfasster Rohdaten (z. B. Packages, Klassen, Methoden, Felder) um projektspezifische und technische Konzepte (z. B. Modul, Controller, Test, EJB, Datasource),
- eine Abfragesprache, die im Sinne der Verständlichkeit bereits formulierter und einfacher Erstellung neuer Regeln möglichst wenig technische Hilfskonstrukte benötigt (z. B. bei der Traversierung von Beziehungen).

Eine passende Lösung bietet sich mit der Verwendung der Graphdatenbank Neo4j an – das ihr zugrunde liegende Datenmodell ist ein sogenannter Property-Graph, welcher Knoten und Beziehungen zwischen diesen als native Elemente definiert, die wiederum mit Attributen (Properties) versehen werden können. Ein Package wird also durch einen Knoten dargestellt, der ein Attribut „fqn“ (fully qualified name) besitzt. Eine darin enthaltene Klasse ist wiederum ein Knoten mit einem Attribut „fqn“. Zwischen beiden existiert eine entsprechende „CONTAINS“-Beziehung, die gerichtet ist. Ein Beispiel mit weiteren Elementen ist in Abbildung 1 dargestellt.

Der Graph beinhaltet ein weiteres wichtiges Element des Neo4j-Datenmodells, welches mit Version 2.0 eingeführt wurde: Ein Knoten kann mit einer beliebigen Menge sogenannter Labels versehen werden. Eine Klasse besitzt beispielsweise die Labels „Type“ und „Class“, ein Interface würde mit „Type“ und „Interface“ versehen sein. Die Präsenz eines Labels an einem Knoten verleiht ihm eine spezifische Rolle und definiert damit sein Schema, das heißt, das Vorhandensein entsprechender Attribute (z. B. „fqn“) und Beziehungen (z. B. „CONTAINS“). Spielt nun ein Knoten eine spezielle Rolle innerhalb der zu betrachtenden Anwendung, kann im Rahmen einer Anreicherung des Datenmodells einfach ein entsprechendes Label hinzugefügt werden.

Ein Beispiel hierfür ist die in Listing 1 dargestellte Abfrage, die in der Sprache Cypher formuliert ist und Elemente im ge-

samten Graphen sucht, welche dem Muster einer Klasse entsprechen, die mit einer Annotation vom Typ „javax.persistence.Entity" versehen ist. Alle derartig gefundenen Klassen-Knoten erhalten die Labels „Jpa2" und „Entity".

```
match
  (e:Type:Class)-[:ANNOTATED_BY]->(a:Annotation),
  (a)-[:OF_TYPE]->(t:Type)
where
  t.fqn = "javax.persistence.Entity"
set
  e:Jpa2:Entity
return
  e as Jpa2Entit
```

Listing 1: Anreichern des Datenmodells

JPA-Entitäten werden damit zu einem eigenständigen Konzept innerhalb des Datenmodells. Basierend darauf lassen sich in einem nächsten Schritt Abfragen formulieren, welche die Einhaltung projektspezifischer Regeln sicherstellen. Listing 2 demonstriert einen Constraint, der sicherstellt, dass sich alle JPA-Entitäten in Packages befinden, deren Name „model" lautet.

```
match
  (p:Package)-[:CONTAINS]->(e:Jpa2:Entity)
where not
  p.name = "model"
return
  e as EntityInWrongPackage
```

Listing 2: Constraint zur Überprüfung von JPA-Entitäten

Beide Beispiele lassen sich gut auf die Denkweise eines Entwicklers in seiner täglichen Arbeit übertragen. Entdeckt er beim Öffnen des Quellcodes einer Klasse die Annotation **@Entity**, ist für ihn die Rolle dieser Klasse innerhalb der Anwendung klar – er wird sich allerdings stirnrunzelnd die Frage stellen, warum sie sich ausgerechnet in einem Package „controller" befindet.

Genauso sollte es sich prinzipiell auch mit den naturgemäß abstrakteren Architektur- und Designkonzepten verhalten – allerdings besteht im Alltag oft das Problem, dass sich Entwickler eben dieser Konzepte im aktuellen Arbeitskontext oftmals nicht bewusst sind und sie verletzen. Das beschriebene Vorgehen des Scannens von Rohdaten, deren Anreicherung um technische und projektspezifische Konzepte, die Überprüfung von Constraints und entsprechendes, vor allem aussagekräftiges Feedback an die Entwickler kann hier Abhilfe schaffen. Dieser Ansatz wird durch ein Open-Source-Werkzeug umgesetzt, auf das im Folgenden überblicksartig eingegangen werden soll.

jQAssistant – Konzepte und Constraints

Das Projekt jQAssistant [jQAssistant] fand seinen Ursprung, als der Autor dieses Beitrages – geprägt durch Erfahrungen in diversen Java EE-Projekten im Unternehmensumfeld – auf Neo4j aufmerksam gemacht wurde und spontan Gefallen an der intuitiven Art der Datenmodellierung und der Ausdrucksstärke von Cypher fand. Schnell entstanden ein Scanner für Java-Artefakte, ein kleines Framework zum Aufbau individueller und wiederverwendbarer Regelsätze sowie ein Maven-Plug-in, das beides kapselte.

Nach nicht allzu langer Zeit stellte sich heraus, dass auch Scanner für andere technische Konstrukte von Interesse sein könnten, zum Beispiel Maven-, CDI- und JPA-Deskriptoren oder auch JUnit-Testreports. Der Ansatz wurde daher auf ein Plug-in-orientiertes Konzept umgestellt, das somit beliebige Artefakte lesen, Regeln anwenden und Reports erzeugen kann.

Die Einbindung in den Maven-Build-Prozess erfolgt wie in Listing 3 angedeutet. Damit werden der Scan erzeugter Artefakte und die Analyse basierend auf definierten Regeln aktiviert.

```
<plugin>
  <groupId>com.buschmais.jqassistant.scm</groupId>
  <artifactId>jqassistant-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>scan</id>
      <goals>
        <goal>scan</goal>
      </goals>
    </execution>
    <execution>
      <id>analyze</id>
      <goals>
        <goal>analyze</goal>
      </goals>
    </execution>
  </executions>
  <dependencies> <!-- jQAssistant plugins --> </dependencies>
</plugin>
```

Listing 3: jQAssistant Maven Integration

Die Einbindung der jQAssistant-eigenen Plug-ins (z. B. für EJB-Unterstützung) erfolgt durch Dependency-Deklarationen am Maven-Plug-in. Die Definition eigener beziehungsweise Verwendung vordefinierter Regeln erfolgt über XML-Deskriptoren, welche in Abhängigkeit zueinander stehen können. Anhand des Datenmodells in Abbildung 2 und des dazugehörigen Beispiels in Listing 4 soll dies verdeutlicht werden.

Es ist eine Gruppe „default" definiert, diese wird – wenn in der Konfiguration des Maven-Plug-ins nicht anders angegeben ist – im Rahmen der Analyse ausgewertet. Sie referenziert eine Regel (genauer einen Constraint) namens „my-rules:ITClassName". Diese stellt sicher, dass alle Testklassen, welche einen Integrationstest darstel-

```
1  package com.buschmais.demo.test;
2
3  import org.junit.Test;
4
5  public class ExampleTest extends AbstractIT {
6
7    @Test
8    public void test() {
9        ...
10   }
11 }
```

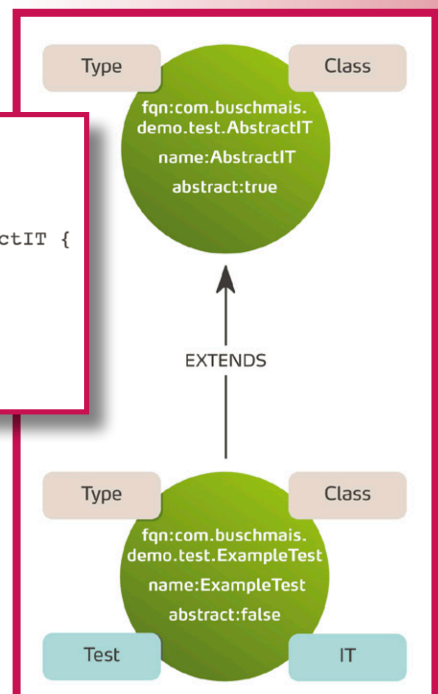

Abb. 2: Testklassen-Hierarchie als Graph

```
<jqa:jqassistant-rules xmlns:jqa=
"http://www.buschmais.com/jqassistant/core/analysis/rules/schema/v1.0">

  <concept id="my-rules:IT">
    <requiresConcept refId="junit4:TestClass" />
    <description>Labels all test classes deriving from
      "AbstractIT" as "IT".</description>
    <cypher><![CDATA[
      match
        (it:Class:Test)-[:EXTENDS*]->(itType:Type)
      where
        itType.fqn = "com.buschmais.demo.test.AbstractIT"
      set
        it:IT
      return
        it as IntegrationTest
    ]]></cypher>
  </concept>

  <constraint id="my-rules:ITClassName">
    <requiresConcept refId="my-rules:IT" />
    <description>All integration tests must have a name with suffix "IT" to
be only executed as part of the integration test build.</description>
    <cypher><![CDATA[
      match
        (it:Class:IT)
      where not
        it.name =~ ".*IT" // regular expression
      return
        it as InvalidIntegrationTest
    ]]></cypher>
  </constraint>

  <group id="default">
    <includeConstraint refId="my-rules:ITClassName" />
  </group>

</jqa:jqassistant-rules>
```

Listing 4: Definition von jQAssistant-Regeln

len, in ihrem Namen das Suffix „IT“ tragen. Die dahinterliegende Idee besteht darin, lang laufende Integrationstests in gesonderte Builds auslagern zu können (Stichwort: failsafe-maven-plugin). Die Regel verfügt über eine ausführliche Beschreibung für Entwickler, mit welcher der Maven-Build abgebrochen wird, wenn eine Verletzung vorliegt, das heißt, durch die Cypher-Abfrage mindestens eine Integrationstest-Klasse gefunden wurde, die nicht das korrekte Suffix verwendet. Die ausgegebene Fehlermeldung enthält selbstverständlich auch Informationen über die betreffende Klasse – der Entwickler erhält also genau das Feedback, welches er benötigt, um das Problem verstehen und den Fehler korrigieren zu können.

Bei Betrachtung der Cypher-Abfrage des Constraints fällt auf, dass Integrationstests in der Match-Klausel über das Label „IT“ identifiziert werden. Dieses wird durch das Konzept „my-rules:IT“ gesetzt und ist durch den Constraint als Vorbedingung referenziert (requiresConcept). jQAssistant sorgt automatisch für die richtige Reihenfolge der Ausführung der beiden Regeln.

Das Konzept selbst sucht nach Testklassen, welche direkt beziehungsweise indirekt von einer Klasse namens „com.buschmais.test.AbstractIT“ ableiten (EXTENDS*). Dafür benutzt es ein weiteres Label names „Test“, welches durch ein referenziertes Konzept namens „junit4:TestClass“ erzeugt wird. Letzteres ist nicht Teil des XML-Deskriptors, sondern wird im Sinne allgemeiner Wiederverwendbarkeit durch das JUnit4-Plug-in von jQAssistant bereitgestellt.

Das geschilderte Szenario verdeutlicht sehr gut die stufenweise, systematische Anreicherung eingelesener Strukturen durch allgemeine technische beziehungsweise projektspezifische Konzepte bis hin zu einer Ebene, auf der sich gut schreib- und lesbare Constraints formulieren lassen. Die beiden demonstrierten Anwendungsfälle – Überprüfung der Namenskonventionen von Klassen beziehungsweise deren Platzierung in Packages – stellen nur eine kleine Auswahl möglicher Regeln dar. Denkbar sind beispielsweise Anreicherungen des Datenmodells dahin gehend, dass Maven-Module oder Packages mit Labels versehen werden, die Architekturkonzepten (z. B. Module) entsprechen, und darauf aufbauend erlaubte beziehungsweise nicht erlaubte interne beziehungsweise externe Abhängigkeiten definiert werden.

Auf weniger abstrakten Ebenen lassen sich Constraints einführen, die den Aufruf unerwünschter Methoden aus dem JRE beziehungsweise eingesetzter Frameworks unterbinden (z. B. „keine assert-Methoden ohne explizite Message“) oder erzwingen (z. B. „jede Test-Methode muss direkt oder indirekt mindestens eine assert-Methode aufrufen“). Aufgrund der Tatsache, dass durch jQAssistant-Plug-ins auch Informationen über Deskriptoren technischer Frameworks zur Verfügung gestellt werden, sind weitergehende Prüfungen möglich. Ein Beispiel hierfür ist die Sicherstellung, dass JPA-Persistence-Deskriptoren (persistence.xml) nur in bestimmten Maven-Modulen erlaubt sind oder JNDI-Datasources verwenden müssen.

Fazit

Ein wichtiger Aspekt der Qualitätssicherung in Java-Projekten besteht in der Sicherstellung definierter Architekturkonzepte. Darüber hinaus kann die Definition konsistenter beziehungsweise verständlicher Strukturen die Zugänglichkeit einer Code-Basis für Entwickler erhöhen und damit Zeiten für die Implementierung von Änderungen auf lange Sicht verkürzen.

Da die Sprache Java naturgemäß selbst keine Mittel zur Absicherung von Architektur- und Designkonzepten oder Namenskonventionen zur Verfügung stellt, muss dies durch entsprechende Analyse-Werkzeuge übernommen werden. Es wurde anhand des Open-Source-Projektes „jQAssistant“ ein Ansatz vorgestellt, der basierend auf der Graphdatenbank Neo4j das automatisierte Einlesen von Softwarestrukturen, die Anreicherung der gewonnenen Daten um allgemeine technische beziehungsweise projektspezifische Konzepte sowie die Überprüfung auf Regelverletzungen ermöglicht. Das Vorgehen zeichnet sich durch hohe Flexibilität und Anpassbarkeit auf individuelle Problemstellungen aus und schließt eine Lücke im derzeit zur Verfügung stehenden Werkzeugsatz für Qualitätssicherung.

Links

[jQAssistant] Quality Assurance using jQAssistant,
`http://jqassistant.org`
[Neo4j] `http://neo4j.org`

Dirk Mahler ist Senior-Consultant bei der buschmais GbR, einem Beratungshaus mit Sitz in Dresden. Der Schwerpunkt seiner mittlerweile mehr als 10-jährigen Tätigkeit liegt im Bereich Architektur für Java-Applikationen im Unternehmensumfeld. Den Fokus setzt er dabei auf die Umsetzung von Lösungen, die im Spannungsfeld zwischen Pragmatismus, Innovation und Nachhaltigkeit liegen. In diesem Rahmen engagiert er sich für die Open-Source-Projekte jQAssistant und eXtended Objects.
E-Mail: dirk.mahler@buschmais.com