JAVA Mag
Javamagazin™
Java | Architektur | Software-Innovation
Sonderausgabe

Ein Tool, viele Möglichkeiten

Eine Einführung in jQAssistant – Teil 1

Ein Tool, viele Möglichkeiten

„jQAssistant? Schon wieder ein neues Tool?“, könnte man fragen. Was für Viele vielleicht noch unbekannt ist, jedoch schon in vielen Projekten erfolgreich eingesetzt wird, ist ein sehr flexibles Werkzeug, mit dem Wissen über Softwaresysteme unkompliziert aufgebaut und dokumentiert werden kann. Architektur und Umsetzung lassen sich einander so wieder näherbringen. In unserer dreiteiligen Artikelserie (Kasten: „Über diese Artikelserie“) gibt Stephan Pirnbaum einen Einstieg in die Verwendung von jQAssistant und zeigt interessante Use Cases.

von Stephan Pirnbaum

Werkzeuge, die Entwicklerteams bei der täglichen Arbeit unterstützen, gibt es heutzutage wie Sand am Meer: Angefangen bei IDEs mit Code Completion und Debuggern über Tools zur Qualitätssicherung bis hin zu Mitteln zur kollaborativen Arbeit. Da dürften eigentlich keine Wünsche mehr offenbleiben, oder? Nicht ganz!

Artikelserie

Teil 1: Ein Tool, viele Möglichkeiten
Teil 2: Software-Analytics
Teil 3: Architekturdokumentation und -validierung

Je größer die Abstraktion weg vom Code ist, desto schwieriger wird es, unterstützende Tools zu finden. Ein paar Beispiele? Sprachkonzepte können heute einfach mit jeder IDE identifiziert und analysiert werden. Für Architekturkonzepte ist das leider schwierig und aufwendig. Auch der Abgleich von Sollarchitektur und der Implementierung ist nicht so einfach wie die Überprüfung von Coding-Standards. Ebenso ist die Dokumentation von Implementierungsdetails, Stichwort Javadoc, häufig fest in den Entwicklungs-Flow integriert. Bei der Architekturdokumentation schaut das schon ganz anders aus.

Ein bunter Strauß an Möglichkeiten

Das Open-Source-Tool jQAssistant schließt die Lücke zwischen Architektur und Implementierung und stellt dafür einen variablen Baukasten mit flexiblen Einsatz-

möglichkeiten bereit. Das Ziel hinter jQAssistant ist es, die Komplexität von Softwaresystemen für den Menschen greifbar zu machen, indem es von Implementierungsdetails abstrahiert und Strukturinformationen hervorhebt. Dabei beschränkt sich jQAssistant nicht auf die Sprachbestandteile von Java, sondern ermöglicht es, auch Architekturmodelle, Informationen über das Build-System oder die Entwicklungshistorie zugänglich zu machen und mit dem Code zu verknüpfen. Durch diese Abstraktion auf das Relevante ist es Entwicklern, Architekten und Entscheidern möglich, den Fokus auf ihr konkretes Problem zu legen. Was diese Abstraktion genau ist, sehen wir im nächsten Abschnitt. Zu den Use Cases gehören unter anderem:

- Software-Analytics
  - Beantwortung alltäglicher Entwicklerfragen, beispielsweise Impact-Analysen für Änderungen
  - Identifikation potenziell problematischer Codestellen, beispielsweise häufig geänderter Code
  - Planung von Refaktorisierungs- und Modernisierungsprojekten
  - Aufwands- und Risikoabschätzung für bevorstehende Änderungen
- Bewertung von Softwaresystemen
  - Automatisierte Erhebung von Metriken und Aggregation zur Systemzustandsbewertung
- Dokumentation und Abgleich von Architektur und Implementierung
  - Dokumentation von Architekturkonzepten und deren Identifikation im Source Code
  - Definition von Sollarchitekturen und Abgleich mit dem Istzustand zur Vermeidung von Architekturerosion
  - Generierung von Diagrammen auf Basis der tatsächlichen Implementierung, z. B. Klassendiagramme, Komponentendiagramme, Context Maps oder C4-Diagramme

Typische Use Cases sind zusätzlich in **Abbildung 1** dargestellt.

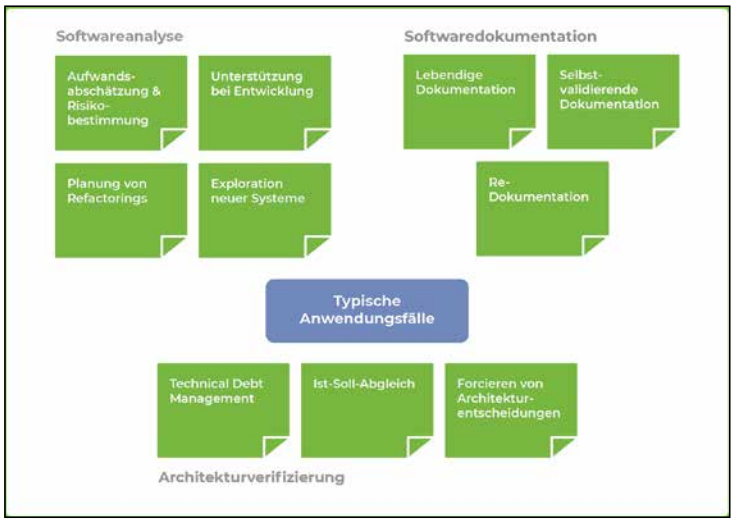


Abb. 1: Use Cases von jQAssistant

Die Funktionsweise

Um die skizzierten Use Cases zu ermöglichen, liest jQAssistant Strukturinformationen von Softwaresystemen ein. Dieser Vorgang wird als Scan bezeichnet. Dabei können unterschiedliche Quellenformate eingelesen werden. Dazu gehören neben Javas Class-Dateien (Bytecode) auch Informationen über Build- und Abhängigkeitsinformationen in Form von *pom.xml*-Dateien. Je nach Konfiguration können zusätzlich auch die Git-Historie, C4- und Context-Mapper-Modelle, Enterprise-Architect-Dateien und vieles mehr gescannt werden. All diese Informationen landen in einer Neo4j-Graphdatenbank, die standardmäßig als Embedded-Version in jQAssistant enthalten ist. Die Verwendung einer Neo4j-DB stellt einen großen Vorteil dar, da dadurch zur Abfrage der Strukturen auch die dazugehörige Abfragesprache Cypher verwendet werden kann. So steht die gesamte offizielle Neo4j-Dokumentation als Grundlage zur Verfügung. Es kann leicht Unterstützung im Internet gefunden und etwaige Erfahrungen in der Verwendung der Datenbank können aus dem Projektalltag direkt übertragen werden. Das ist natürlich auch in die andere Richtung interessant. Dank jQAssistant können erste Erfahrungen mit Neo4j gemacht werden, die vielleicht später in einem Projekt vorteilhaft sind.

Auf den gescannten Informationen werden anschließend Abfragen in der Abfragesprache Cypher ausgeführt, um beispielsweise herauszufinden, welche Codebestandteile besonders häufig verändert werden. Diese Analysen können entweder manuell in der bereitstehenden Oberfläche oder automatisiert bei der Ausführung von jQAssistant durchgeführt werden. Da-

Über diese Artikelserie

Dieser Artikel ist der erste Teil einer dreiteiligen Artikelserie. Er soll die Funktionsweise und die Integration von jQAssistant in das eigene Projekt erklären. Dabei sollen exemplarisch die ersten Schritte zur Exploration von Softwaresystemen aufgezeigt werden, um jQAssistant im eigenen Projekt erfolgreich einsetzen zu können. Im zweiten Teil dreht sich alles um Software-Analytics. Es soll auf Best Practices eingegangen werden, um Software-Analytics bestmöglich einsetzen zu können. Daneben werden mögliche Fragestellungen und deren Lösung mit Software-Analytics besprochen und es wird gezeigt, welche zentrale Rolle jQAssistant dabei spielt und wie Ergebnisse interaktiv für die verschiedensten Zielgruppen aufgearbeitet werden können. Im dritten und letzten Teil soll schließlich gezeigt werden, welche Mittel und Wege jQAssistant bietet, um Architektur und Implementierung wieder zusammenzuführen. Dabei soll auch darauf eingegangen werden, wie Architekturdokumentation richtig, nachhaltig und dennoch mit wenig Aufwand umgesetzt werden kann, um schlussendlich einen Mehrwert für das gesamte Team zu liefern.

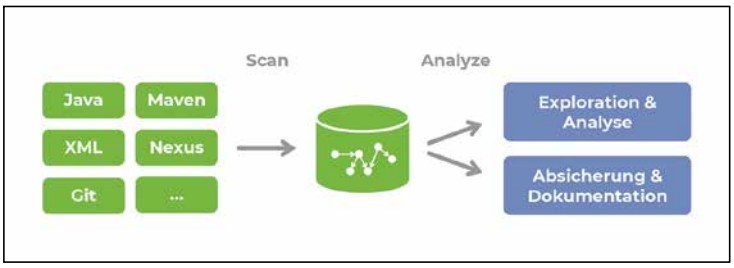


Abb. 2: Die Funktionsweise von jQAssistant

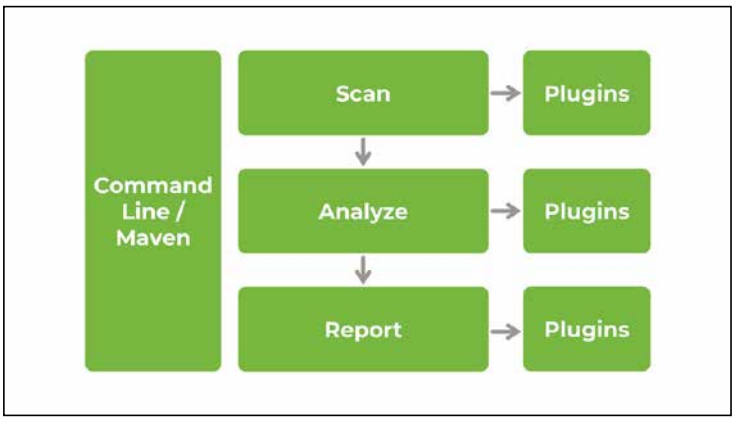


Abb. 3: Modularer Aufbau von jQAssistant

neben ist es auch möglich, die eingelesenen Strukturen gegen dokumentierte Sollarchitekturen abzugleichen oder die tatsächliche Implementierung zur Generierung von Dokumentationsartefakten, beispielsweise Diagrammen, zu nutzen. **Abbildung 2** zeigt den groben Ablauf.

jQAssistant ist modular aufgebaut und kann durch Plug-ins erweitert werden. Damit sind der Datenbasis für die spätere Analyse nahezu keine Grenzen gesetzt (**Abb. 3**). So ist die Funktionalität, um Java-Code scannen zu können, selbst ein Scan-Plug-in. Das Gleiche gilt für alle anderen Dateiformate, die aktuell von Haus aus unterstützt und über den offiziellen Contrib-Bereich [1] oder sonstige Quellen durch die Community bereitgestellt werden. Neben den in jQAssistant standardmäßig

Listing 1

```
<build> <plugins> <plugin>
  <groupId>com.buschmais.jqassistant</groupId>
  <artifactId>jqassistant-maven-plugin</artifactId>
  <version>1.11.1</version>
  <executions>
    <execution>
      <id>default-cli</id>
      <goals>
        <goal>scan</goal>
      </goals>
    </execution>
  </executions>
</plugin> </plugins> </build>
```

enthaltenen Plug-ins werden häufig folgende Scan-Plug-ins ergänzt:

- *de.kontext-e.jqassistant.plugin:jqassistant.plugin.git* [2]: ein Scan-Plug-in, um Git-Historien einlesen zu können
- *org.jqassistant.contrib.plugin:jqassistant-context-mapper-plugin* [3]:ein Scan-Plug-in, um DDD Context Maps einlesen zu können
- *org.jqassistant.contrib.plugin:jqassistant-c4-plug-in* [4]: ein Scan-Plug-in, um Architekturdiagramme des C4-Modells einlesen zu können

Falls doch mal etwas fehlen sollte, ist es einfach möglich, ein eigenes Scan-Plug-in zu entwickeln [5]. Dieser Plug-in-Ansatz setzt sich ebenfalls in der Analyse fort, beispielsweise für die Prüfung von Architekturverstößen. Regeln, die einmal definiert wurden, können so in verschiedenen Projekten wiederverwendet werden. Ein Beispiel stellt hier das jqassistant-jmolecules-plugin [6] dar, das Prüfungen für Architekturkonzepte wie Domain-driven Design oder Schichtenarchitekturen bereitstellt. Aber auch unternehmensspezifische Regeln, z. B. Standardisierungen in Microservices-Landschaften, können auf diese Art wiederverwendbar umgesetzt werden. Im Bereich des Reportings stehen ebenso Plug-ins bereit, beispielsweise für AsciiDoc oder PlantUML.

Abbildung 3 zeigt auch, wie jQAssistant verwendet werden kann. Die häufigste Möglichkeit ist, jQAssistant direkt als Maven-Build-Plug-in einzusetzen. Sollte Maven nicht eingesetzt werden, existiert eine Command-Line-Distribution, die unabhängig vom Build-System verwendet werden kann. Die Ausführung von jQAssistant innerhalb von Maven unterteilt sich in einzelne Goals, u. a.:

- *jqassistant:scan:* Einlesen von verschiedenen Dateiformaten und Aufbau der Datenstrukturen in Neo4j
- *jqassistant:analyze:* Ausführen von Regeln zur Prüfung der Strukturen gegen Entwicklungs- und Architekturregeln
- *jqassistant:report:* Erstellen von zusätzlichen Reports, beispielsweise einer HTML-Übersicht ausgeführter Regeln inklusive Regelverletzungen
- *jqassistant:server:* Starten der eingebetteten Neo4j-Datenbank, um manuell Abfragen gegen diese ausführen und die Strukturen explorieren zu können

Los geht's: jQAssistant in das eigene Projekt integrieren

Bis hierhin wurde erläutert, welche Use Cases mit jQAssistant umsetzbar sind und wie jQAssistant intern aufgebaut ist. Um mit einem der zuvor genannten Use Cases starten zu können, ist es immer notwendig, jQAssistant in das eigene Projekt zu integrieren, schließlich müssen die Daten zunächst ihren Weg in die Datenbank finden. Für diesen ersten Teil soll die Integration von jQAssistant anhand des in [7] bereitgestellten, Java- und Ma-

ven-basierten Beispielprojekts erläutert werden. Dieses Beispielprojekt hat die Besonderheit, dass es intern in einzelne Packages zerfällt, die fachliche Teilbereiche darstellen. Um erstes Wissen über das System aufzubauen, soll herausgefunden werden, wie diese Teilbereiche miteinander interagieren und ob es gegebenenfalls sogar zyklische Abhängigkeiten, ein klares Anti-Pattern und einen guten Indikator für eine falsche Strukturierung gibt. Dazu soll natürlich jQAssistant verwendet werden.

Zur Integration in ein Projekt ist die Nutzung des Maven-Build-Plug-ins zu empfehlen. Dadurch bindet sich jQAssistant direkt in den Build-Prozess ein, kann abhängig von Maven-Profilen (de-)aktiviert werden sowie umgebungsspezifische Schritte und Prüfungen ausführen. Die in Listing 1 dargestellte Integration basiert auf dem Beispielprojekt, enthält aber nur die notwendige Minimalkonfiguration zur Beantwortung der im Rahmen dieses Artikels gestellten Frage.

Neben der Spezifikation der Maven-Koordinaten ist auch das jQAssistant Goal *scan* definiert, das, wie der Name vermuten lässt, zum Scannen des Projekts führt. Mit diesem Set-up kann jQAssistant nun während des Maven-Builds ausgeführt und anschließend das System exploriert werden, um die eingangs gestellte Frage zu fachlichen Zusammenhängen zu beantworten.

jQAssistant läuft standardmäßig in der Verify-Phase von Maven mit. Ein einfacher Aufruf von *mvn verify* würde entsprechend das spezifizierte jQAssistant Goal *scan* ausführen. Möchte man einzelne Goals abseits von *mvn verify* ausführen, wäre das auch direkt über *mvn jqassistant:scan* möglich.

Während des *scan* Goals wurden die Quellen des Projekts, u. a. Java-Klassen und Maven-*pom.xml*-Dateien, in die Datenbank eingelesen und das der Neo4j zugrunde liegende Property-Graph-Modell aufgebaut. Um herauszufinden, wie die fachlichen Teilbereiche (also Packages) des Beispielprojekts zusammenhängen, ist es hilfreich, sich die aufgebauten Strukturen anzuschauen. Nehmen wir als Beispiel die nachfolgend dargestellte Java-Klasse aus dem Beispielprojekt:

```
package com.buschmais.gymmanagement.user;
public class UserService { }
```

Mit dem Scan dieser Klasse entsteht das in **Abbildung 4** dargestellte Modell. Für jedes Element wird ein Knoten in der Datenbank angelegt, hier also ein Knoten für die Klasse und ein Knoten für das Package, in dem sich diese Klasse befindet. Diese Knoten werden durch Labels beschrieben, wobei jeder Knoten auch mehrere Labels aufweisen kann. An jedem dieser Knoten können zusätzlich Properties in Form von Key-Value-Paaren hängen. In diesem Beispiel sind das der Name der Klasse, ihre Sichtbarkeit und auch der Name des Packages. Als drittes Element existieren noch Beziehungen zwischen den Knoten, beispielsweise eine *:CONTAINS*-Beziehung zwischen Package-Knoten und Type-Knoten, an der wiederum auch Properties hängen können.

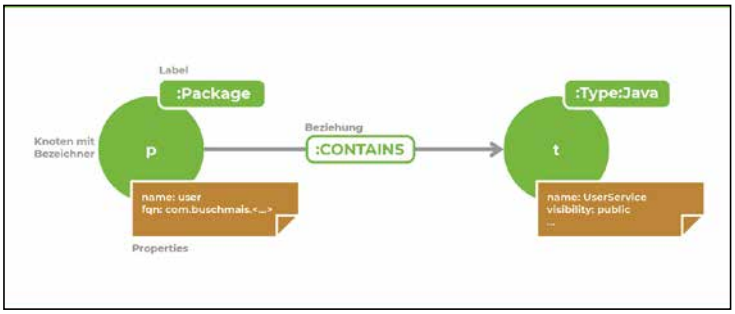


Abb. 4: Das Property-Graph-Modell

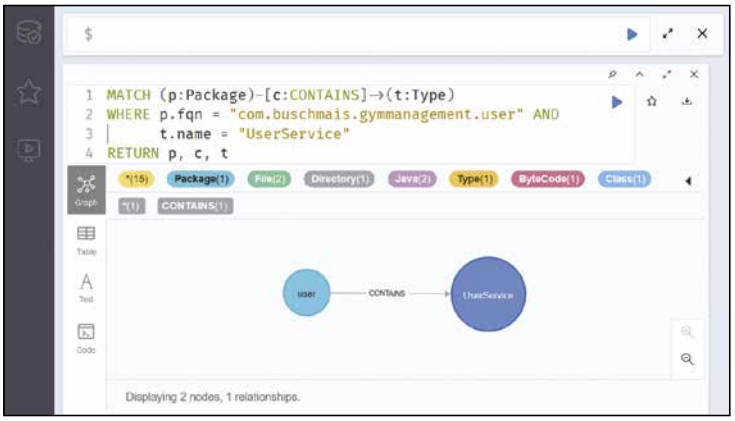


Abb. 5: Der Neo4j-Browser

Entdecke dein System

Um nun herauszufinden, wie die fachlichen Packages zusammenhängen und ob Zyklen zwischen ihnen existieren, werden zwei Dinge benötigt. Zum einen natürlich eine passende Abfrage und zum anderen eine Möglichkeit, diese auch auszuführen. Letzteres ist nutzerfreundlich über die Weboberfläche der Neo4j-Datenbank möglich. Dazu muss auf der Kommandozeile die von jQAssistant ausgelieferte Embedded-Version via *mvn jqassistant:server* gestartet werden. Anschließend kann die Oberfläche im Browser unter *http://localhost:7474* erreicht werden.

Nach dem Öffnen der Seite können in dem oberen Textfeld Abfragen eingegeben und über den blauen Pfeil oder über STRG + ENTER ausgeführt werden. Anschließend erscheint wie in **Abbildung 5** dargestellt das Ergebnis der Abfrage. Hier ist eine Cypher-Abfrage zu sehen, die die Struktur aus **Abbildung 4** abfragt. Diese besteht primär aus drei Teilen:

- MATCH*:* sucht nach dem spezifizierten Graph-Pattern
- WHERE*:* grenzt die Suchergebnisse auf die jeweils spezifizierten Bedingungen ein
- RETURN*:* gibt die Ergebnisse zurück

Anzumerken ist hier die beabsichtigte Ähnlichkeit des Match-Patterns zu ASCII-Art. Knoten bilden sich in der Abfrage mit *(bezeichner:Label)* ab, während sich Beziehungen zwischen diesen mit *-[bezeichner:RelationType]->* umsetzen lassen.

Ausgehend von dieser Query kann nun auch die Query zum Abfragen der Package-Abhängigkeiten umgesetzt werden. Dafür ist es notwendig, sich zu überlegen, wie Abhängigkeiten zwischen Packages gefunden werden können. Zwischen Package-Knoten existieren standardmäßig keine Beziehungen, die derartige Rückschlüsse zulassen. Jedoch sind auf Type-Knoten-Ebene diese Informationen in Form von *:DEPENDS_ON*-Beziehungen vorhanden. Damit kann wie nachfolgend dargestellt eben diese Information gefunden werden:

```
MATCH  (p1:Package)-[:CONTAINS]->(t1:Type),
       (p2:Package)-[:CONTAINS]->(t2:Type),
       (t1)-[:DEPENDS_ON]->(t2)
RETURN DISTINCT p1.fqn AS Source, p2.fqn AS Target
```

Nun ist diese Abfrage bereits relativ umfangreich. Es wäre schön, wenn beispielsweise die Abhängigkeiten zwischen Packages direkt in der Datenbank verfügbar wären. Listing 2 bewerkstelligt eben dies. Anzumerken sind hier zwei Aspekte:

1. Es wird eine Einschränkung der Package-Namen auf das Basis-Package vorgenommen, da es die erste für die Analyse interessante Ebene darstellt.
2. Es findet der Befehl *MERGE* Verwendung. Er fügt einen Knoten oder eine Beziehung hinzu, wenn diese(r) noch nicht existiert.

Nun ist es leicht, Zyklen zwischen Packages zu identifizieren. Alles, was dafür benötigt wird, ist nachfolgend dargestellt.

```
MATCH  (p1:Package)-[d1:DEPENDS_ON]->(p2:Package),
       (p2)-[d2:DEPENDS_ON]->(p1)
RETURN *
```

Das Ergebnis dieser Abfrage liefert für unser Beispielsystem glücklicherweise keine Ergebnisse. Wir haben also keine zyklischen Abhängigkeiten zwischen fachlichen Aspekten.

Natürlich sind noch viel mehr Möglichkeiten durch Cypher gegeben, die aber den Rahmen dieses Artikels sprengen würden. In den folgenden Artikeln dieser Serie wird das Schreiben von Abfragen jedoch immer wieder aufgegriffen, um einen tieferen Einblick zu gewähren. In der Zwischenzeit sei das Cypher Cheat Sheet [8] empfohlen.

Listing 2

```
MATCH  (p1:Package)-[:CONTAINS]->(t1:Type),
       (p2:Package)-[:CONTAINS]->(t2:Type),
       (t1)-[:DEPENDS_ON]->(t2)
WHERE  p1.fqn STARTS WITH "com.buschmais.gymmanagement" AND
       p2.fqn STARTS WITH "com.buschmais.gymmanagement"
MERGE  (p1)-[d:DEPENDS_ON]->(p2)
RETURN p1, d, p2
```

Fazit und Ausblick

Wenn es darum geht, Softwaresysteme zu analysieren und zu bewerten, Wissen darüber (wieder) zu gewinnen und zu dokumentieren sowie Architektur und Implementierung einander näherzubringen, führt kein Weg an jQAssistant vorbei. Zugegebenermaßen mag die Einstiegshürde zu Beginn hoch sein, da man im Zweifel eine neue Abfragesprache lernen muss. Allerdings können in der Graphdatenbank beliebige Daten aggregiert werden, was unzählige Strukturanalysen ermöglicht. Das liegt auch an dem flexiblen, plug-in-basierten Ansatz, der es einem gestattet, selbst neue Funktionalitäten zu entwickeln oder neue Programmiersprachen zu unterstützen.

Dieser Artikel ist nur ein kurzer Einstieg, der das Potenzial von jQAssistant verdeutlichen und die Einbindung in Maven-basierte Projekte erklären soll. In den nächsten beiden Artikeln dieser Serie soll dieses Wissen aufgegriffen werden, um zum einen in das Thema Software-Analytics und zum anderen in das Thema Architekturdokumentation und -validierung einzutauchen. In der Zwischenzeit kann alles Gelernte im Beispielprojekt nachvollzogen oder direkt im eigenen Projekt ausprobiert werden. Ich bin gespannt, wie ihr es findet und freue mich auf Feedback!

Stephan Pirnbaum ist Consultant bei der BUSCHMAIS GbR. Er beschäftigt sich leidenschaftlich gern mit der Analyse und strukturellen Verbesserung von Softwaresystemen im Java-Umfeld. In Vorträgen und Workshops präsentiert er seine gesammelten Erfahrungen und genutzten Methodiken.

stephan.pirnbaum@buschmais.com

Links & Literatur

[1] https://github.com/jqassistant-contrib

[2] https://github.com/kontext-e/jqassistant-plugins/blob/master/git/src/main/asciidoc/git.adoc

[3] https://github.com/jqassistant-contrib/jqassistant-context-mapper-plugin

[4] https://github.com/jqassistant-contrib/jqassistant-c4-plugin

[5] https://101.jqassistant.org/implementation-of-a-scanner-plugin/index.html

[6] https://github.com/jqassistant-contrib/jqassistant-jmolecules-plugin

[7] https://github.com/buschmais/The-Perfect-Greenfield

[8] https://neo4j.com/docs/cypher-refcard/3.5/

Eine Einführung in jQAssistant – Teil 2

Versteh dein System mit Software Analytics

Wie viel wissen wir eigentlich über unsere Softwaresysteme? Nun, meistens sind sie groß, vereinen eine Vielzahl fachlicher und technischer Anforderungen und, so unser Bauchgefühl, sind schwieriger weiterzuentwickeln und zu warten als notwendig. Wäre es nicht schön, wenn man das Bauchgefühl durch quantitative Analysen ersetzen könnte, um in Zukunft Aufwände und Risiken von Änderungen besser einschätzen und Problemstellen in Systemen frühzeitig identifizieren zu können? Software Analytics bietet eine Lösung.

von Stephan Pirnbaum

Wir alle kennen die Probleme, die bei der Arbeit an komplexen Softwaresystemen entstehen: Die Entwicklung gestaltet sich immer schwerfälliger und Aufwände sind zunehmend weniger abschätzbar. Obwohl wir immer mehr Zeit mit der Entwicklung verbringen, wird das System, insbesondere in großen Teams, uns immer unbekannter und es wird von Tag zu Tag wahrscheinlicher, über Ecken zu stolpern, die wir noch nie gesehen haben. Dadurch wird das Projekt einerseits schwerer zu planen, andererseits steigen auch die Fehleranfälligkeit und der Frust im Team. Wenn dann auch noch erfahrene Entwickler das Team verlassen, bleibt nur noch der Blick in die IDE, um herauszufinden, was sich im System verbirgt.

Und spätestens hier wird es kompliziert. Denn so mächtig moderne IDEs zum Nachvollziehen von Implementierungsdetails auch sind, so schwierig ist es, für höhere Abstraktionsebenen und komplexere Fragestellungen Antworten zu finden. Solche Fragen können die technische Umsetzung, die fachliche Architektur oder auch die Team- und Organisationsebene betreffen. Jeweilige Beispiele wären:

- Welchen Impact auf das System hat eine geplante Technologiemigration und welche fachlichen sowie technischen Komponenten sind betroffen?
- Existieren zyklische Zusammenhänge zwischen den fachlichen Teilbereichen des Systems?
- Wo liegen Blindspots in der Software, also Stellen, an denen bis jetzt kein Entwickler des aktuellen Teams je gearbeitet hat?

Wir wollen uns in diesem Beitrag mit Software Analytics einer Methode nähern, die es erlaubt, derartige Fragen konkret zu beantworten, und es so ermöglicht, zielgerichtet die nächsten Schritte zu definieren.

Was ist Software Analytics?

In aller Kürze lässt sich Software Analytics als Analyse der Daten von Softwaresystemen beschreiben. Software Analytics dient Entwicklern, Architekten und Entscheidern beim Verstehen und Bewerten von Softwaresystemen sowie der Identifikation von Problemstellen. Damit wird es möglich, Wissen über das System aufzubauen bzw. wiederzugewinnen und im Team zu teilen sowie es zum Treffen besserer Entscheidungen und zur strategisch sinnvollen Planung der Projekttätigkeiten zu nutzen. Unterm Strich ist das Ziel, das Bauchgefühl durch Wissen zu ersetzen und damit die Software kurz-, mittel- und langfristig zu verbessern. Der Einsatzbereich erstreckt sich von Fragen des alltäglichen Entwicklungs-

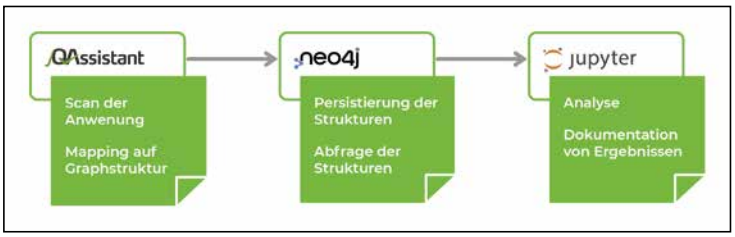


Abb. 1: Die Software-Analytics-Toolchain

geschäfts über die Planung und Abschätzung der demnächst anstehenden Themen bis hin zur Identifikation geschäftskritischer Probleme. Das Besondere ist dabei, dass durch die Abstraktion weg vom Source Code hin zu Daten nicht nur Entwickler profitieren, sondern bei geschickter Aufbereitung auch Entscheider bei ihrer täglichen Arbeit unterstützt werden.

Software Analytics besteht bei näherer Betrachtung aus vier Bestandteilen:

- *Gather* ist das Sammeln von Datenquellen und Daten, die zur Beantwortung der gestellten Frage oder zur Lösung des definierten Problems nützlich sind. Im Kontext von Software Analytics sind das beispielsweise der Byte- oder Source Code, Informationen über das Build- und Dependency-Management sowie die Entwicklungshistorie in Form von Git-Repositories.
- *Measure* ist die Messung des Systems unter Verwendung der zuvor definierten Datenquellen. Dazu zählen zunächst die Identifikation potenziell hilfreicher Messungen und die Definition, ob und wie diese anhand der vorliegenden Daten erhoben werden können. Anschließend erfolgen die statistische Aufbereitung relevanter Aspekte und die Erhebung von Metriken.
- *Analyze* ist die Auswertung der zuvor erhobenen Messwerte, um Rückschlüsse auf Ursachen des definierten Problems ziehen oder eine Antwort auf die Fragestellung geben zu können. Hier stellt sich schließlich heraus, ob der Lösungsweg zielführend war oder ob Anpassungen an der Analyse vorgenommen werden müssen.
- *Visualize* ist die Aufbereitung relevanter Daten in einer geeigneten Form, beispielsweise grafisch oder tabellarisch. Dieser Schritt wird umso bedeutender, je stärker die Ergebnisse an andere Entwickler und insbesondere an Architekten oder Entscheider kommuniziert werden müssen. Es muss auf den ersten Blick deutlich werden, ob und wo Probleme existieren und wie gravierend sie sind. Gleichzeitig ist es hilfreich, Möglichkeiten zum Drill-down in Details der Ergebnisse zu geben, um einen Mehrwert für Entwickler zu bieten. Zu wissen, an welche Zielgruppe wir uns richten, ist hier also entscheidend.

Über diese Artikelserie

Dieser Artikel ist der zweite Teil einer dreiteiligen Artikelserie. Er stellt Software Analytics als Methode zum Explorieren und Analysieren von Softwaresystemen einschließlich eines in der Praxis bewährten Set-ups vor. Ebenfalls soll auf mögliche Fragestellungen, deren optimale Beantwortung und auf Best Practices der Software Analytics eingegangen werden. Damit knüpft der zweite Teil nahtlos an den ersten Teil an, in welchem Funktionsweise und Integration von jQAssistant in das eigene Projekt erklärt wurden. Im dritten und letzten Teil in der nächsten Ausgabe soll schließlich gezeigt werden, welche Mittel und Wege jQAssistant bietet, um Architektur und Implementierung wieder zusammenzuführen. Dabei soll auch darauf eingegangen werden, wie Architekturdokumentation richtig, nachhaltig und dennoch mit wenig Aufwand umgesetzt werden kann, um schlussendlich einen nachhaltigen Mehrwert für das gesamte Projekt zu liefern.

Eine Analyse ist dann erfolgreich, wenn die Ergebnisse aufschlussreich für die Zielgruppe(n) sind und einen Mehrwert erzeugen. Zusätzlich müssen es die Ergebnisse auch ermöglichen, klare Handlungsschritte abzuleiten und mit der dokumentierten Analyse umsetzen zu können. Ein valider Ausgang zeigt auch, ob zusätzliche weiterführende Analysen durchgeführt werden müssen, die Teilaspekte der initialen Analyse beleuchten sollen.

Mach's richtig!

Um unser Softwaresystem erfolgreich analysieren zu können, benötigen wir natürlich die passenden Tools. Als Datenlieferant dient uns jQAssistant, das bereits in der letzten Ausgabe [1] vorgestellt wurde. Mit jQAssistant können wir die Strukturen von Softwaresystemen scannen und in einer Neo4j-Graphdatenbank für spätere Abfragen bereitstellen. Der Softwaregraph beinhaltet dann Repräsentationen z. B. für Java-Klassen, Packages und Maven-Module oder auch Git Commits bzw. Committer und erlaubt, diese sowie ihre Beziehungen untereinander abzufragen.

Ein wichtiger Aspekt bei Software Analytics ist die Dokumentation der Analyse und der Ergebnisse, sodass die Analyse auch für Dritte reproduzierbar und nachvollziehbar ist. Da der mitgelieferte Neo4j-Browser nur die Ausführung von Queries erlaubt und daher eher für kleinere On-Demand-Analysen (Exploration) geeignet ist, setzt die hier vorgestellte Software-Analytics-Toolchain auf Jupyter Notebook [2]. Die gesamte Toolchain ist in **Abbildung 1** dargestellt.

Mit Jupyter Notebook können in einer interaktiven Weboberfläche sogenannte Notebooks erstellt werden, in denen strukturierte Analysen mitsamt Text, Code und Visualisierungen durchgeführt werden können. Dafür bietet Jupyter einen Python-Kernel, der es ermöglicht, Analysen mit Python zu implementieren. Da für Python eine Vielzahl von Bibliotheken zur Datenauswertung und -visualisierung existiert, sind Jupyter Notebooks im Bereich Software Analytics ein sehr mächtiges Werkzeug. Die Notebooks können anschließend in verschiedenen Formaten, u. a. HTML, exportiert werden, um so

die Ergebnisse einem breiteren Kreis von Interessenten zur Verfügung zu stellen. Auch wenn es so erscheinen mag, sind keine tiefen Python-Kenntnisse notwendig, um schnell zu aussagekräftigen Resultaten zu kommen. Es sind häufig nur wenige Zeilen Code, die dank guter Dokumentation der Bibliotheken auch für Python-Neulinge leicht umsetzbar sind. An dieser Stelle noch zwei Empfehlungen für hilfreiche Bibliotheken:

- *Pandas* [3]: eine umfangreiche Bibliothek zur Datenanalyse, die Unterstützung für zahlreiche Dateiformate und Datenstrukturen bietet und die Aspekte Datenbereinigung, -aufbereitung, -transformation und -auswertung bedient.
- *Pygal* [4]: eine Visualisierungsbibliothek mit Standardvisualisierungen, wie z. B. Balkendiagramm, Kuchendiagramm und XY-Diagramm.

Neben den technischen Vorbedingungen ist erfahrungsgemäß auch die inhaltliche Strukturierung der Analyse von großer Bedeutung. Schließlich nützt uns Software Analytics nichts, wenn wir mit den Ergebnissen aufgrund einer schlechten Aufbereitung nichts anfangen können. Dafür eignet sich das Software Analytics Canvas [5] sehr gut, denn es betrachtet die vier vorgestellten Bereiche von Software Analytics „Gather“, „Measure“, „Analyse“ und „Visualize“, setzt diese in einen Problemkontext und ermöglicht einen Ausblick auf die nächsten Schritte. Das Template zum Software Analytics Canvas zeigt **Abbildung 2**. Die Inhalte der einzelnen Abschnitte betrachten wir im nächsten Abschnitt anhand eines Beispiels.

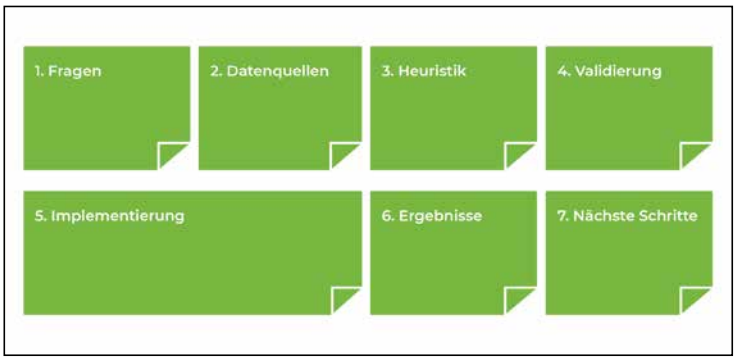


Abb. 2: Software Analytics Canvas

Abb. 3: Startoberfläche der Jupyter Notebooks

Auf die Plätze, fertig, los!

Um nicht nur in der schnöden Theorie zu bleiben, wollen wir uns nun Software Analytics in der Praxis nähern. Dafür wollen wir uns anhand des aus Teil 1 der Artikelserie bekannten Beispielsystems [6] folgender Fragestellung widmen: Existieren zyklische Zusammenhänge zwischen den fachlichen Teilbereichen des Systems?

Vor dem Start noch eine kleine Anmerkung: Sowohl die hier durchgeführte Analyse als auch die beiden anderen, eingangs gestellten Fragen sind Bestandteile des Beispielprojekts und finden sich im Projektordner unter /jupyter. Darin befindet sich darüber hinaus ein leeres Template für eigene Analysen.

Um mit Software Analytics starten zu können, muss jQAssistant in das Projekt integriert und die Neo4j-Datenbank mittels *mvn clean verify* mit Inhalten gefüllt werden. Die Integration findet sich in der *pom.xml* des Beispiels und wurde im ersten Teil dieser Artikelserie beschrieben. Danach ist es notwendig, die Analyseplattform Jupyter Notebook zu starten. Es empfiehlt sich, das bereitgestellte Docker Image *jqassistant/jupyter-notebook:1.11.0* zu nutzen. Dafür muss nur eine *jqa-docker-compose.xml*-Datei im Projektordner angelegt und mit den in Listing 1 gezeigten Inhalten gefüllt werden. Hier wird auch der lokale Projektordner */jupyter* als Bind-Mount zur Verfügung gestellt, sodass neu erstellte Notebooks und Änderungen an bestehenden Notebooks in Jupyter auch direkt im Projekt hinterlegt sind.

Anschließend lässt sich der Stack mit *docker-compose -f jqa-docker-compose.xml u*p starten. In der entstehenden Ausgabe auf der Kommandozeile ist dann der URL zu Jupyter angegeben (der URL ist aufgrund des Access-Tokens ausführungsspezifisch und beinhaltet: *127.0.0.1:8888*). Nach dem Öffnen zeigt sich die in **Abbildung 3** dargestellte Ansicht. Im Ordner *work* befinden sich die im Verlauf des Artikels betrachteten Analysen.

Listing 1

```
version: '3'
services:
  neo4j:
    image: neo4j:3.5.24
    ports:
      - 7474:7474
      - 7687:7687
    environment:
      - NEO4J_AUTH=none
    volumes:
      - ./target/jqassistant/store:/data/databases/graph.db
  jupyter:
    image: jqassistant/jupyter-notebook:1.11.0
    ports:
      - 8888:8888
    environment:
      - NEO4J_URL=http://neo4j:7474
    volumes:
      - ./jupyter:/home/jovyan/work
```

```
%%cypher
MATCH          (:Package{fqn: "com.buschmais.gymmanagement"})-[:CONTAINS]->(p:Package)
MERGE          (f:FunctionalComponent{name: p.name})
WITH           p, f
OPTIONAL MATCH (p)-[:CONTAINS]->(t:Type)
MERGE          (f)-[:CONTAINS]->(t)
RETURN         f.name AS FunctionalComponent, count(t) AS ContainedTypes
```

4 rows affected.

FunctionalComponent	ContainedTypes
keycard	1
user	6
training	8
attendance	8

Abb. 4: Die Umsetzung der Analyse im Jupyter Notebook

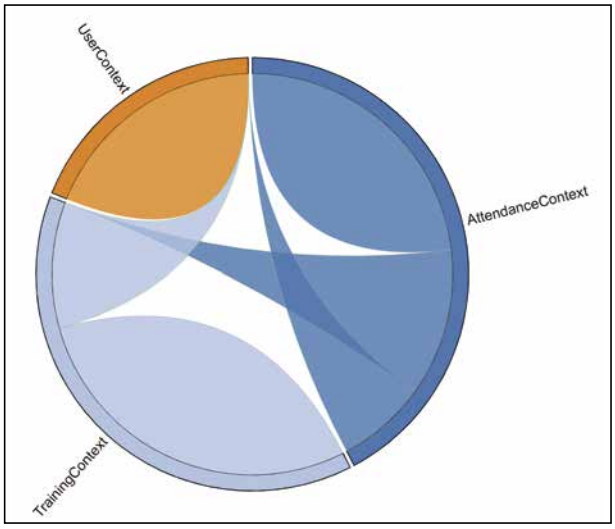


Abb. 5: Die Ergebnisse der Analyse im Jupyter Notebook

Nun wollen wir das zweite Analyse-Notebook betrachten, in dem es um die Frage der fachlichen Dekomposition geht und uns anhand des Software Analytics Canvas vortasten.

- Fragestellung (Question):
  - Welches Problem oder welche Fragestellung soll in dieser Analyse betrachtet werden? Hier ist es wichtig, möglichst präzise zu formulieren, um eine zielgerichtete Analyse zu ermöglichen.
  - Beispiel: Existieren zyklische Zusammenhänge zwischen den fachlichen Teilbereichen des Systems?
- Datenquellen (Data Sources):
  - Welche Daten benötigen wir zur Beantwortung der Frage? Handelt es sich nur um die Java-Implementierung oder werden auch Informationen des Build-Systems, des Dependency-Managements, der Git-Historie oder Sonstiges benötigt? Welche Bedingungen müssen die Daten erfüllen?
  - Beispiel: Die Java-Strukturen müssen eingescannt und in der Datenbank abfragbar sein. Zusätzlich müssen sich die fachlichen Teilbereiche identifizieren lassen.
- Heuristiken (Heuristics):
  - Mit welchen Annahmen können wir unsere Analyse vereinfachen? Welches Vorwissen können wir in die Analyse einbringen? Das können beispielsweise bekannte Strukturen oder auch Systembestandteile sein, die nicht betrachtet werden müssen.
  - Beispiel: Jedes Package unter *com.buschmais.gymmanagement* stellt einen fachlichen Teilbereich dar. Klassen, die in diesen und darunterliegenden Packages enthalten sind, zählen zu diesem fachlichen Teilbereich.
- Validierung (Validation):
  - Welche Ergebnisse erwarten wir von unserer Analyse? Es soll klar erkennbar sein, wie eine verständliche, zielgruppenorientierte Aufbereitung aussehen soll und wie sowie durch wen die Ergebnisse verwendet werden.
  - Beispiel: Es soll eine grafische Übersicht über die fachlichen Teilbereiche und deren Beziehungen untereinander erstellt werden. Zyklische Abhängigkeiten sollen zusätzlich tabellarisch dargestellt werden. Fachexperten sollen die Aufteilung auf Korrektheit prüfen und, im Falle von Fehlern, deren Behebung durch die Projektleitung planen und durch Entwickler beheben lassen.
- Umsetzung (Implementation):
  - Wie kann die Analyse mit den zur Verfügung stehenden Mitteln umgesetzt werden? Dieser Teil beinhaltet nicht nur die theoretische Betrachtung, sondern auch die Umsetzung der benötigten Abfragen.
  - Beispiel: Die Umsetzung ist in **Abbildung 4** anhand des Jupyter Notebooks dargestellt und weiter unten näher erläutert.
- Ergebnisse (Results):
  - Was sind die wichtigsten Ergebnisse unserer Analyse? Hieraus muss deutlich werden, was die Antwort auf die eingangs gestellte Frage ist. Visualisierungen können unterstützend verwendet werden.
  - Beispiel: Die Ergebnispräsentation ist in **Abbildung 5** anhand des Jupyter Notebooks dargestellt und weiter unten näher erläutert.
- Nächste Schritte (Next Steps):
  - Welche Schritte lassen sich von den Analyseergebnissen ableiten? Ist es notwendig, zusätzliche Analysen für Teilaspekte durchzuführen oder beantwortet die Analyse die Frage in ausreichender Form? In letzterem Fall: Welche Aktionen, beispielsweise Refaktorisierungen des Codes, sind nun notwendig?
  - Beispiel: Die Analyse hat ergeben, dass keine zyklischen Abhängigkeiten zwischen fachlichen Teilaspekten existieren. Dieser Zustand soll mit jQAssistant dokumentiert und abgesichert werden.

Die Umsetzung der Analyse erfolgt im fünften Abschnitt des Software Analytics Canvas und ist in **Abbildung 4** ausschnittsweise dargestellt. Das dargestellte Cypher-Statement befindet sich in einer sogenannten Codezelle, die bei Ausführung beliebigen Python-Code oder, durch Plug-ins ermöglicht, Cypher-Statements ausführen und deren Ergebnis direkt rendern kann. Die dargestellte Abfrage stellt die Grundbedingung für die Analyse her, indem fachliche Komponenten identifiziert und explizit im Graphen modelliert werden.

Ergebnisse von Abfragen müssen aber nicht direkt dargestellt werden, sondern können wie in Listing 2 ge-

zeigt auch in Variablen gespeichert werden. Das ist hilfreich, wenn die Ergebnisse entweder noch transformiert werden müssen oder wenn diese als Teil des sechsten Abschnitts, der Ergebnispräsentation, dargestellt werden sollen. In der gezeigten Abfrage, die sich ebenfalls in einer Codezelle befindet, werden die Abhängigkeiten zwischen Typen auf der Komponentenebene aggregiert und im Anschluss zurückgegeben.

Das Ergebnis kann dann wie in **Abbildung 5** als Chord-Diagramm dargestellt werden, was eine sehr praktische Möglichkeit ist, um Abhängigkeiten und Kopplungsstärke zwischen Elementen übersichtlich darzustellen. Dafür wird in diesem Beispiel D3JS verwendet, das über ein im Jupyter Docker Image bereitgestelltes Template erzeugt werden kann. Dargestellt ist das in Listing 3. Typischerweise kommt man aber auch mit der weiter oben erwähnten Bibliothek pygal sehr weit, da sie eine Vielzahl von Standarddiagrammen bereitstellt.

Bereits hier sehen wir, dass das betrachtete System keine Zyklen aufweist. Interessant wird es, wenn zukünftig Zyklen auftreten sollten. Das in der Rückgabe von Listing 2 enthaltene *weight*, das die Stärke der Kopplung darstellt, ist ein guter Indikator für entstehende Aufwände. Je höher der Wert ist, desto stärker sind zwei fachliche Teilbereiche miteinander gekoppelt und desto aufwendiger wäre es, diese Zyklen aufzulösen. Damit zeigt sich, dass die Ergebnisse der Softwareanalyse auch genutzt werden können, um grobe Einschätzungen in Richtung Risiko und Aufwand zu treffen, eine Möglichkeit, die in der IDE schwierig bis unmöglich wäre.

Wurde das zugehörige Jupyter Notebook geöffnet, kann die Analyse über die Navigationsleiste via Kernel | Restart & Run All ausgeführt werden. So ist es möglich, jederzeit, auch bei Veränderungen im Source Code, die Analyse nachzuvollziehen und dabei auch zu erkennen, ob es zu einer Verbesserung oder Verschlechterung der Situation gekommen ist. Ein weiterer Vorteil ist, dass nicht nur der Ersteller detaillierten Einblick erhält, sondern auch die umsetzenden Entwickler das Notebook als Leitlinie für Änderungen verwenden können. Möchte man die Ergebnisse separat zur Verfügung stellen, bietet sich außerdem der HTML-Export via File | Download as | HTML an.

Listing 2

```
# Result is stored in a variable for usage in the result section
dependencies = %cypher \
  MATCH  (fC1:FunctionalComponent)-[:CONTAINS]->(t1:Type), \
         (fC2:FunctionalComponent)-[:CONTAINS]->(t2:Type), \
         (t1)-[d:DEPENDS_ON]->(t2) \
  WITH   fC1, fC2, sum(d.weight) AS weight \
  MERGE  (fc1)-[:DEPENDS_ON{weight: weight}]->(fc2) \
  RETURN fc1.name AS Source, fc2.name AS Target, weight AS X_Count
```

Listing 3

```
text = Template(open('../vis/chord/chord-diagram.html', 'r').read()
  .replace("\n",""")).substitute({
    'chord_data': dependencies.get_dataframe().to_csv(index = False)
    .replace("\r\n","\\n").replace("\n","\\n"), 'container': 'bc-chord-diagram'})

HTML(text)
```

Fazit und Ausblick

Um der Komplexität von Softwaresystemen Herr zu werden, bietet sich der strukturierte Ansatz mit Software Analytics an. Damit wird es möglich, Zusammenhänge und Probleme im System zu identifizieren, zu analysieren und Lösungswege zu definieren. Dafür stellt die Kombination aus jQAssistant, Neo4j und Jupyter Notebook eine mächtige Lösung bereit, die schon mit wenig Aufwand verwertbare Ergebnisse liefert, aber gleichzeitig erfahrenen Nutzern nahezu unbegrenzte Möglichkeiten bietet. So ist es endlich möglich, das gute alte Bauchgefühl mit Zahlen, Daten und Fakten zu untermauern, fundiert zu kommunizieren und begründete Entscheidungen zu treffen.

Dieser Artikel stellt die Grundprinzipien hinter und Best Practices in der Umsetzung von Software Analytics dar. Im Beispielprojekt finden sich dafür drei vorgefertigte Analysen. Wer mehr Beispiele an einem größeren Projekt sehen möchte, dem sei das Repository unter [7] empfohlen. Alle Analysen können ohne lokalen Set-up-Aufwand direkt im Browser nachvollzogen werden.

Im nächsten und letzten Teil der Artikelserie schauen wir, wie wir das Erstellen der Dokumentation zu einer interessanten Angelegenheit machen und gleichzeitig Abweichungen zur Implementierung proaktiv verhindern können. Auch an dieser Stelle gilt wieder: Viel Spaß beim Ausprobieren und Erforschen – es lohnt sich! Ich freue mich darauf, von euren Erkenntnissen zu hören.

Stephan Pirnbaum ist Consultant bei der BUSCHMAIS GbR. Er beschäftigt sich leidenschaftlich gern mit der Analyse und strukturellen Verbesserung von Softwaresystemen im Java-Umfeld. In Vorträgen und Workshops präsentiert er seine gesammelten Erfahrungen und genutzten Methodiken.

stephan.pirnbaum@buschmais.com

Links & Literatur

[1] Stephan Pirnbaum: „Ein Tool, viele Möglichkeiten“; in Java Magazin 6.2022

[2] https://jupyter.org

[3] https://pandas.pydata.org

[4] https://www.pygal.org/en/stable/

[5] https://www.feststelltaste.de/software-analytics-canvas/

[6] https://github.com/buschmais/The-Perfect-Greenfield

[7] https://github.com/buschmais/software-analytics-starter

Eine Einführung in jQAssistant – Teil 3

Architektur dokumentieren und validieren

„Architektur- und Softwaredokumentation zu erstellen und konsumieren macht Spaß!“ – das würde wohl kaum jemand sagen, oder? Das ist sehr schade, denn Softwaresysteme werden technisch und fachlich immer komplexer, und eine gute Dokumentation kann den entscheidenden Unterschied zwischen langfristig erfolgreichen und fehlgeschlagenen Projekten ausmachen. Doch die Motivation ist gering. Lebendige und codenahe Dokumentation ist hier der Schlüssel zum Erfolg.

von Stephan Pirnbaum

Wenn wir ehrlich sind, ist Dokumentation ein leidiges Thema. Wir wissen, dass sie notwendig ist, um nachhaltig gute Software entwickeln zu können. Doch aufgrund sperriger Word-Dokumente, unübersichtlicher Wikis und veralteter PowerPoint-Präsentationen schaut man bei Fragen oder Unklarheiten gar nicht erst in der Dokumentation nach und ist entsprechend wenig motiviert, sie aktuell zu halten. Es scheint, als schleiche sich über die Jahre eine gewisse Müdigkeit ein, die darin begründet liegt, dass die Dokumentation noch in keinem Projekt erste Priorität hatte. Am Ende ist das ein Teufelskreis, bei dem sich schlechte Dokumentation und Mangel an Motivation gegenseitig befeuern.

Lassen wir es also am besten gleich bleiben, das Themenfeld Dokumentation wieder aufleben zu lassen? Entwickeln wir weiterhin mit vollem Einsatz moderne Anwendungen mit den neuesten und tollsten Frameworks und lassen die Dokumentation verstauben? Oder können wir dem Teufelskreis entrinnen und mit genauso hippen Technologien die Dokumentation wiederbeleben, wie wir es regelmäßig mit der eigentlichen Software tun? Die Antwort auf diese Fragen existiert bereits in Form der sogenannten lebendigen Dokumentation.

Was das ist, warum und wie lebendige Dokumentation den Weg aus dem Teufelskreis weist, soll dieser Artikel zeigen.

Artikelserie

Teil 1: Ein Tool, viele Möglichkeiten
Teil 2: Versteh dein System mit Software Analytics
Teil 3: Architektur dokumentieren und validieren

Klassische Dokumentation und ihre Probleme

Dokumentation dient dazu, Wissen zu manifestieren. Neu ins Team gekommene Entwickler können sie während des Onboardings konsultieren, um einen Überblick über die Architektur, ihre Treiber und Hintergründe sowie Begrifflichkeiten zu erhalten. In der Entwicklung dient die Dokumentation dazu, offene Fragen zu beantworten und eine Leitlinie zu bieten. Außerdem dient sie als Kommunikations- und Entscheidungsgrundlage für Änderungen. Das setzt natürlich voraus, dass sie überhaupt vorhanden, aussagekräftig und aktuell ist.

Klassischerweise findet sich die Architekturdokumentation aber in sperrigen Word-Dokumenten und unübersichtlichen Wikis. Somit gibt es bereits eine räumliche Lücke zwischen der Dokumentation und dem zu dokumentierenden Code. Diese Lücke erschwert es, Fragen aus der Entwicklung schnell zu beantworten und Fehler oder fehlende Informationen zu beheben bzw. zu ergänzen. Nicht dass es nicht möglich wäre, aber hier spielt nicht nur der hohe Aufwand

durch den Medienbruch, sondern auch der gefühlte, viel schwerwiegendere Eindruck des Aufwand-Nutzen-Verhältnisses eine Rolle. Wird im Projekt nicht explizit darauf geachtet, dass die Dokumentation regelmäßig einer Review unterzogen und bei Bedarf aktualisiert wird, kommt es schnell zu einem sogenannten Documentation-Code-Gap (**Abb. 1**). Und selbst wenn, ist es mit klassischen Tools schwierig, Änderungen nachzuvollziehen, wie es bei Pull-Request-Reviews und mit *git diff* in der Umsetzung schon lange Standard ist. Neben diesen internen Faktoren sind es häufig auch externe, die die Pflege der Dokumentation unattraktiv machen: Unter Zeitdruck wird immer zuerst versucht, das Feature umzusetzen, die passende Dokumentation ist dann selten Teil der Definition-of-Done. Hier rückt der langfristige Nutzen im Vergleich zum kurzfristigen Aufwand in den Hintergrund.

Das Problem ist, dass die Dokumentation irgendwann so stark von der Umsetzung abweicht, dass sich die Frage stellt, ob es überhaupt noch sinnvoll ist, in sie zu investieren. Lohnt der Aufwand, eine Dokumentation zu erstellen und wird sie von den Entwicklern überhaupt noch als relevant angesehen (**Abb. 2**)? Relevanz ist hier ein gutes Stichwort, denn Dokumentation sollte nicht um ihrer selbst willen existieren, sondern reale Probleme lösen, z. B. Wissenstransfer bei Onboardings.

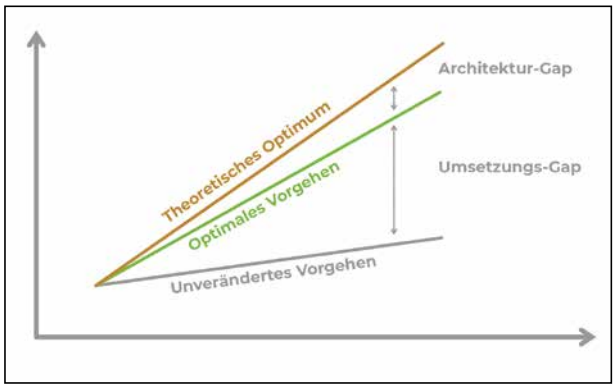


Abb. 1: Soll-Ist-Gap zwischen Architektur, Dokumentation und Umsetzung

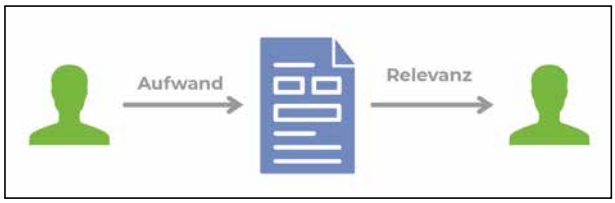


Abb. 2: Aufwand und Relevanz bei der Dokumentationserstellung

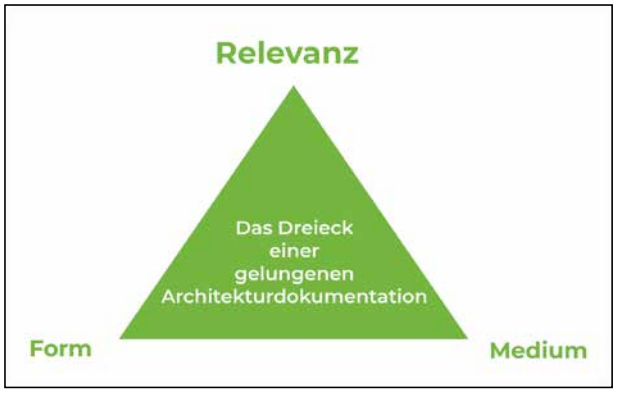


Abb. 3: Das Dreieck einer gelungenen Architekturdokumentation

Worauf es (wirklich) ankommt

Die Güte einer Architekturdokumentation lässt sich nicht an einer einzelnen Eigenschaft festmachen. Nur weil die Dokumentation bereits mit AsciiDoc geschrieben wird und nicht mehr in den SharePoints dieser Welt vor sich hin vegetiert, heißt das nicht, dass sie korrekt und für den Konsumenten hilfreich ist. Umgekehrt hilft relevanter Inhalt nicht, wenn dieser schlecht strukturiert und kaum auffindbar ist. Die Güte der Dokumentation ist vielmehr eine Kombination aus den drei Aspekten Relevanz, Form und Medium (**Abb. 3**).

Das wichtigste Kriterium bei der Bewertung der Qualität der Dokumentation stellt ihre Relevanz dar. Erstelle ich eine Dokumentation, muss ich zunächst bedenken, wer die Zielgruppe ist. Das betrifft sowohl die Inhalte als auch die Detailtiefe. Das wird schon bei der Betrachtung von Entwickler- vs. Architekturdokumentation deutlich. Während bei Ersterer der Entwickler nach Umsetzungsvorgaben sucht, an die er sich konkret halten muss, dient Letztere dazu, einen Überblick über das System zu bekommen, und ist für den Architekten die Arbeits- und Entscheidungsgrundlage. Auch die Aktualität der Dokumentation beeinflusst ihre Relevanz. Natürlich ist eine Dokumentation, die die mittlerweile veränderte Architektur von vor drei Jahren darstellt, bei aktuellen Fragen nicht hilfreich. Das größere Problem ist jedoch das sinkende Vertrauen in die Dokumentation, sodass sie gleich gar nicht mehr konsultiert wird. Um dem entgegenzuwirken, bieten Tools wie jQAssistant die Möglichkeit, eine Dokumentation einerseits auf Basis der Implementierung zu generieren und sie anderseits zu nutzen, um die Implementierung gegen die Architektur zu validieren.

Das zweite Kriterium bei der Dokumentationserstellung ist die Form. Als Entwickler wissen wir, wie wichtig die richtige Strukturierung des Source Code für das Verständnis ist. Das Gleiche gilt für die Dokumentation. Ein logischer, wohlstrukturierter und zielgruppenorien-

Über diese Artikelserie

Dieser Artikel ist der letzte Teil einer dreiteiligen Artikelserie. Er stellt moderne und nachhaltige Ansätze zur Dokumentation von Softwaresystemen vor und geht dabei auf häufig anzutreffende Fehler ein. Mit jQAssistant und weiteren Open-Source-Tools wie Context Mapper und AsciiDoc wird ein Weg aufgezeigt, Code und Dokumentation wieder näher zusammenzubringen. Somit setzt der letzte Teil der Artikelserie die Einführung in jQAssistant und Software Analytics fort und ermöglicht es nicht nur, Probleme zu identifizieren, sondern auch, diese zu vermeiden und somit langfristigen Mehrwert für alle Projektbeteiligten zu schaffen.

tierter Aufbau hilft sowohl dem Ersteller als auch dem Leser. Selbstverständlich sollte von Anfang an ein grobes Gerüst für die Dokumentation stehen, welches später nur noch gefüllt werden muss. So können Struktur und Stringenz zeitig bewertet und verbessert werden. Einen guten Startpunkt stellt für die Architekturdokumentation das Template Arc42 [1] dar. Der Vorteil ist, dass derartige Templates bereits einige Evolutionsstufen hinter sich haben und ihre Strukturierung mit Hilfe zahlreicher Projekte in der Praxis validiert wurde. Insbesondere, wenn man häufig in neuen Projekten arbeitet, bietet die Verwendung von Arc42 zusätzlich eine gewisse Stabilität, die das Zurechtfinden stark erleichtert. Die Verwendung eines Template erlaubt zudem Abweichungen. So kann im Projekt individuell entschieden werden, welche Abschnitte zum jeweiligen Zeitpunkt am wichtigsten für das Projektteam sind.

Last, but not least muss die Frage nach dem Dokumentationsmedium geklärt werden. Häufig anzutreffen sind mittlerweile Arc42-basierte Architekturdokumentationen in Word oder Confluence. Der Schwachpunkt liegt darin, dass derartige Dokumentationen weit entfernt von der Umsetzung sind. So sind sie für Entwickler schwerer zu erreichen und eine Nachverfolgung von Änderungen ist insbesondere bei Word über die Zeit schwer bis gar nicht möglich. Auch die Zuordnung der Dokumentation zu bestimmten Softwareversionen erweist sich schnell als unlösbare Aufgabe. Erfahrungsgemäß stellt dieses zusätzliche Medium im Entwickleralltag ein großes Motivationsproblem dar, wenn es um Aktualisierung und Erweiterung der Dokumentation geht. Insbesondere neue Entwickler überspringen aufgrund der räumlichen Distanz, der schlechten Navigierbarkeit und der unbekannten Aktualität häufig den Schritt, die Dokumentation zu lesen, und arbeiten sich direkt mit Hilfe des Codes ein. Über die Zeit hat sich ein Ansatz namens Docs-as-Code [2] etabliert, der das Ziel hat, diese Hemmschwelle durch den Medienbruch und die damit einhergehende Komplexität in der Pflege herabzusetzen. Ermöglicht wird das dadurch, dass die Dokumentation wie Source Code behandelt wird. Das bedeutet:

- die Ablage der Dokumentation im Source-Code-Repository
- die Erstellung der Dokumentation in einem versionierbaren, leichtgewichtigen Format, z. B. AsciiDoc
- die Integration der Dokumentationserstellung (Rendering) in den Build-Prozess
- das Schließen des Documentation-Code-Gap durch Generierung von Dokumentation und Validierung der Umsetzung

Ein häufig anzutreffendes Gegenargument stellt die Erreichbarkeit der Dokumentation für Nichtentwickler dar. Dieses valide Problem kann jedoch gelöst werden, indem die während des Build-Prozesses erstellte Dokumentation z. B. in einem Nexus-Site-Repository abgelegt wird oder eine direkte Integration durch die Doc-Toolchain Richtung Confluence erfolgt [3]. Das betrifft natürlich nur die Leserschaft, auf die Arbeitsweise der Ersteller der Dokumentation hat das keine Auswirkung.

Tools, Tools und noch mehr Tools

Gesamtheitlich betrachtet ist also eine Arc42-basierte Dokumentation, verfasst in AsciiDoc und abgelegt im Source-Code-Repository, ein großartiger erster Schritt in die richtige Richtung. Um langfristig mit der Dokumentation glücklich zu werden, muss sie aktuell gehalten werden. Ein erneutes Aufklaffen des zuvor beschriebenen Documentation-Code-Gap lässt uns sonst schnell wieder in alte Muster fallen. Zur Sicherstellung der Aktualität und Korrektheit bieten sich grundsätzlich zwei Wege an:

- Generierung der Dokumentation auf Basis des Source Code
- Validierung des Source Code gegen die Dokumentation
- Für diesen Ansatz hat sich der Begriff „lebendige Dokumentation“ etabliert. Der Grundgedanke dabei ist, Brücken zwischen Dokumentation und Code zu schlagen und somit für einen konsistenten Zustand zu sorgen. Möchte man beispielsweise dokumentieren, welche Endpunkte ein System bereitstellt, ist es natürlich möglich, diese Informationen manuell in einer Tabelle zusammenzutragen und bei jeder Änderung zu aktualisieren. Sinnvoll ist das durch die potenziell entstehende Lücke zwischen Dokumentation und Code aber nicht. Zugegeben, das Problem ist bereits seit Jahren durch Swagger [4] gelöst, mit dem die Dokumentation direkt an die Implementierung heranrückt und anschließend eine Übersicht über die existierenden Endpunkte generiert werden kann. Die Dokumentation der Endpunkte ist aber nur ein Teilaspekt. Durchsucht man die Dokumentation verschiedenster Systeme nach ähnlichen Generierungsansätzen, beispielsweise einer Context Map (also der fachlichen Dekomposition des Systems) oder Diagrammen des C4 Models [5] aus dem Code, sieht es schnell düster aus. Für die Prüfung der Implementierung gegen die Architekturdokumentation haben ebenfalls nur die wenigsten Projekte eine Lösung parat.
- Eigentlich ist das verwunderlich, denn in der Dokumentation beschreiben wir Architekturkonzepte und visualisieren diese in Diagrammen, die später im Code implementiert werden. So werden beispielsweise Bounded Contexts aus der Dokumentation als Top-Level-Maven-Module umgesetzt, und asynchrone Beziehungen zwischen Microservices landen als Event Publisher und Listener mit gemeinsamen Payload DTOs im Code. Legt man sich nun noch das in **Abbildung 4** vorausgefüllte Arc42-Template daneben, erkennt man schnell, dass viele Abschnitte als lebendige Dokumentation festgehalten werden können.

Der Weg zur lebendigen Dokumentation besteht aus einer Kombination mehrerer Open-Source-Tools. Dazu gehören zunächst einmal AsciiDoc als Mittel zur Dokumentation, Arc42 als Template zum Befüllen sowie jQAssistant, um sie lebendig zu gestalten. In den ersten beiden Teilen dieser Artikelserie haben wir jQAssistant als Tool kennengelernt, um die Strukturen eines Softwaresystems in eine Neo4j-Graphdatenbank einzulesen und Analysen auf dieser durchzuführen. Ein weiteres wichtiges Feature von jQAssistant stellt die Möglichkeit dar, AsciiDoc-Dokumente während des Builds als HTML zu rendern, wobei die im AsciiDoc enthaltenen Datenbankabfragen ausgeführt und ihre Ergebnisse direkt mit im Ergebnis-HTML angereichert werden. So ist es möglich, Abfragen zu nutzen, um beispielsweise Architekturentscheidungen abzusichern und mit den aufgedeckten, in der Dokumentation enthaltenen tagesaktuellen Verletzungen direkt einen Überblick über die Architekturerosion zu erhalten (siehe Arc42, Kapitel 11: „Risiken und technische Schulden“).

Soll es grafisch werden, kann jQAssistant auch PlantUML- und ContextMapper-Diagramme [6] generieren sowie ContextMapper- und C4-PlantUML-Diagramme [7] einlesen. Damit wird es möglich, Diagramme, die im Rahmen der Architekturdokumentation sowieso erstellt werden müssen, zu nutzen, um die Soll-Architektur in der Graphdatenbank anzureichern und für einen Soll-Ist-Abgleich zu nutzen.

Abb. 4: Exemplarisch angereichertes Arc42-Template

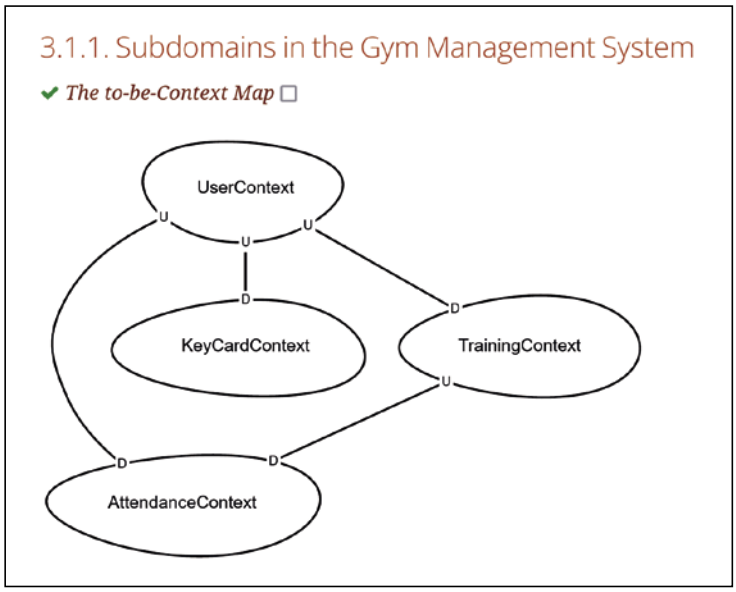


Abb. 5: Die gerenderte Context Map

Ein bisschen Code muss sein

Exemplarisch wollen wir uns noch einmal anschauen, wie ein Projekt-Set-up aufgestellt sein kann, um die fachliche Architektur einer Anwendung zu dokumentieren. Dabei soll die Soll-Architektur als Context Map dokumentiert und gleichzeitig genutzt werden, um die Korrektheit der Implementierung zu prüfen. Abweichungen zwischen der dokumentierten Soll-Architektur und dem Ist-Zustand sollen ebenfalls in die Dokumentation aufgenommen werden – das Ganze natürlich möglichst automatisiert und basierend auf dem bekannten Beispielprojekt [8].

Für die Dokumentation ist das als AsciiDoc zur Verfügung gestellte Arc42-Dokument empfehlenswert, das im Projekt unter *documentation/arc42* abgelegt ist. Für jedes Kapitel existiert ein separates Dokument, die alle in der index.adoc zusammengezogen werden. Unsere fachliche Soll-Architektur gehört dabei in Kapitel 3 „System Scope and Context“.

Für die grafische Dokumentation der fachlichen Architektur wollen wir kein Diagramm malen und dann im Repository ablegen, denn das wäre auch wieder schwer änderbar und schlecht zu versionieren. Vielmehr nutzen wir die textuelle Beschreibungssprache von Context Mapper, um die Dokumentation unter documentation/context-mapper/Context-Map.cml zu beschreiben (Listing 1). Später werden wir es so umsetzen, dass auch die Context Map als Diagramm (**Abb. 5**) gerendert wird.

Eine kleine Info vorab: Es existiert für jQAssistant ein Plug-in, das auch die Context Map mit all ihren Bestandteilen in die Graphdatenbank einliest. Die Integration wird am Ende noch gezeigt. Damit erreichen wir, dass sowohl die Java-Umsetzung als auch die Soll-Architektur in der gleichen Datenbank landen. Jetzt stellt sich nur noch die Frage, wie wir die zwei Welten zusammenbringen.

Zunächst muss man sich überlegen, wie ein Bounded Context im Code umgesetzt wird, denkbar wären zum Beispiel separate Maven-Module oder entsprechend benannte Java-Packages. Da diese Information jedoch recht implizit ist und sich explizit selbst dokumentierender Code während der Entwicklung auszahlt, soll

Listing 1: Context-Map.cml

```
ContextMap GymSystem {
  type = SYSTEM_LANDSCAPE
  state = TO_BE

  contains UserContext
  contains KeyCardContext
  contains TrainingContext
  contains AttendanceContext

  UserContext [U]->[D] KeyCardContext
  UserContext [U]->[D] TrainingContext
  UserContext [U]->[D] AttendanceContext
  TrainingContext [U]->[D] AttendanceContext
}

BoundedContext UserContext { }
BoundedContext KeyCardContext { }
BoundedContext TrainingContext { }
BoundedContext AttendanceContext { }
```

Listing 2: jqassistant/arc42/visualizations.adoc

```
[[arc42:ContextMap]]
[source,cypher,role=concept,reportType="context-mapper-diagram"]
.The to-be-Context Map
----
MATCH (c:ContextMapper:ContextMap{state: 'TO_BE'}),
      (c)-[:SHOWS]->(b1:ContextMapper:BoundedContext)
OPTIONAL MATCH (b1)-[d:DEFINES_DEPENDENCY]->(b2:ContextMapper:
                                                BoundedContext)
RETURN b1, d, b2
----
```

Listing 3

```
<constraint id="adr:IllegalDependenciesBetweenBoundedContext">
  <requiresConcept refId="jmolecules-ddd:BoundedContextPackage"/>
  <description>...</description>
  <cypher><![CDATA[
    MATCH  (bC1:JMolecules:BoundedContext),
           (bC2:JMolecules:BoundedContext),
           (bC1)-[:DEPENDS_ON]->(bC2),
           (c:ContextMapper:ContextMap{state: 'TO_BE'})
    WHERE  bC1 <> bC2 AND
           NOT (c)-[:SHOWS]->(:ContextMapper:BoundedContext{name:
                    bC1.name})-[:DEFINES_DEPENDENCY]->(:ContextMapper:
                         BoundedContext{name: bC2.name})<-[:SHOWS]-(c)
    RETURN bC1 AS Source, bC2 AS Target
  ]]></cypher>
</constraint>
```

der Weg über jMolecules [9] gegangen werden. Diese Bibliothek stellt Annotationen und Interfaces für Architekturkonzepte, z. B. Bounded Contexts, technische Schichten, Konzepte des Tactical DDD oder auch für Event-driven Systems bereit. Der Vorteil ist, dass jQAssistant auch dafür ein Plug-in mitbringt, wodurch diese Informationen direkt im Graphen angereichert werden. Fachliche Komponenten können wir somit einfach in

Listing 4

```
[[section-technical-risks]]
== Risks and Technical Debts

include::jQA:Summary[constraints="*",importedConstraints="*"]

include::jQA:Rules[constraints="*"]
```

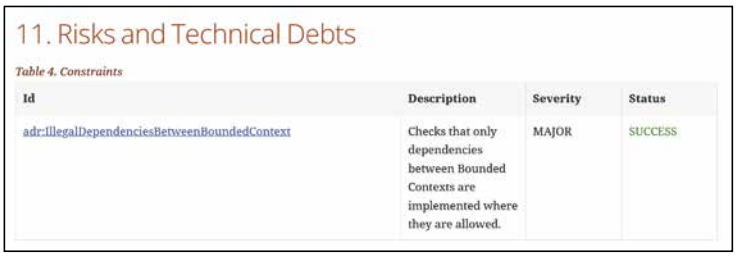
Listing 5

```
<plugin>
  <groupId>com.buschmais.jqassistant</groupId>
  <artifactId>jqassistant-maven-plugin</artifactId>
  <executions> <execution>
    <id>default-cli</id>
    <goals> ... </goals>
    <configuration>
      <groups>
        <group>jmolecules-ddd:Default</group>
        <group>gym-management:Default</group>
      </groups>
      <scanIncludes>
        <scanInclude>
          <path>${project.basedir}/documentation/context-mapper</path>
        </scanInclude>
      </scanIncludes>
      <reportProperties>
        <asciidoc.report.rule.directory>${project.basedir}/documentation/
          arc42</asciidoc.report.rule.directory> <asciidoc.report.file.include>
                                          index.adoc</asciidoc.report.file.include>
      </reportProperties>
    </configuration>
  </execution> </executions>
  <dependencies>
    <dependency>
      <groupId>org.jqassistant.contrib.plugin</groupId>
      <artifactId>jqassistant-context-mapper-plugin</artifactId>
    </dependency>
    <dependency>
      <groupId>org.jqassistant.contrib.plugin</groupId>
      <artifactId>jqassistant-jmolecules-plugin</artifactId>
    </dependency>
  </dependencies>
</plugin>
```

package-info.java-Dateien markieren. Mit ein bisschen Magie des Plug-ins werden dann auch automatisch alle im annotierten Package enthaltenen Klassen dieser fachlichen Komponente zugeordnet:

```
@BoundedContext(name = "TrainingContext")
package com.buschmais.gymmanagement.training;

import org.jmolecules.ddd.annotation.BoundedContext;
```



11. Risks and Technical Debts

Table 4. Constraints

Id	Description	Severity	Status
adr:IllegalDependenciesBetweenBoundedContext	Checks that only dependencies between Bounded Contexts are implemented where they are allowed.	MAJOR	SUCCESS

Abb. 6: Das Ergebnis des Soll-Ist-Vergleichs

Aus den vorhergehenden Teilen dieser Artikelserie sind bereits jQAssistants Constraints und Concepts bekannt. Letztere können verwendet werden, um Diagramme während des Build-Prozesses zu rendern. So kann zum Beispiel die Context Map auf Basis der zuvor angereicherten Information tagesaktuell visualisiert werden (Listing 2).

Bei jeder Änderung an der textuellen Context Map wird somit während des Builds eine neue grafische Context Map generiert. Die Zeiten, in denen in PowerPoint Diagramme erstellt, aktualisiert und exportiert wurden, um sie anschließend ins Wiki zu kopieren, sind also endgültig vorbei. Am Rande sei noch erwähnt, dass für C4-Diagramme ein ähnliches Plug-in existiert [10], auf das hier aber nicht näher eingegangen werden soll.

Da im Graphen die Soll-Architektur aus der Context Map und die Ist-Architektur via jMolecules verfügbar ist, wollen wir noch prüfen, ob es den gefürchteten Documentation-Code-Gap gibt. Dafür definieren wir in jqassistant/adr/003-Implementing-Bounded-Contexts.xml ein Constraint (Listing 3), also eine Architekturregel, die von jQAssistant während des Build ausgeführt wird. Liefert diese Regel ein nichtleeres Ergebnis zurück, gilt sie als verletzt.

Da die Verletzungen als technische Schulden zu einer Architekturentscheidung angesehen werden können, binden wir das Ergebnis in Abschnitt 11 „Technical Risks“ von Arc42 ein (Listing 4). In unserem Projekt existieren keine Verletzungen, weswegen das Ergebnis wie in **Abbildung 6** aussieht. Verletzungen würden ansonsten tabellarisch aufgelistet. Listing 5 zeigt abschließend die Integration von jQAssistant mitsamt der Plug-ins in das Projekt.

Fazit und Ausblick

„Das Erstellen und Konsumieren von Architektur- und Softwaredokumentation macht Spaß!“ – während diese Aussage am Anfang noch komplett unwahrscheinlich erschien, kann das nach diesem Artikel hoffentlich jeder selbstbewusst sagen. Denn mit der richtigen Mischung aus Relevanz, Form und Medium ist es möglich, Dokumentationen zu erstellen, die langfristig einen Mehrwert bieten und durch Tools wie jQAssistant gleichzeitig helfen, den gefürchteten Documentation-Code-Gap erst gar nicht entstehen zu lassen. Ganz im Gegenteil wird damit die Dokumentation wieder ein essenzieller Aspekt der Softwareentwicklung und hilft, die Qualität und Architekturkonformität der Umsetzung stark zu verbessern.

Damit schließt dieser Artikel die Serie rund um jQAssistant ab, in der das Tool und dessen Funktionsweise vorgestellt (Teil 1) sowie im Rahmen von Software Analytics zum Aufbau von Wissen über Softwaresysteme eingesetzt wurde (Teil 2). Mit der Erstellung lebendiger Softwaredokumentation und der Absicherung der Architektur positioniert sich jQAssistant als flexibles und leistungsstarkes Tool, um den Entwicklerallag auch in Architektur- und Qualitätsfragen zu versüßen.

Stephan Pirnbaum ist Consultant bei der BUSCHMAIS GbR. Er beschäftigt sich leidenschaftlich gern mit der Analyse und strukturellen Verbesserung von Softwaresystemen im Java-Umfeld. In Vorträgen und Workshops präsentiert er seine gesammelten Erfahrungen und genutzten Methodiken.

Links & Literatur

[1] https://www.arc42.de

[2] https://www.writethedocs.org/guide/docs-as-code/

[3] https://doctoolchain.org

[4] https://swagger.io

[5] https://c4model.com

[6] https://contextmapper.org/docs/home/

[7] https://github.com/plantuml-stdlib/C4-PlantUML

[8] https://github.com/buschmais/The-Perfect-Greenfield

[9] https://github.com/xmolecules/jmolecules

[10] https://github.com/jqassistant-contrib/jqassistant-c4-plugin