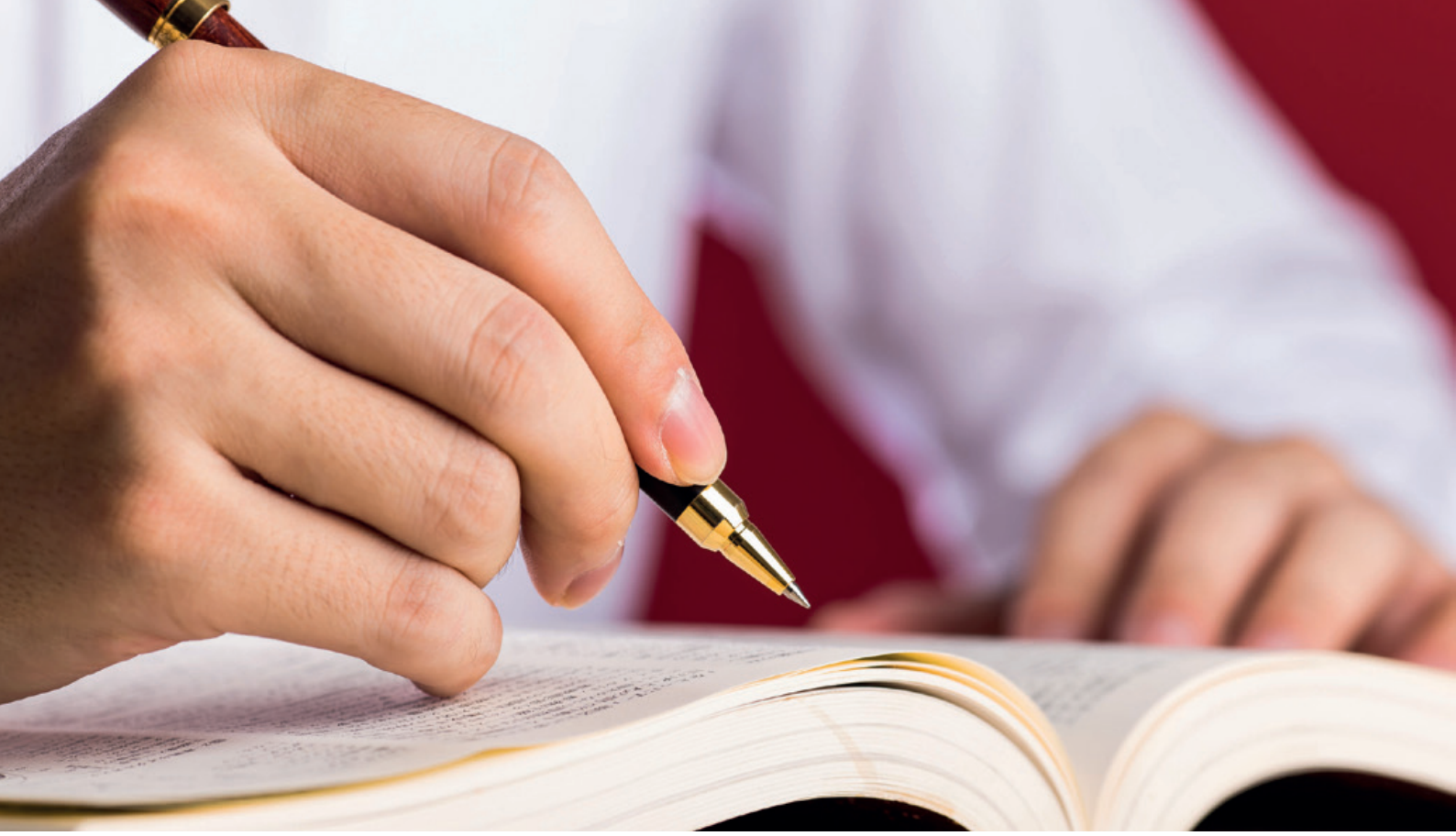

Continuous Documentation

Daniel Kocot, codecentric AG

Wir leben in einem Software-Entwicklungszeitalter der kontinuierlichen Prozesse. Funktionierende Software wird im Regelfall aus einem kontinuierlichen Prozess gewonnen. Der Artikel zeigt, wie man eine solche funktionierende Software dokumentiert.

Ein kontinuierlicher Entwicklungsprozess basiert auf den Phasen „Develop", „Build", „Test", „Deploy" und „Release". Sie stellen das Grundmodell von Continuous Delivery (CD) dar *(siehe Abbildung 1)*. Im Rahmen von CD findet auch immer der reine Development-Prozess nach agiler Art und Weise statt. Somit kommt als Startpunkt das Agile Manifest (AM) zum Tragen. Wenn man nun das AM bezogen auf die Dokumentation von Software betrachtet, findet sich dort der Satz „Working software over comprehensive documentation." Je nach Lesart kann der Eindruck entstehen, dass Dokumentation nur ein notwendiges Übel ist und daher vernachlässigt werden kann. Es geht jedoch vielmehr um den Umfang und den jeweiligen Adressatenkreis der Dokumentation.

Dokumentation im agilen Produktkonzept

Wenn man Dokumentation als Teil des agilen Entwicklungsprozesses betrachtet, ist es sinnvoll, Dokumente inkrementell zu erstellen. Mit jedem Update können Inhalte ergänzt oder aktualisiert werden. Idealerweise sind dabei die Zeiträume für einzelne Inkremente allerdings länger als bei der Software-Entwicklung.

Die Projektplanung gewinnt an Transparenz, wenn die Erstellung von Dokumenten darin als Arbeitspaket einfließt wie andere Tätigkeiten im Projekt auch. Damit lassen sich für Dokumentationsaufgaben auch Ressourcen einplanen und Deadlines definieren. Sinnvollerweise definiert ein Projekt einen Dokumentations-Verantwortlichen, der proaktiv dokumentationsrelevante Themen aufgreift und als Ansprechpartner für alle Frage der Dokumentation zur Verfügung steht.

Bei der Entwicklung von Software ist Anforderungsdokumentation nötig und wünschenswert, indem sie den Entwicklern bei der Umsetzung der Anforderungen tatsächlich hilft. Dabei müssen Anforderungen erst zu dem Zeitpunkt beschrieben sein, zu dem mit der Implementierung der gewünschten Funktionalität begonnen wird. Die Anforderungsdokumentation wird im Laufe des Projekts sukzessive weiterentwickelt. Neben der reinen Dokumentation für Entwickler wird auch Dokumentation nötig, die einen Überblick über ein Projekt in all seinen Facetten gibt; dies ist für viele Projekt-Beteiligte über einen langen Zeitraum hinweg nützlich.

Die Dokumentation gewinnt immens an Nutzen, wenn sie nicht nur die gewählten Konzepte beschreibt, sondern auch die Alternativen,

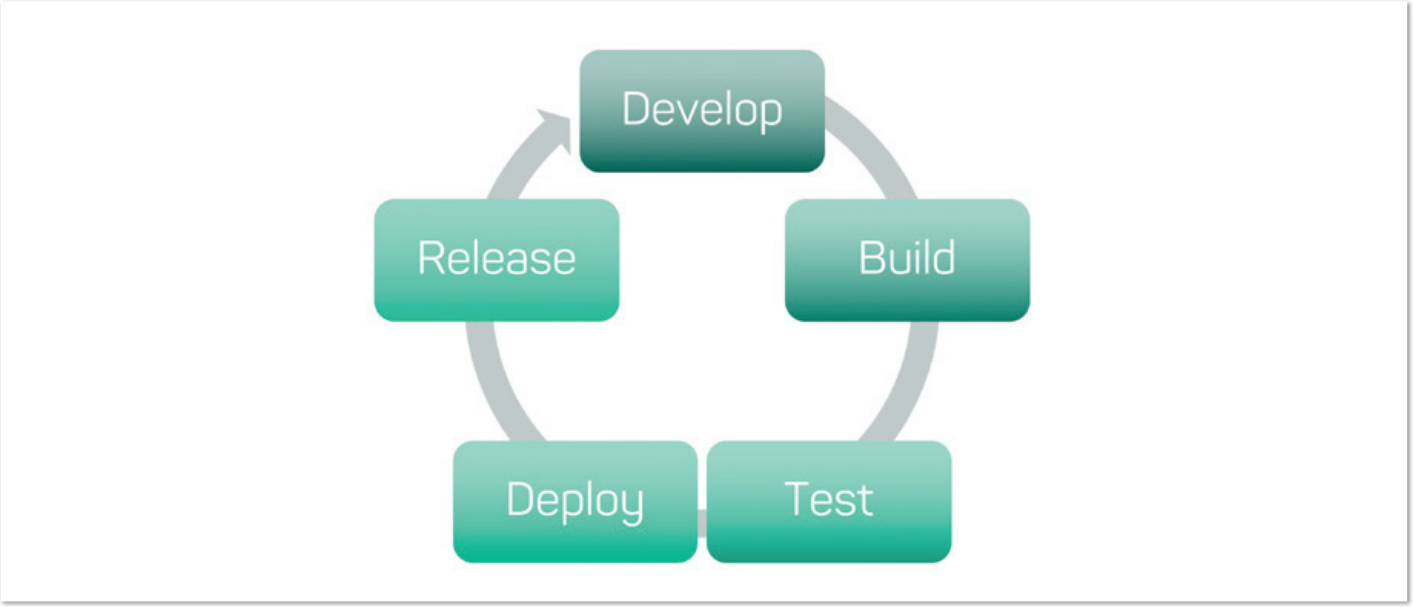

Abbildung 1: Der Continuous-Delivery-Lifecycle

die betrachtet worden sind, sowie die Gründe, die für die Entscheidungen ausschlaggebend waren. Nutzungsanleitungen sind am besten verständlich, wenn sie in Form eines Drehbuchs bereitgestellt werden. Dokumente sind sehr viel zugänglicher für den Leser, wenn sie mit konkreten Beispielen arbeiten. Jedes einzelne Dokument profitiert von einer klaren und übersichtlichen Struktur, die auch ein gewisses Maß an Querlesen erlaubt. Jedes Projekt-Dokument sollte mit Richtlinien für die Leser beginnen, die in kurzer und prägnanter Form klarmachen, für wen und in welcher Situation das Dokument geeignet ist.

Der punktuelle Einsatz von Diagrammen trägt spürbar zur Verständlichkeit der Materie bei. Ein häufiger Einsatz von Tabellen ermöglicht die systematische und übersichtliche Präsentation von Informationen. In Online-Dokumenten sollte von Hyperlinks Gebrauch gemacht werden, die auf andere Dokumente zu verwandten Themen verweisen beziehungsweise auf Abschnitte, Textpassagen oder Abbildungen darin. Auch die Dokumentation in IT-Projekten profitiert von einem ansprechenden und leserfreundlichen Layout. Idealerweise stehen für unterschiedliche Dokumenttypen geeignete Templates zur Verfügung.

Dokumente können sehr viel Nutzen entfalten, wenn die Autoren damit aktiv auf interessierte Leser zugehen. Im besten Fall entsteht im Projekt eine Dokumentationslandschaft, also ein navigierbarer Raum, in dem die unterschiedlichen Projekt-Dokumente in übersichtlicher Form abgelegt sind. Es ist sinnvoll, Retrospektiven auch dafür zu nutzen, Erkenntnisse über die Dokumentation zu gewinnen und diese Erkenntnisse in Best Practices einfließen zu lassen, die für zukünftige Projekte genutzt werden. Ein organisationsweites Wissensmanagement profitiert davon, wenn Projekt-Retrospektiven auch zu dem Zweck genutzt werden, projektübergreifende Erkenntnisse zu sammeln sowie Ideen und Konzepte zu identifizieren, die über den aktuellen Projekt-Kontext hinaus von Bedeutung sind.

Tools

Nachdem das Thema „Continuous Documentation" zuerst von der theoretischen Seite betrachtet wurde, nun der Blick auf Tools, die die Philosophie von Documentation as Code (Doc-as-Code) entsprechend unterstützen können. Doc-as-Code geht grundsätzlich davon aus, dass Dokumentation nahezu mit denselben Tools und Workflows erstellt wird wie der Sourcecode der Entwicklung. Der Einsatz von Git als Versionskontrolle und Gradle als Build-Management wird als gegeben vorausgesetzt.

Um Dokumentation nah an der Software zu schreiben, wird Plain Text Markup verwendet. Im Folgenden wird hierfür Asciidoc(tor) eingesetzt, wobei Asciidoc die reine Markup-Sprache darstellt und Asciidoctor für die Konvertierung in verschiedene Endformate benutzt wird. Ein großer Vorteil von Asciidoc und Plain Text Markup im Allgemeinen ist, dass die Dokumente schon mit einfachem Editor vor jeglicher Konvertierung les- und durchsuchbar sind. In Verbindung mit einer Versionskontrolle ist die Dokumentation genauso versionierbar wie der reine Entwicklungscode *(siehe Abbildung 2)*.

Die Dokumentation von Software beinhaltet im Regelfall auch eine Vielzahl von Grafiken und Diagrammen. Um diese nun in einen Workflow mit Asciidoc(tor) zu integrieren, soll PlantUML verwendet werden. Es bietet unterstützt durch Plain Text Markup die Möglichkeit, UML-Diagramme und weitere zu erstellen. Für Asciidoctor ist eine Erweiterung (Asciidoctor Diagram) verfügbar, die es dem Anwender ermöglicht, PlantUML-Notation direkt in das Asciidoc-Dokument zu integrieren *(siehe Abbildung 3)*.

Wie bereits zu Beginn erwähnt, ist es für die Entwicklung wichtig, dass die Anforderungen an die Software entsprechend dokumentiert werden. Wie kann der Entwickler auf direkte Änderungen der Anforderung innerhalb eines Sprints reagieren? Lassen sich Anforderungen auch durch Product Owner ohne Entwicklungshintergrund als „as Code" abbilden? Um das beschriebene Ziel zu erreichen, kommt Gherkin, eine sogenannte „Business Readable DSL", zum Einsatz. Gherkin ist eine zeilenorientierte Sprache, die Einrückungen für die Strukturierung benutzt *(siehe Listing 1)*.

Für das Testen der Anforderungen soll Spock verwendet werden. Spock ist in der JVM-Sprache Groovy geschrieben und nutzt eine

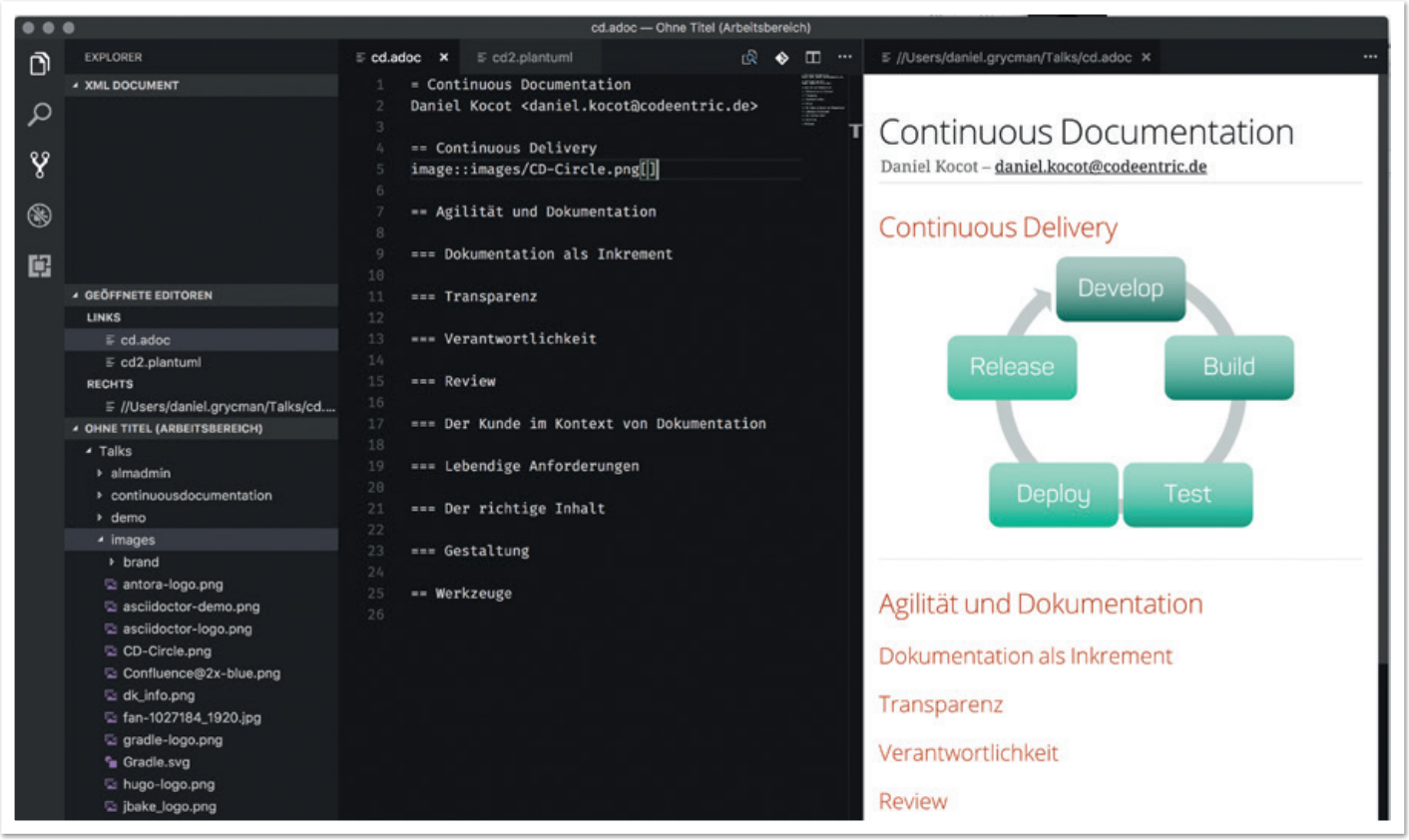

Abbildung 2: Asciidoc mit Preview im Visual Studio Code Editor

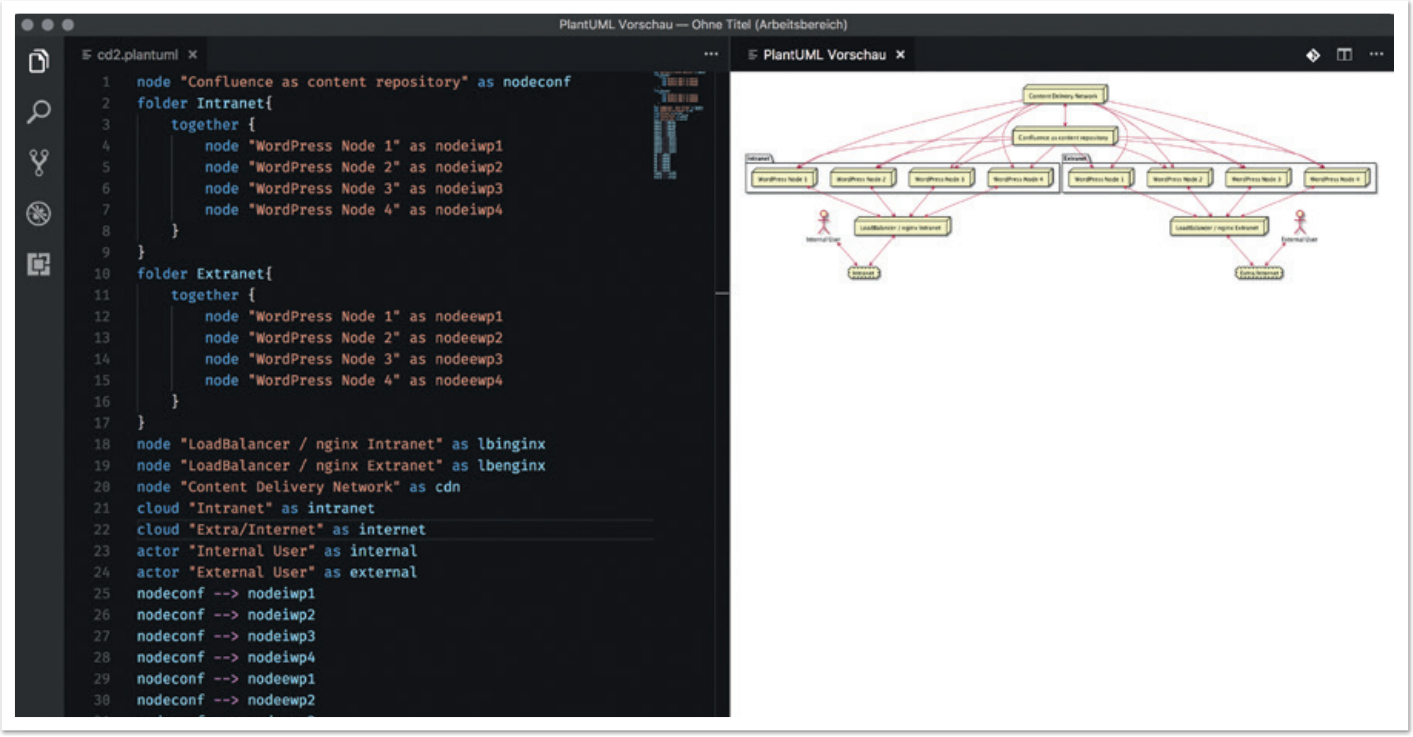

Abbildung 3: PlantUML mit Preview im Visual Studio Code Editor

```
Feature: Search
  Scenario: Simple Search
  Given a web browser is on the Google page
  When the search phrase "codecentric" is entered
  Then results for "codecentric" are shown
```

Listing 1

eigene Domain Specific Language (DSL) für die Beschreibung der Tests. Genau diese DSL fördert die Übersichtlichkeit und Lesbarkeit der Tests. Natürlich lässt sich auch reiner Java-Code mit Spock ausführen.

Eine weitere Besonderheit von Spock ist die Darstellung von fehlgeschlagenen Tests. Hiermit wird die fehlgeschlagene Bedingung bis in ihre Einzelkomponenten visualisiert. Mit einem selbst geschriebenen Gradle-Task können nun die einzelnen Gherkin-Specifications in Spock-Specifications transformiert werden. Für den Artikel umfasst der Gradle-Task nur die Transformation der einfachsten Gherkin-Struktur *(siehe Listing 2)*.

Das Groovy-Skript stellt eine erste Basisumsetzung der Gherkin-Spezifikation dar. Durch Gherkin können wir nun einen Workflow zwischen dem Produkt-Owner und den Entwicklern umsetzen, der

```
task createSpockSpecs {
  group = "Generating Spock Specs from gherkin"
  doLast {
     def gherkinFiles = fileTree("src/test/resources/Features").files
     def specBlock
     gherkinFiles.each {

        File transformFile = new File("$projectDir/src/test/groovy/$it.name".replaceFirst(~/\.[^\.]+$/,'Spec.groovy'))

        if (!transformFile.exists()) {
           transformFile.createNewFile()

           def completeScenarioString
           def completeGivenString
           def fileScenarioString
           def completeWhenString
           def completeThenString

           println "Generate Spock Spec"
           def multiline = it.text
           def list = multiline.readLines()
           list.removeAll { it.startsWith("Feature:") }
           list.find {
              if (it.contains("Scenario")) {
                 def scenarioString = it.toString().replace("Scenario: ", "def ")
                 fileScenarioString = scenarioString.substring(4)
                 completeScenarioString = scenarioString + "() {"
              }
              if (it.contains("Given")) {
                 String givenString = it.toString().replace("Given", "given: \"")
                 completeGivenString = givenString + "\""
              }
              if (it.contains("When")) {
                 String whenString = it.toString().replace("When", "when: \"")
                 completeWhenString = whenString + "\""
              }
              if (it.contains("Then")) {
                 def thenString = it.toString().replace("Then", "then: \"")
                 completeThenString = thenString + "\"\n}"
              }
           }

           def minusContent = new StringBuilder(it.text)
           def fileContentSb = new StringBuilder(it.text)

           specBlock = fileContentSb.insert(0, "import spock.lang.Specification\n" +
                  "\n" +
                  "\n" +
                  "class " + it.name.take(it.name.lastIndexOf('.')) + "Spec extends Specification {\n"
                  + completeScenarioString + "\n"
                  + completeGivenString + "\n"
                  + completeWhenString + "\n"
                  + completeThenString + "\n}").minus(minusContent)

           def spockFiles = fileTree("src/test/groovy").files

           spockFiles.each {
              println it.name
              println it.name.take(it.name.lastIndexOf('.'))
              if (specBlock.contains(it.name.take(it.name.lastIndexOf('.')))) {
                 it.setText(specBlock)
              }
           }
        }
     }
  }
}
```

Listing 2

```
com.athaydes.spockframework.report.IReportCreator=com.athaydes.spockframework.report.template.TemplateReportCreator
com.athaydes.spockframework.report.template.TemplateReportCreator.specTemplateFile=/template/spec-template.ad
com.athaydes.spockframework.report.template.TemplateReportCreator.reportFileExtension=ad
com.athaydes.spockframework.report.template.TemplateReportCreator.summaryTemplateFile=/template/summary-template.ad
com.athaydes.spockframework.report.template.TemplateReportCreator.summaryFileName=index.ad
com.athaydes.spockframework.report.outputDir=src/documentation/content/test-reports
com.athaydes.spockframework.report.hideEmptyBlocks=false
```

Listing 3

```
plugins {
   id 'org.jbake.site' version '1.0.0'
}
```

Listing 4

```
jbake {
   srcDirName = 'src/documentation'
   destDirName = 'documentation'
}
```

Listing 5

```
= Project documentation
Daniel Kocot
2017-10-03
:jbake-type: page
:jbake-tags: documentation, manual
:jbake-status: published
```

Listing 6

zu weniger Iterationen zwischen einer Anforderung und deren Umsetzung innerhalb eines Sprints beitragen kann.

Für die Darstellung der transformierten Anforderungen wird Spock Reports verwenden. Damit haben wir die Möglichkeit, die Ergebnisse des Spock-Tests mithilfe von Templates in Asciidoc darzustellen. Um diese Templates anpassen zu können, müssen wir eine Properties-Datei unter „test/resources" anlegen *(siehe Listing 3)*. Nun wird bei jedem Build der Software die Dokumentation um die getesteten Anforderungen erweitert.

Mit dem wachsenden Bedarf von APIs stellt sich auch die Frage, wie man deren Dokumentation in einen kontinuierlichen Prozess einbeziehen kann. Hier zeichnen sich nun zwei alternative Wege ab, „contract-first" und „code-first". Bei „contract-first" wird die Dokumentation des Rest-API aus der Swagger-YAML erstellt. „Code-first" hingegen erzeugt die entsprechende Dokumentation Test-getrieben. Die bei beiden Vorgehen erstellten Asciidoc-Dateien lassen sich dann auch wieder in den schon vorhandenen kontinuierlichen Prozess einbinden.

Bisher haben wir die Dokumentation als lokale Generierung betrachtet. Im Continuous Delivery kommen jedoch grundsätzlich sogenannte „Build-Server" wie Jenkins zum Einsatz. Hier werden die Software-Artefakte entsprechend den jeweiligen Quellen gebaut und im Anschluss verteilt. Neben der reinen Verteilung der Artefakte kann über den Build-Server eine separate Bereitstellung der Dokumentation gewährleistet werden.

Bislang haben wir die mögliche Generierung von Inhalten und deren Bereitstellung mithilfe von Tools angeschaut. Die Dokumentation sollte Multi-Channel-basiert aufbereitet sein, um diese dann gemäß dem Single-Source-Publishing-Prinzip zu verbreiten. Genau diese Vorgehensweise haben wir durch den Einsatz der weiter oben vorgestellten Werkzeuge erreicht. Nun gilt es einen Blick auf die verschiedenen Ausgabekanäle zu werfen.

Den wohl wichtigsten Kanal zur Bereitstellung von Informationen innerhalb von Software-Entwicklungsprozessen stellen Wikis dar. Beispielhaft seien hier Confluence und XWiki genannt, die beide über ein Rest-API verfügen. Darüber lassen sich innerhalb des Deployment-Prozesses Elemente zu einer Dokumentation hinzufügen oder aktualisieren.

Die Static Site Generators (SSGs) sind der zweite Kanal, den wir uns etwas genauer anschauen wollen. Zuerst wollen wir allerdings klären, was SSGs eigentlich sind. Damit lassen sich statische Webseiten auf der Grundlage von Plain Text Markup erstellen, wie in unserem Falle Asciidoc. Die SSGs werden für verschiedenste Programmiersprachen angeboten. Da wir uns im Java-Umfeld bewegen, soll unser Schwerpunkt auf JBake liegen. JBake lässt sich zum einem über die Kommandozeile und zum anderem auch über Build-Management steuern und ausführen. Durch die Benutzung von Gradle als Build Management Tool muss die „gradle.build"-Datei um das „jbake gradle"-Plug-in erweitert werden *(siehe Listing 4)*.

Standardmäßig erwartet das Gradle-Plug-in die Dokumentationsquellen unter „src/jbake". Mit Parametern im Build-File lässt sich der Quell- und Zielort jedoch anpassen, wobei das Ziel immer im Build-Ordner liegt. In unserem Fall mit den Werten in *Listing 5*.

Innerhalb des Asciidoctor-Dokuments können JBake-Metainformationen hinterlegt sein wie Seitentyp, Tags und der Status der jeweiligen Seite. Letzterer regelt die Sichtbarkeit der Seiten nach dem „Backprozess". Wem das Standard-Template nicht gefällt, kann dieses mithilfe der Template-Sprachen Freemaker, Groovy Simple, Groovy Markup, Thymeleaf und Jade seinen Wünschen nach anpassen *(siehe Listing 6)*.

Somit sind die Grundlagen für die Benutzung von JBake gelegt. Es soll nicht unerwähnt bleiben, dass Dan Allen, Asciidoctor-Lead, mit Antora einen neuen Ansatz in dem Bereich „Static Site Generation" veröffentlicht hat. Damit ist es möglich, anhand von sogenannten „Playbooks" Dokumentation auf Basis von verschiedensten Git-Repositories zu erstellen und somit ein noch umfassenderes Verständnis von Projektdokumentation und deren Erstellung zu schaffen.

Wir wollen uns noch einen weiteren und zugleich letzten Ausgabekanal anschauen, das Portable Document Format (PDF). Dieses

Format begleitet uns schon seit Anfang der 1990er Jahre und ist ein sogenanntes „plattformunabhängiges Dateiformat". Es stellt sich nun die Frage, wie wir aus der erstellten Asciidoc-Datei zu einem PDF-Dokument kommen. Genau für dieses Vorhaben existiert eine Asciidoctor-Erweiterung (Asciidoctor-PDF). Über diese lässt sich im gesamten Prozess, sowohl lokal als auch über den Build-Server, ein PDF-Dokument erstellen. Wichtig ist es, hierbei anzumerken, dass es sich bei dem PDF dann um einen nicht veränderbaren Snapshot der jeweiligen Dokumentation handelt.

Fazit

Dieser Artikel zeigt, dass es möglich ist, eine Dokumentation von kontinuierlichen Prozessen umsetzen. Hierzu wurde mit verschiedensten Tools eine Continuous-Delivery-Pipeline erstellt. Die aufgezeigten Möglichkeiten sollen aber nicht als alleinige Lösung für die Herausforderung der Dokumentation von Software verstanden werden. Es sollen vielmehr Denkanstöße geliefert werden, sich mit dem Thema auseinanderzusetzen.

Daniel Kocot
daniel.kocot@codecentric.de

Daniel Kocot ist seit Oktober 2016 Mitglied des Teams der codecentric am Standort in Solingen. Seit mehr als fünfzehn Jahre setzt er sich mit IT-Herausforderungen und deren Lösungen auseinander. Schwerpunktmäßig widmet er sich gerade den Themen „API Thinking" und „Legacy Modernization".