Ausgabe 04|2023
Deutschland € 15,90 Österreich € 16,90 Schweiz sfr 24,20
4 194164 115901 04

www.ITSpektrum.de

IT Spektrum

vormals OBJEKTspektrum

Digitaler Wandel & Software-Architektur für Profis

Navigieren im Brown Field:

Wie sich das Miteinander von alter und neuer IT effektiv gestalten lässt

Interview mit Alexander Neumann

Alexander wird ab Juli der neue Chief Content Officer bei SIGS DATACOM. Er erzählt, wie er in seiner aktuellen Rolle als Head of Events bei Heise zum Corona Gewinner werden konnte und über aktuelle IT-Themen wie Fortschritte bei der KI und Green-IT.

Zero Trust:
Vertraue niemanden und verifiziere jeden

Netzwerk-Security
Kubernetes – aber sicher!

Architektur-Porträt
Der Instant-Messenger Threema

Navigieren im Brown Field: Wie sich das Miteinander von alter und neuer IT effektiv gestalten lässt

G 6540F

Dokumentation wie Code behandeln

Auszeichnungssprachen in der Softwaredokumentation

Der großen Bedeutung von Softwaredokumentation kann heute mit zeitgemäßeren Umgebungen als Word oder Confluence begegnet werden. Dabei spielen Auszeichnungssprachen eine zentrale Rolle. Dieser Artikel zeigt auf, was vereinfachte Auszeichnungssprachen genau sind, welche Vorteile sie bieten und wie Unternehmen aus der Vielzahl der heute zur Verfügung stehenden Sprachen eine möglichst universell einsetzbare finden. Mit ihr ergeben sich Möglichkeiten, die jedes Unternehmenswiki in den Schatten stellen.

Dokumente in vereinfachten Auszeichnungssprachen lassen sich mit jedem beliebigen Texteditor erstellen und bearbeiten. Ihre schlichte Syntax ist für Mensch und Maschine gleichermaßen gut lesbar. Die Rohdokumente lassen sich wie Programmcode unter Versionsverwaltung stellen, und eine Build-Pipeline kann dann die formatierten Dokumente generieren, verlinken, exportieren und deployen. Ein solches Vorgehen wird gemeinhin als *Docs as Code* bezeichnet: Dokumentations-„Code" wird genauso behandelt wie Programmcode.

Dieser Artikel soll die Grundlagen vor allem für den ersten Schritt, der Auswahl einer passenden vereinfachten Auszeichnungssprache, beleuchten. Unsere gesuchte Sprache soll sich für den Großteil der Softwaredokumentationsformen eignen und auch von jeder sich mit IT beschäftigenden Person anwendbar sein – also nicht nur für Softwareentwickler selbst, sondern auch für Softwarearchitekten, Projektleiter, Produktverantwortliche und alle anderen Entscheider.

Softwaredokumentation umfasst in Anlehnung an [LudLic13] zuerst einmal alle Arten von *Systemdokumentation*. Diese beinhaltet zum Beispiel Spezifikationen, Architekturbeschreibungen, Anleitungen, aber auch APIs und als *letzte* Rückfallebene den eigentlichen Programmcode. Von den letzten beiden Punkten wollen wir (vor allem im Java-Bereich) vorerst einmal absehen, denn Programmcode bleibt Programmcode, und APIs werden in der Java-Welt typischerweise mittels Javadoc dokumentiert. In Python hingegen lässt sich eine API theoretisch in jeder Markup-Sprache dokumentieren, aber inwieweit es sinnvoll ist, von den etablierten Quasi-Standards abzuweichen, muss jeder für sich selbst entscheiden.

Der Bereich der Softwaredokumentation umfasst aber auch *Projektdokumentation*, also alles zur Projektplanung, sowie *Qualitätsdokumentation*, zum Beispiel Testberichte. Letztere können manuell oder automatisiert erstellt werden.

Zusammengefasst möchten wir in diesem Artikel unter Softwaredokumentation all das verstehen, was in Unternehmen typischerweise in separaten Dokumenten (oft Word-Dateien) oder auf Unternehmenswiki-Seiten (oft Confluence, aber auch BlueSpice oder andere Wikis) dokumentiert wird – beziehungsweise dokumentiert werden sollte …

Gründe für Softwaredokumentation

Die Gründe für Dokumentation sind jeder in der Softwareentwicklung tätigen Person eigentlich bekannt. Schlechte oder gar fehlende Softwaredokumentation ist aber nicht nur gefühlt, sondern auch in der Literatur belegt ein ewiges Sorgenkind (stellvertretend für viele: [LudLic13]). Sie wird oft als überflüssig angesehen oder – wenn überhaupt – erst am Ende einer Entwicklungsphase erstellt. Nicht selten geht dies mit dem Gefühl einer lästigen Pflichtübung einher, „damit diese Formalität auch erledigt ist".

Ein kleiner Trost vorab: Vor ein, zwei, drei Jahrzehnten waren das Wissen und die Möglichkeiten betreffend effektive Softwaredokumentation noch ganz andere als heute. Überlange Word-Dokumente, irgendwo auf SharePoint mit Dateisuffixen wie `-neu.doc`, `-aktuell.doc`, `-final.docx`, `-jetzt-aber-wirklich-final-final.docx` abgelegt, haben auch beim motiviertesten Dokumentierer das letzte Flämmchen Spaß an dieser Arbeit ersticken lassen.

Heute haben sich allgegenwärtige Vernetzung, Webbrowser mit hervorragender Funktionalität und Darstellung, Wikis, integrierte Entwicklungsumgebungen (IDEs) und Versionsverwaltungen (meist Git) etabliert. Ein Teil der IT-Branche hat die Wichtigkeit des Themas Dokumentation erkannt und entsprechende Fachbücher (zum Beispiel [Zoe22] als guter Einstieg) und Vorlagen für immer wiederkehrende Fragestellungen hervorgebracht (zum Beispiel das *C4-Modell* oder *arc42*). Und natürlich leistet der Einsatz einfacher und funktionaler Auszeichnungssprachen – die es natürlich auch noch nicht seit Ewigkeiten gibt – große Dienste. Genau davon handelt vorliegender Artikel.

In einem JavaSPEKTRUM-Artikel [Hei22] zum Thema Javadoc habe ich mich intensiv mit der Notwendigkeit von (quellcodenaher) Dokumentation befasst. Ich werde diese Argumentation hier nicht wiederholen, sondern stattdessen nur kurz meine persönliche Top 3 wiedergeben:

Reverse Engineering
Quellcode wird viel öfter gelesen als geschrieben. Gedankliches „Dekompilieren“ ist bei unkommentierten Klassen schon schlimm, bei ganzen Softwarearchitekturen geradezu unverantwortlich. Derartige Sisyphusarbeit ist nicht nur hochgradig ermüdend, sondern auch sehr zeitintensiv, fehleranfällig und damit teuer. Wer zu Zeiten des Fachkräftemangels seine Entwickler wie Aschenputtel Erbsen suchen lässt, darf sich über fehlende Mitarbeiter und Effizienz nicht wundern.

Agilität
„Agil“ heißt *nicht* „keine Dokumentation“. Es war und ist nirgendwo die Rede davon, dass mit agilen Methoden wie Scrum keine Softwaredokumentation mehr erstellt werden muss. Das *Agile Manifest* besagt lediglich, dass (im Zweifel) „working software over comprehensive documentation“ der Vorzug zu geben ist (man beachte das Wort „comprehensive“). Gleichzeitig schreibt es aber auch: „[…] while there is value in the items on the right, we value the items on the left more.“ [Manifesto]
Es ist auch nützlich, dies unter einem historischen Aspekt zu betrachten. Früher – und das war die eigentliche Motivation für das agile Manifest – wurden überbordende Spezifikationen geschrieben, bevor auch nur eine einzige Zeile codiert oder der Kunde für eine erste Vorschau mit ins Boot geholt wurde. Nicht selten kam bei solchen klassischen Wasserfallprojekten später dann das böse Erwachen. Das Agile Manifest soll die Softwareverantwortlichen deshalb zurückbesinnen: Verfahre dich nicht vorzeitig an exzessiver Spezifikation, sondern schau, dass zuerst mal ein Prototyp läuft.
Gegen das ist auch überhaupt nichts einzuwenden. Aber weder ersetzt das die Notwendigkeit von Dokumentation (im Idealfall während der Entwicklungsphase erstellt), noch rechtfertigt es, aus diesen agilen Grundsätzen gleich eine Religion zu machen.

Unternehmenswert
Auch im Hinblick auf den monetären Wert einer Software – und damit deren Inhaberfirma – ist eine „ausreichende und für einen sachverständigen Dritten nachvollziehbare Dokumentation“ unabdingbar. Spätestens wenn eine Softwarefirma (oder deren Produkt) eines Tages verkauft werden soll, wird sich ein Gutachter im Rahmen einer Due Diligence die ganze Sache ansehen. „Nur wenn entsprechende Unterlagen die Weitergabe der Kenntnisse auch ohne Mitwirkung der Mitarbeiter ermöglichen, ist davon auszugehen, dass das Unternehmen über die Technologie verfügen kann“.
Diese beiden Zitate stammen nicht von irgendwo, sondern aus den *Grundsätzen zur Bewertung immaterieller Vermögenswerte* vom *Institut der Wirtschaftsprüfer (IDW)* [IDWS5]. Unsachgemäße und nicht verständliche Dokumentation gilt also nicht nur betriebswirtschaftlich, sondern auch rechtlich als typischer Mangel (mehr zum Ganzen: [Dem23]).

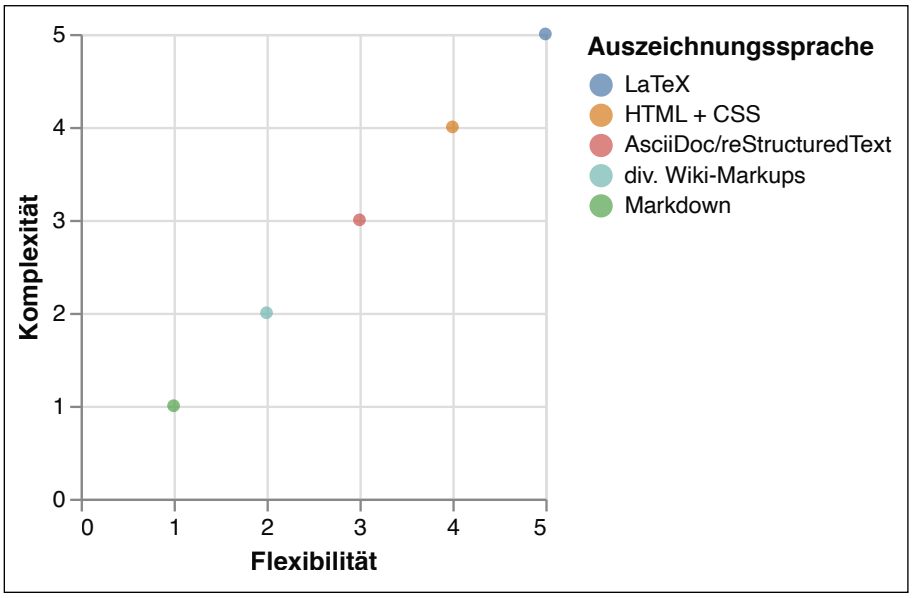

Abb. 1: Flexibilität vs. Komplexität ausgewählter Auszeichnungssprachen

Liest sich das alles sehr dramatisch? Richtig, ist es auch.

Was zeichnet (vereinfachte) Auszeichnungssprachen aus?

Eine Auszeichnungssprache (engl. *markup language*) ist eine maschinen- und meist auch menschenlesbare Sprache für die Gliederung und Formatierung von Texten und anderen Daten. Der bekannteste Vertreter ist die *Hypertext Markup Language (HTML)* [WikiAuszSpr]. Dokumente in Auszeichnungssprachen können manuell von Menschen geschrieben oder automatisch generiert werden.
Typische (normale) Auszeichnungssprachen im Hinblick auf die Erzeugung formatierter Dokumente sind zum Beispiel HTML (in Kombination mit CSS) oder LaTeX. Dass sich mit HTML und CSS hervorragend gestalten lässt, bedarf keiner weiteren Erläuterungen. LaTeX ist eine Sprache zum Setzen von Texten und erfüllt auch die höchsten typografischen Ansprüche. Der ein oder andere Leser wird sich vielleicht noch an sein Hochschulstudium erinnern, wo es vor allem in mathematischen und wissenschaftlichen Arbeiten sehr verbreitet ist.
So genial die Funktionalität dieser Sprachen auch ist, das manuelle Schreiben ihrer Befehle kostet viel Zeit und ist somit für unsere eingangs erwähnten Zwecke nicht „massentauglich“. Den Rohling meiner Masterarbeit habe ich dazumal in HTML geschrieben, und vor über 12 Jahren habe ich begonnen, mich systematisch mit LaTeX zu beschäftigen. Damit konnte ich in meiner didaktischen Ausbildung die schönsten Arbeiten, im Unterricht die schönsten Aufgabenblätter und seit meiner Firmengründung die schönsten Briefbögen, Vertragsdokumente und Präsentationsfolien gestalten. Wenn ich aber heute zurückblicke, darf ich gar nicht aufrechnen, wie viel Zeit mich das alles gekostet hat, weil die Sprache recht bockig und inkonsistent sein kann und es immer wieder etwas anzupassen gab – selbst bei einfachsten Dingen wie Tabellen. Ich kann also jeden verstehen, der sich damit nicht (mehr) herumärgern will. Nun ja, dafür habe ich jetzt schönes Briefpapier …
In **Abbildung 1** habe ich einmal (ganz unwissenschaftlich und ohne Referenzen) typische Vertreter von Auszeichnungssprachen aufgeführt. Als Grundsatz wage ich die These, dass eine Sprache komplizierter ist, je mehr (gestalterische) Flexibilität sie bietet.
Eine *vereinfachte Auszeichnungssprache* (engl. *lightweight markup language*) setzt an diesem Punkt an und stellt eine Syntax zur Verfügung, die sich einfach erlernen, schreiben und ohne nennenswerte Störung des Leseflusses auch lesen lässt. So kann zum Beispiel ein == am Zeilenanfang für eine Überschrift stehen, und Aufzählungslisten werden einfach mit einem vorangestellten * pro Aufzählungspunkt

erzeugt. Ein YouTube-Video lässt sich mit einer einzigen Zeile einbinden:

```
video::dQw4w9WgXcQ[youtube]
```

Wer mit Jira oder einem Unternehmenswiki arbeitet, wird mit solchen vereinfachten Auszeichnungssprachen wohl schon Bekanntschaft gemacht haben. Confluence stellt hier eine große Ausnahme dar, da es zwar Wiki-Markup und Markdown-Syntax als *Eingabe* akzeptiert, es dann aber in seinem internen Rich-Text-Format abspeichert, auf dessen Quellcode der Benutzer anschließend *keinen* Zugriff mehr hat [ConfWikiMarkup]. Damit gesellt sich Confluence in die Reihe reiner WYSIWYG-Textverarbeitungsprogramme wie Word und dessen Open-Source-Mitstreiter. Derartige proprietäre (Binär-)Formate sind allerdings indiskutabel (wobei ich in diesem Fall als „proprietär" auch das bezeichne, was nicht menschenlesbar ist, obwohl die Office-Formate theoretisch Open-Source-Formate wären). Das liegt zum einen natürlich am Vendor-Lock-in, von dem momentan eine ganz große Menge Unternehmen betroffen ist, nachdem Atlassian (der Hersteller von Confluence) angekündigt hat, On-Site-Lizenzen künftig nur noch für horrende fünfstellige Summen pro Jahr zu verkaufen und alle anderen damit in ihre (auch unter US-Recht stehende) Cloud zu verschieben [Los21]. Zum anderen erachte ich jedes Dateiformat als unzureichend, aus dem nicht menschenlesbar ersichtlich ist, was genau der Inhalt oder die Einstellungen sind. Wer schon einmal mit kaputten Tabellen, Aufzählungslisten oder inkonsistenter Formatierung in Word zu tun hatte, weiß, wovon ich spreche. Vor diesem Hintergrund wird es für alle wechselwilligen Confluence-Kunden erst recht eine große Herausforderung werden, ihre Dokumente sauber aus der Plattform zu extrahieren.
Generierte Texte in einer (vereinfachten) Auszeichnungssprache können via Skript noch immer problemlos in ein Unternehmenswiki exportiert werden. Eine Bearbeitung *im* Wiki hingegen ist nicht vorgesehen. Der Master muss immer das Auszeichnungssprachendokument selbst sein.

Populäre vereinfachte Auszeichnungssprachen und Auswahlkriterien

Nachdem wir jetzt wissen, was vereinfachte Auszeichnungssprachen sind und wo ihre Vorteile liegen, sehen wir uns ein paar typische Vertreter dieser Sprachen an. Auf der englischen Wikipedia finden sich eine Auflistung und ein Vergleich von aktuell 21 vereinfachten Auszeichnungssprachen [WikiLightMarkLang]. Viele davon existieren nur für ein bestimmtes Produkt oder einen bestimmten Anwendungsbereich, so zum Beispiel *BBCode* (für (BB-)Internet-Foren), *Jira Formatting Notation*, *Org-mode* (sehr Emacs-bezogen), *Slack*, *WhatsApp* sowie die Wiki-Dialekte *PmWiki* und *TiddlyWiki*. Andere wiederum sind veraltet oder führen nur (noch) ein Schattendasein, so zum Beispiel *Creole*, *Gemtext*, *POD*, *setext*, *Texy* und *txt2tags*.
Die Verbreitung und Popularität einer Markup-Sprache soll zwar nicht *das* Hauptargument, aber zumindest *ein* Argument für die Entscheidungsfindung sein. Es hilft nichts, wenn eine Sprache – selbst wenn sie technisch überzeugend wäre – in den letzten 15 Jahren nicht aus einer nerdigen einseitigen Website oder einem GitHub-Repository einer Einzelperson herausgefunden hat. Oder wie es in Open-Source-Kreisen heißen würde: Wenn sie keine Community hat. Wie wir gleich noch sehen werden, stellt das Aussortieren der ersten zwei Drittel der Sprachen kein Problem dar – wir werden auch so fündig.
In der Liste finden sich weiter vier Markdown-Dialekte: *Markdown*, *Markdown Extra*, *GitHub Flavored Markdown* und *MultiMarkdown*.

Markdown

Markdown ist ein zweischneidiges Schwert. Es ist nicht nur gefühlt, sondern auch objektiv bei meiner Recherche diverser IT-Zeitschriften der letzten 5 bis 15 Jahre mit großem Abstand die populärste vereinfachte Markup-Sprache (stellvertretend für viele: [Tre22]). Geschätzt wird seine sehr einfache, effiziente Syntax und Unterstützung durch viele Programme und Plattformen – allen voran GitHub. Problematisch hingegen ist, dass Markdown bis heute nicht standardisiert ist und so eine Vielzahl an Dialekten (engl. flavors) entstanden ist. Hinzu kommt, dass viele Sprachmerkmale fehlen und nur separat über Plug-ins oder eingebetteten HTML-Code realisiert werden können. Das zieht sich von elementaren Dingen, wie fehlenden Tabellen, Querverweisen und Fußnoten, bis hin zur fehlenden Einbettung von YouTube-Videos. Eine vereinfachte Auszeichnungssprache, die nicht portierbar ist und doch wieder auf HTML zurückgreifen muss, scheidet also für unsere Zwecke als universelle Softwaredokumentationssprache aus. Dieser Meinung schließen sich auch andere Stimmen an [Hol16].

AsciiDoc
MediaWiki
reStructuredText (RST)
Textile

Tabelle 1: Unsere Top 4 der vereinfachten Auszeichnungssprachen

Tabelle 1 listet die vier verbleibenden vereinfachten Auszeichnungssprachen in alphabetischer Reihenfolge auf.

Textile

Textile hat einen überschaubaren Funktionsumfang, der auf der offiziellen Website prägnant dokumentiert ist [Textile4]. Sowohl in der Praxis als auch in der Theorie (Literatur) ist mir Textile allerdings noch nie begegnet. Es besteht die Gefahr, dass für größere Anwendungsfelder letztendlich dann doch Funktionen fehlen, die vielleicht – oder vielleicht auch nicht – mit Plug-ins abgedeckt werden können. Dieses Risiko ist zu groß. Bereits jetzt ist ersichtlich, dass in der Standardausführung dateiübergreifende Includes wie auch ein mathematischer Formelsatz fehlen.

MediaWiki

MediaWiki wird in der weltberühmten Wikipedia wie auch in der Unternehmenswiki-Software BlueSpice eingesetzt. Eine verbindliche Spezifikation habe ich allerdings auch nach langer Suche nicht gefunden, vielmehr muss offenbar eine Aufzählung von Beispielen genügen [WikitextExamples]. Wie so vieles in der Wiki-Welt liegt das, wonach man sucht, hinter teilweise kryptisch anmutenden Verlinkungen verstreut herum. Der Versuch, MediaWiki-Markup formal zu spezifizieren, wurde 2010 abgebrochen [MediaWikiMarkSpec]. Problematisch sehe ich auch die Fixierung auf die Wiki-Plattform selbst. Wünschenswert wäre ein wirklich offlinetaugliches und plattformunabhängiges *Datei*-Format.

reStructuredText (RST)

RST hat seine Verbreitung in der Python-Welt. Was Javadoc für Java-Programmierer, ist RST für Pythonisten – allerdings mit einem entscheidenden Unterschied: Während Javadoc speziell auf die Programmiersprache zugeschnitten ist (was man vor allem an den vielen spezifischen Tags wie `@param`, `@return` oder `@throws` sieht), wird Python-Quellcode in sogenannten *Docstrings* dokumentiert. Ein Docstring ist prinzipiell nichts anderes als ein mehrzeiliger String, der sogar zur Laufzeit ausgelesen werden kann und in dessen „Gestaltung" (formell richtig: in

dessen semantischer Auszeichnung) der Verfasser prinzipiell frei ist [PEP257].
Es gibt also keine fixen Dokumentations-Tags wie in Javadoc. Das hat den Vorteil, dass die daraus generierten Dokumente (meist mit dem Generator *Sphinx*) nicht so monoton und streng wie Javadoc-APIs aussehen, sondern oft „kreativer", „lebendiger" und „bunter" daherkommen (Geschmacksproben siehe [pandRead Pickle] oder [TensFlowModel]). Als Nachteil erfordert das vom Autor allerdings deutlich mehr Verantwortung und Disziplin, um einem einheitlichen Stil nachzugehen und nichts zu vergessen. Mehr zum detailliert richtigen Einsatz von Javadoc finden Sie übrigens in meinem JavaSPEKTRUM-Artikel „Javadoc mit Stil" [Hei22].
Obwohl RST also eher im Python-Bereich angesiedelt ist, ist seine Syntax völlig Python-unabhängig. RST eignet sich darum prinzipiell vollumfänglich für Softwaredokumentationen aller Art. Sowohl die Sprache selbst als auch der Quasi-Standardgenerator Sphinx lässt in Sachen Funktionsumfang keine Wünsche offen. Für ein nicht so sehr programmiertechnisch verliebtes Publikum wirkt RST allerdings nicht sonderlich einladend. Das mag zum einen an den etwas kahlen, doch sehr formell wirkenden Websites liegen [RST, Sphinx], zum anderen an der teilweise etwas ungeschickten Syntax, die vor allem bei den Tabellen auffällt. Auch keine Freunde gewinnt RST am Online-Editor, der schon seit Monaten, wenn nicht gar seit Jahren, offline ist [ninjs].

AsciiDoc

AsciiDoc ist dafür die Eier legende Wollmilchsau der Dokumentationssprachen. Mit Includes, mathematischem Formelsatz in TeX-Notation und nativ anfühlender Einbettung von (auch textbeschriebenen) Diagrammen verschiedenster Art erfüllt der Funktionsumfang auch die letzten von uns gewünschten Kriterien. Die Syntax ist angenehm, die Dokumentation ansprechend und reicht von einer Quick Reference [AscDocSyntaxQuickRef] bis hin zu einer vollständigen, geradezu üppigen Language Documentation [AscDocLangDoc]. Der Standardgenerator für AsciiDoc heißt *Asciidoctor* und beherbergt prinzipiell die gesamte Dokumentation, die für uns als Anwender (Autoren) relevant ist. Die offizielle Website der „Sprache" AsciiDoc [AscDoc] ist für uns hingegen nicht interessant.
Wer die Auszeichnungssprache gleich einmal ausprobieren will, findet unter [ascdocLIVE] einen Live-Editor. Mit dem Chrome-Plug-in *Asciidoctor.js Live Preview* lassen sich AsciiDoc-Rohdateien fertig gerendert im Webbrowser anzeigen. Das IntelliJ-IDEA-Plug-in *AsciiDoc* erlaubt das einfache Betrachten und Editieren von AsciiDoc-Dokumenten. Andere Browser und IDEs bieten mit Sicherheit vergleichbare Plug-ins. Der geehrten Leserschaft stelle ich den hier vorliegenden Artikel unter [Art] auch im AsciiDoc-Format zur Verfügung, denn die Rohfassung dieses Artikels wurde – wie könnte es auch anders sein – ebenfalls in AsciiDoc geschrieben.

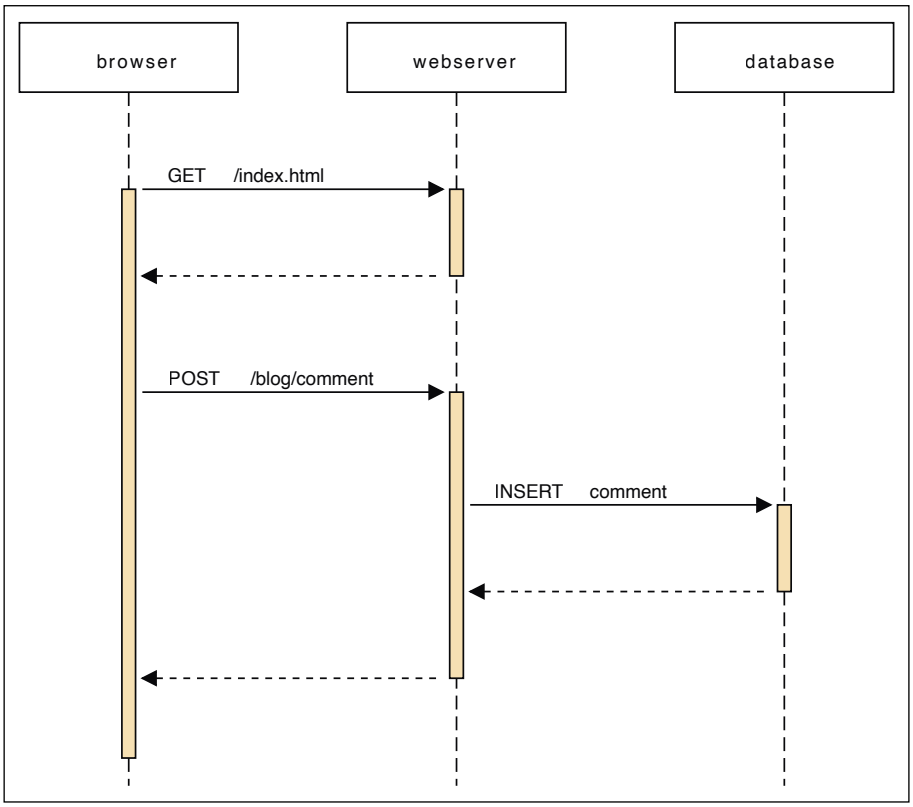

Abb. 2: Sequenzdiagramm als Beispiel eines Diagrams as Code (aus [Kro])

Organisatorisches und Politisches

Zugegeben, ein sauber in die Firmenstruktur eingebettetes (AsciiDoc-)Auszeichnungssprachen-Dokument lässt sich anfangs nicht so schnell und einfach wie eine Word-Datei oder eine Confluence-Seite erstellen. Die Sprachelemente müssen erst erlernt und eine Infrastruktur mit Dokumentationsgenerator, Versionsverwaltung und CI-/CD-Pipeline konfiguriert werden.
Für die *Sprachelemente* bietet sich eine Mitarbeiterschulung durch eine Fachperson an. Diese kann hausintern als Expertenrolle rekrutiert oder von extern beigezogen werden. In einem halben bis ganzen Tag Schulung – je nachdem, wie tief man gehen will, vor allem bei Diagrammen – sollten die entscheidenden Grundkonzepte vermittelt werden können. Nicht empfehlenswert ist meines Erachtens, jeden Mitarbeiter „auf eigene Faust" ein bisschen herumspielen zu lassen. Die eigenen Ansprüche der jeweiligen Personen sind bekanntlich sehr verschieden. Eine Schulung hingegen kann die Essenzen herausarbeiten, Wichtiges betonen und Unwichtiges weglassen, sodass nicht nur der Arbeits- und Zeitaufwand unterm Strich deutlich geringer ausfällt, sondern auch sichergestellt wird, dass alle Mitarbeiter auf dem gleichen Wissensstand sind.
Mit der Einrichtung und Konfiguration der *technischen Infrastruktur* sollten die Softwareentwickler betraut werden. Sie kennen sich am besten mit dem Versionsverwaltungssystem und der Build-Pipeline aus. Für alle Nicht-Entwickler, die sich (verständlicherweise) nicht mit Git auf der Kommandozeile herumschlagen möchten, gibt es einfach zu bedienende GUI-Clients, mit denen sie Dokumente ablegen und herausholen können – nicht komplizierter als ein FTP-Client. Das bedingt aber auch, dass sich die für die Dokumentationsinfrastruktur verantwortlichen Softwareentwickler gedanklich in die Nicht-Entwickler hineinversetzen können und diesen folglich sämtliche unnötige Komplexität vom Leibe halten.
Stefan Zörner schlägt in seinem Buch [Zoe22] gar eine Rolle namens „Doctator" vor. Diese Person verschafft und besitzt den Überblick über die Dokumen-

```
seqdiag {
  browser -> webserver [label = "GET /index.html"];
  browser <-- webserver;
  browser -> webserver [label = "POST /blog/comment"];
  webserver -> database [label = "INSERT comment"];
  webserver <-- database
  browser <-- webserver;
}
```

Listing 1: Code für Sequenzdiagramm aus Abbildung 2 (aus [Kro])

tation im Haus, kümmert sich sowohl um die organisatorischen als auch technischen Angelegenheiten des Dokumentationsprozesses, unterstützt die Teams beim Dokumentieren (schreibt die Dokus aber nicht zwingend selbst!) und überwacht die Aktualität und Konsistenz der Inhalte. Diese Idee kann ich nur unterstützen. Ich persönlich vertrete die Auffassung, dass die Arbeit des Dokumentierens auf ein paar Personen konzentriert sein sollte, die das a) gerne und b) gut machen. Nicht jeder sollte einfach Dokumente erstellen und bearbeiten dürfen – und erst recht nicht müssen. Dazu habe ich in meiner Karriere schon zu viele Fälle gesehen, in denen diese (im Kern gut gemeinte) „Wiki-Philosophie" letzten Endes doch Opfer aus einer Mischung von Verantwortungsdiffusion und gegensätzlichen Qualitätsansprüchen wurde. Ich weiß aber auch, dass die Meinungen zu diesem Thema weit auseinandergehen und letzten Endes politisch sind. Eine kleine, aber sehr gute tabellarische Gegenüberstellung der beiden Wissensmanagement-Philosophien „Knowledge Sharing" vs. „klassisches Knowledge Management" findet sich in [Heigl21].

Ich bin doch immer wieder erstaunt, wie wenig das Thema „Dokumentation" in Unternehmen *systematisiert* wird. Sofern überhaupt dokumentiert wird, wird im besten Falle improvisiert. Die Regelwerke für Versionierungsschemata, Spesenabrechnungen und E-Mail-Signaturen sind bis ins kleinste Detail ausgearbeitet, dem Thema Dokumentation – der entscheidenden Grundlage für neue Mitarbeiter, Weiterentwicklung der Softwarearchitektur und eines belastbaren Firmenwertes – begegnen Unternehmen hingegen oft nur mit Schulterzucken.

Der *GitLab Documentation Style Guide* (dessen konkreten Inhalt ich hier weder propagieren noch als gut oder schlecht bewerten will) zeigt exemplarisch auf, was es in Sachen Dokumentation prinzipiell zu regeln gäbe [GitLabDocStyleGuide]. Das reicht von groben Themen wie der „Single Source of Truth" über die Sprache (Deutsch oder Englisch? Und wenn Letzteres, *welches* Englisch?) bis hin zu Details wie der Groß- und Kleinschreibung von Überschriften.

Fazit: Die Reise geht weiter

Dieser Fachartikel könnte problemlos noch weitergehen. Er hat in (hoffentlich) objektiver und nachvollziehbarer Form die Vorteile vereinfachter Auszeichnungssprachen dargelegt und ebenso transparent das Feld der infrage kommenden Sprachen vorbereinigt. Ich kann aus eigener Erfahrung berichten, dass, je länger man sich mit der Thematik befasst, es mehr oder weniger auf einen Docs-as-Code-Ansatz mit – je nach Technologie und Vorliebe – AsciiDoc oder reStructuredText als Auszeichnungssprache hinausläuft.

Wie geht die Reise weiter? Mit der in diesem Artikel beschriebenen technischen und organisatorischen Grundlage, Dokumentationstexte unter Beibehaltung ihrer einfachen Lesbarkeit wie Quellcode zu behandeln, zu versionieren und zu deployen, ist bereits die erste große Stufe erreicht. Das Ökosystem mit *Asciidoctor* als Standardgenerator für AsciiDoc und *Sphinx* als Quasi-Standardgenerator für reStructuredText ist dabei überschaubar.

In einer nächsten Stufe können *Diagramme* nicht nur klassisch in (AsciiDoc-) Dokumente eingebunden, sondern darin sogar direkt in Textform beschrieben und generiert werden. Analog zu Docs as Code spricht man hierbei dann von *Diagrams as Code*. Das „Kästchenschieben" aus Grafikprogrammen hat also endlich ein Ende. Ein Bild sagt bekanntlich mehr als tausend Worte, weswegen ich den Leser direkt an [Kro] verweise, um sich eine Übersicht der zur Verfügung stehenden Diagrammtypen und deren oft einfache Beschreibungssyntax zu verschaffen.

Abbildung 2 zeigt exemplarisch ein Sequenzdiagramm, welches aus dem kurzen Code in **Listing 1** erzeugt wurde. Im Übrigen wurde der Plot aus **Abbildung 1** auch direkt aus Text erzeugt. Dessen Quellcode lässt sich bekanntlich unter [Art] einsehen.

Zur *Strukturierung* ganzer Architekturdokumentationen – und damit als weitere Evolutionsstufe – seien vor allem das eingangs kurz erwähnte *C4-Modell* und das *arc42-Template* erwähnt. Eine hervorragende prägnante Übersicht zu Letzterem findet sich im JavaSPEKTRUM-Artikel [Sta22] von Gernot Starke. Wie AsciiDoc für *Internationalisierung* verwendet werden kann, beschreibt Dr. Starke ebenfalls im JavaSPEKTRUM [Sta19]. Zusammen mit [Sch17] wird auch ersichtlich, warum *Includes* für eine Auszeichnungssprache so wichtig sind und was sich damit unter anderem realisieren lässt.

Wer ganz hoch hinaus will, kann sich auf Basis des bisher Gezeigten schließlich auch *ausführbarer Dokumentation* widmen. Dabei werden je nach eingesetztem Tool entweder direkt im Programmcode, in AsciiDoc-Dokumenten oder separat als XML-Dateien Architekturvorgaben beschrieben. Diese lassen sich automatisiert testen und dienen gleichzeitig als Datengrundlage für die generierte Architekturdokumentation. So ist sichergestellt, dass die Anforderungen an die Architektur stets synchron mit den dazugehörigen Dokumenten sind. Die bekanntesten Vertreter solcher Tools sind *jQAssistant* und *ArchUnit*.

Nun lasse ich Sie, liebe Leser, gerne auf Entdeckungsreise in die Welt der vereinfachten Auszeichnungssprachen und Docs-as-Code-Konzepte gehen. ||

Der Autor

Christian Heitzmann
(christian.heitzmann@simplexacode.ch)
ist Java-, Python- und Spring-zertifizierter Softwareentwickler und Inhaber der SimplexaCode AG in Luzern. Er entwickelt seit über 20 Jahren Software und gibt seit über 12 Jahren Unterricht und Kurse im Bereich der Java- und Python-Programmierung, Mathematik und Algorithmik. Als Technical Writer dokumentiert er Softwarearchitekturen für Unternehmen.

Literatur & Links

[Art] Vorliegender Fachartikel im AsciiDoc-Format, siehe: **https://link.simplexacode.ch/nmbj**

[AscDoc] AsciiDoc, siehe: **https://asciidoc.org**

[AscDocLangDoc] AsciiDoc Language Documentation, siehe: **https://docs.asciidoctor.org/asciidoc/latest/**

[AscDocSyntaxQuickRef] AsciiDoc Syntax Quick Reference, siehe: **https://docs.asciidoctor.org/asciidoc/latest/syntax-quick-reference/**

[ascdocLIVE] asciidocLIVE, siehe: **https://asciidoclive.com**

[ConfWikiMarkup] Confluence Wiki Markup, siehe: **https://confluence.atlassian.com/doc/confluence-wiki-markup-251003035.html**

[Dem23] C. Demant, Software Due Diligence, 2. Auflage, Springer Gabler, 2023

[GitLabDocStyleGuide] GitLab, Documentation Style Guide, siehe: **https://docs.gitlab.com/ee/development/documentation/styleguide/**

[Hei22] C. Heitzmann, Javadoc mit Stil, in: JavaSPEKTRUM, 6/2022, siehe: **https://link.simplexacode.ch/ne6c**

[Heigl21] R. Heigl, Unternehmenswissen, in: iX 2/2021, Heise Medien, 2021

[Hol16] E. Holscher, Why You Shoudn't Use "Markdown" for Documentation, siehe: **https://ericholscher.com/blog/2016/mar/15/dont-use-markdown-for-technical-docs/**

[IDWS5] IDW Standard: Grundsätze zur Bewertung immaterieller Vermögenswerte, IDW S 5, 2015

[Kro] Kroki, Examples, siehe: **https://kroki.io/examples.html**

[Los21] M. G. Loschwitz, Wissen ist Macht, in: iX 2/2021, Heise Medien, 2021

[LudLic13] J. Ludewig, H. Lichter, Software Engineering, dpunkt.verlag, 2013

[Manifesto] Manifesto for Agile Software Development, siehe: **https://agilemanifesto.org**

[MediaWikiMarkSpec] Media Wiki, Markup spec, siehe: **https://www.mediawiki.org/wiki/Markup_spec**

[ninjs] **http://rst.ninjs.org** (offline)

[pandReadPickle] pandas API reference, pandas.read_pickle, siehe: **https://pandas.pydata.org/docs/reference/api/pandas.read_pickle.html**

[PEP257] PEP 257 – Docstring Conventions, siehe: **https://peps.python.org/pep-0257/**

[RST] reStructuredText, siehe: **https://docutils.sourceforge.io/rst.html**

[Sch17] M. Schlichting, Lebendige Dokumentation mit AsciiDoctor, in: Java aktuell 3/2017, DOAG, 2017

[Sphinx] Sphinx, reStructuredText, siehe: **https://www.sphinx-doc.org/en/master/usage/restructuredtext/index.html**

[Sta19] G. Starke, Internationalisierung von Dokumenten – i18n-light mit AsciiDoc & Co., in: JavaSPEKTRUM, 5/2019

[Sta22] G. Starke, arc42, die Achte, in: JavaSPEKTRUM, 1/2022

[TensFlowModel] TensorFlow Core v2.11.0 API Documentation, tf.keras.Model.compile, siehe: **https://www.tensorflow.org/api_docs/python/tf/keras/Model#compile**

[Textile4] Textile 4.0.0, siehe: **https://textile-lang.com**

[Tre22] S. Tremmel, # Überschrift – Mit Markdown schnell und einfach Texte auszeichnen, in: c't, 18/2022, Heise Medien

[WikiAuszSpr] Wikipedia, Auszeichnungssprache, siehe: **https://de.wikipedia.org/wiki/Auszeichnungssprache**

[WikiLightMarkLang] Wikipedia, Lightweight markup language, siehe: **https://en.wikipedia.org/wiki/Lightweight_markup_language**

[WikitextExamples] Wikimedia Meta-Wiki, Help:Wikitext examples, siehe: **https://meta.wikimedia.org/wiki/Help:Wikitext_examples**

[Zoe22] S. Zörner, Softwarearchitekturen dokumentieren und kommunizieren, 3. Auflage, Hanser, 2022