Literate Programming zur Dokumentation in der Systemadministration

Meik Teßmer

Computergestützte Methoden
Fakultät für Wirtschaftswissenschaften
Universitätsstraße 25
33615 Bielefeld
mtessmer@wiwi.uni-bielefeld.de

Abstract: Die Systemadministration kann auf ähnliche Weise von Literate Programming profitieren wie die Softwareentwicklung: Umfassende, lesbare und in Folge verständliche Systemdokumentation. Im Hinblick auf die Rolle der Dokumentation sowohl in der Softwareentwicklung als auch der Systemadministration betrachtet dieser Beitrag am Beispiel einer Fakultät der Universität Bielefeld die Anforderungen, die an eine Systemadministration gestellt werden, welche Werkzeuge dafür zur Verfügung stehen und wie Literate Programming helfen kann, diesen Anforderungen besser gerecht zu werden.

1 Einführung

Dieser Aufsatz soll den Betrieb von Informationssystemen betrachten, genauer die Dokumentation desselben. Um die Domäne nicht ausufern zu lassen, beschränke ich mich auf den Systembetrieb in Universitäten, in denen besonderer Handlungsbedarf zu bestehen scheint. Im Gegensatz zu Degenhard et al. [De12] wird die Systemadministration einer Dezentrale zu Grunde gelegt (Fakultät, Fachbereich oder Institut) und nicht die eines zentralen Hochschulrechenzentrums.

Der hier gezeigte Entwicklungs- und Dokumentationsansatz für die Systemadministration geht auf eine etwa 15 Jahre währende Tradition zurück und wurde erstmals 2001 in einer Dissertation vorgestellt [Kr01]. Seitdem wird in der Fakultät zusätzlich zum Betrieb eines heterogenen Netzwerks, das mit rund 200 Clients die Größenordnung eines mittelständischen Unternehmens hat, auch Anwendungssoftware entwickelt. Für all diese Aufgaben wird die hier vorzustellende Dokumentationsmethode seit Jahren angewandt. Ohne eine Dokumentationsdisziplin erscheint uns eine solche Aufgabe nicht mit der nötigen Effizienz und Sicherheit lösbar.

2 Systemadministration an einer Fakultät

2.1 Der Ist-Zustand

Die Aufgaben einer Systemadministration sind über die Jahre dieselben geblieben [He99, S. 2]), aber wo früher nur wenige große Systeme oder auch nur einzelne Dienste betreut werden mussten, wird ein Administrator heute mit einer Vielzahl verschiedener Hardware-Plattformen, Betriebssysteme und Diensten konfrontiert. Dazu kommt eine Nutzerschaft, die zum Teil umfangreiche Erfahrungen mit Computern und deren Verwendung besitzen und dementsprechend eigene Vorstellungen davon haben, wie Systeme funktionieren sollen. Beides hat zur Folge, dass altbewährte Verwaltungsmethoden überdacht und ggf. angepasst werden müssen.

Im Vergleich zu einem Unternehmen, wo die IT großen Einfluss auf Geschäftsprozesse und damit den Erfolg des Unternehmens hat (zur Rolle der IT siehe bspw. [Ba99]), gestaltet sich die Systemadministration in einer Fakultät ungleich schwieriger. Die Fachbereiche können weitgehend autonom agieren (auch finanziell) und müssen sich demzufolge nicht zwangsläufig an die Hard- und Software-Standards halten, die von einer Systemadministration erarbeitet werden. Der Administrator wird z. T. nur als Erfüllungsgehilfe gesehen, der die jeweiligen Wünsche der Bereiche zu erfüllen hat, unabhängig von Zweckmäßigkeit, Machbarkeit und Aufwand. Besonders deutlich wird dies bei der Beschaffung. Trotz entsprechender Beratung werden Geräte gekauft, die sich nicht sauber in die vorhandene Netzwerkinfrastruktur einbinden lassen und demzufolge nur eingeschränkt administriert und verwendet werden können. Ein zweites Beispiel ist Entwicklung von Software, ohne dass der Auftraggeber bereit ist, die dafür benötigten Mittel bereit zu stellen.

Als Folge hat sich nicht nur an der hier besprochenen Fakultät in den letzten Jahren etwa im Bereich mobiler Geräte (Laptops, Tablets) und Drucker eine Vielzahl unterschiedlicher Hersteller und Modelle angesammelt. Dies gilt ebenso für die verwendeten Betriebssysteme; es kommen auf den Desktops verschiedene Versionen von Microsoft Windows zum Einsatz (XP und 7, auch 8), einzelne Bereiche nutzen eine oder mehrere GNU/Linux-Distributionen oder Versionen von Mac OS X. Der Großteil der Installationen wird zentral betreut, für die übrigen versuchen die jeweiligen Nutzer dies in Eigenregie.

Die Aufgaben, die die Systemadministration der betrachteten Fakultät zu bewältigen hat, lassen sich aus Sicht der Nutzer zu folgendem rudimentären Funktionsmodell (s. dazu [He99, S. 108–111]) zusammenfassen (aufgeführt in absteigender Reihenfolge der Häufigkeit):

- Fehlerbehebung bei Hard- und Software
- Installation neuer Systeme
- Systemmanagement (Verwaltung von Accounts, Druckerrechten und Storage)
- Software-Deployment (Installation und Update)
- Beschaffung von Hard- und Software

- Reparatur von Hardware
- Entsorgung von Altsystemen samt Deinventarisierung
- Inventarisierung
- Beratung bei besonderen Anschaffungen und Projekten
- Bereitstellung spezieller Dienste (bspw. Evaluationsplattform)
- Entwicklung von Software

Im Hintergrund betreibt die Fakultät zur Bewältigung dieser Aufgaben u. a. eine Reihe von Servern (als Hardware und virtualisiert), die folgende Dienste bereit stellen:

- Virtualisierung von Systemen
- Backup-Service für Server
- Computing-Server für rechenintensive Aufgaben
- mehrere Netzwerkspeicher (zusätzlich zum AD-Storage, der durch das Hochschulrechenzentrum angeboten wird)
- mehrere Webserver und Wikis
- Versionierungsserver (Mercurial, Git, Subversion)
- Informationssysteme (Eigenentwicklungen) für das Prüfungsamt
- Monitoring-Plattform (Überwachung von USVs, Servern, Diensten, Außenanbindungen)
- Trouble-Ticket-System mit angeschlossenem Wiki

Angesichts dieser vielfältigen Aufgaben stellt eine sachgerechte Dokumentation eine große Herausforderung dar. Andere Fakultäten haben sich dieser entweder gar nicht gestellt und betreiben personalintensive und fehleranfällige „Turnschuh- Administration“ oder sie kombinieren mehrere Dokumentationssysteme und -formen: Listen in Tabellenkalkulationen, Wikis, Office-Dokumente, aber auch Ausdrucke und Notizzettel. Im günstigsten Fall kann auf Runbooks o. Ä. zurück gegriffen werden, die übliche Arbeitsgänge dokumentieren und neuen Mitarbeitern der Systemadministration als einführendes Material dienen können.

2.2 Eine sachgerechte Dokumentation

Welchen Kriterien muss eine sachgerechte und effiziente Dokumentation genügen? Nach Gaus ist der übergeordnete Zweck einer Dokumentation Wiederfinden und Nutzbarmachen von Dokumenten und Informationen [Ga05, S. 11]. Aufbauend darauf muss sie folgende Teilprozesse unterstützen:

- Sicherstellen des laufenden Betriebs durch Statusüberwachung und die Behandlung von Ausnahmefällen. Dazu gehören Fehlerbehebung auf Hard- und Softwareseite sowie die Reparatur von Hardware.
- Ausführen von Routinearbeiten wie Systemmanagement, Beschaffung von Hard- und Software, die Installation neuer Systeme, das Software-Deployment,

die Entsorgung von Altsystemen und deren Deinventarisierung sowie die Inventarisierung vorhandener Hard- und Software.

- Auftragsarbeiten, bspw. die Entwicklung von Software, das Bereitstellen spezieller Dienste und die Beratung bei besonderen Anschaffungen und Projekten.
- Betriebslenkungsaufgaben, die Erstellung von Statistiken oder die Planung strategischer Investitionen.

Welche formalen Anforderungen leiten sich daraus ab? Was hat sich in der Praxis bewährt? In der Fachliteratur finden sich viele Beschreibungen, *wie* einzelne Teilaufgaben des Systemmanagements zu behandeln sind. Meist handelt sich dabei um technische Beschreibungen eines Systems oder Dienstes, ergänzt durch eine entsprechende Anleitung zur Einrichtung (siehe dazu bspw. [AL07], [Ba05], [Bu04], [Ko13], [PW12], [Po13]). Es wird jedoch nicht darauf eingegangen, was der Systemadministrator beim alltäglichen Umgang auf welche Weise dokumentieren sollte. Eine Ausnahme stellt der Beitrag von Langston für die USENIX Special Interest Group for Sysadmins (LISA) dar. Er formuliert in [La03] aus Sicht eines Systemadministrators vier Eigenschaften, die eine gute Dokumentation haben sollte:

1. **Useful**: Sie enthält die benötigten Informationen.
2. **Accessible**: Zweck und Zielgruppe der Dokumentation sind klar und eindeutig; die einzelnen Elemente der Dokumentation lassen sich schnell auffinden und verwenden.
3. **Accurate**: Die vorhandenen Informationen sind vollständig, korrekt und aktuell.
4. **Available**: Sie ist in jeder Situation verfügbar.

Der Vergleich dreier in der Administration verbreiteter Dokumentationsformen (Papier, Web-basiert, einfache Textdateien) anhand dieser Kriterien führt ihn zu folgender Empfehlung:

- Eine Dokumentation sollte in digitaler Form vorliegen und online verfügbar sein. Sie ist dann mit den üblichen Werkzeugen (Webbrowser) zeit- und ortsunabhängig zu erreichen und lässt sich schnell aktualisieren. Zudem unterstützen Suchfunktionen oder Werkzeuge wie `grep` das Auffinden von Informationen.
- Für die Bearbeitung sollten möglichst keine zusätzlichen Werkzeuge installiert werden müssen, bspw. eine Datenbank oder ein spezieller Editor.
- Einfache Textdateien sind wesentlich einfacher mit Versionsverwaltungssystemen zu verwalten als Binärformate.
- Für den Notfall (Totalausfall des Netzwerks oder der Stromversorgung), wenn kein Zugriff auf die digitale Dokumentation möglich ist, ist eine gedruckte Fassung sinnvoll. Hierbei muss jedoch darauf geachtet werden, dass es sich um die jeweils aktuelle Fassung handelt.

Wie steht es um das *Nutzbarmachen*? Vorstellbar ist, dass die Verarbeitung einer Information nicht wie üblich in Form einer manuellen administrativen Handlung erfolgt, sondern dass das Dokument selbst „ausführbar“ ist, die vorzunehmenden Änderungen also

aus ihm abgeleitet und umgesetzt werden können. Burgess formuliert drei *Meta-Prinzipien*, aus denen er die Notwendigkeit einer solchen, auf hohem Abstraktionsniveau agierenden und standardisierten Administration ableitet. Dazu stellt er u. a. `Cfengine` vor, ein Werkzeug, dass mit Hilfe einer deklarativen Sprache Sys- temkonfigurationen automatisiert einrichten und prüfen kann [Bur04, S. 6, 238, 258]. Eingebettet in eine digitale Form der Dokumentation würden so beschriebene Konfigurationen – Konfigurations-Code – diese ganz konkret *nutzbar* machen.

Dieses direkte Nebeneinander von Dokumentation und Code trägt zur Lösung eines weiteren Problems bei, das insbesondere bei der Pflege externer Dokumentation auftritt: Wie lässt sich die Konsistenz von Code und zugehöriger Dokumentation sicherstellen? Ohne entsprechende Gegenmaßnahmen tendiert Dokumentation dazu, schnell zu veralten (s. bspw. [KF09]). Wenn sich nun sowohl Code als auch Dokumentation im selben Dokument befinden, ist es wahrscheinlicher, dass beide zueinander konsistent sind und gemeinsam zeitnah gepflegt werden.

- Fasst man die zuvor beschriebenen Anforderungen zusammen, dann sollte eine Dokumentation für die Systemadministration folgenden formalen Anforderungen genügen:
- Das Ausgangsformat erlaubt eine Konvertierung in eine Online-Repräsentation sowie in eine für den Druck geeignete Form.
- Das Ausgangsformat nutzt eine einfache Auszeichnungssprache und lässt sich als einfache Textdatei speichern.
- Die Erstellung und Pflege erfordert keine umfangreichen Installationen wie Datenbanken, Webserver etc.
- Die „ausführbare" Beschreibung der Konfigurationen befindet sich eingebettet im Text.

Bis auf die Integration der Konfigurationen in die Dokumentation erfüllen einfache Textdateien, formatiert mit Hilfe von Markup-Sprachen wie `reST` oder `Markdown`, die ersten drei Kriterien. Sie können mit Konvertern wie `Docutils` oder `pandoc` in verschiedene Online- und Druckformate übersetzt werden und benötigen für die Bearbeitung nur einen gewöhnlichen Editor. Werkzeuge wie `Cfengine`, `Chef`, `Puppet` oder `Salt` können System- oder Dienstkonfigurationen aus einfachen Textdateien extrahieren und sie automatisiert in administrative Handlungen überführen.

Offen ist aber noch die Frage, wie sich Beschreibungen und Text so kombinieren lassen, dass auf der einen Seite eine lesbare Dokumentation entsteht und auf der anderen ein „ausführbarer" Text. Eben diese Kombinationsmöglichkeit bietet das *Literate Programming*. Entworfen mit dem Ziel, die Dokumentation im Rahmen der Softwareentwicklung zu verbessern, bietet dieser Ansatz auch Vorteile für die Dokumentation der Systemadministration.

3 Literate Programming

3.1 Die (Neben-)Rolle der Dokumentation in der Softwareentwicklung

Dokumentation in der Softwareentwicklung lässt sich in die Kategorien *extern* und *intern* aufteilen. Externe Dokumentation umfasst alle Formen von Informationen, die zusätzlich zum Quellcode erfasst werden: Vision, Design, Anforderungen etc. Mögliche Formen sind Prosa, Bilder, Diagramme, aber auch Einträge in Wikis und Ticketsysteme. Die interne Dokumentation besteht aus den Kommentaren, um die Entwickler den Quellcode ergänzen. Man könnte aber auch den Quellcode selbst dazu zählen, denn das tatsächliche Design des Softwaresystems existiert nur im Code.

Unabhängig davon, ob sie extern oder intern vorliegt, soll die Dokumentation den Quellcode und die damit verbundenen Entwurfsentscheidungen, die während des Entwicklungsprozesses gefallen sind, *kommunizierbar* machen. Ziel ist, die Wartbarkeit (Fehlerbehebung, Anpassung und Weiterentwicklung) und Portabilität (Übertragung auf neue Hard- und Softwareplattformen) des Systems zu gewährleisten und es damit zukunftssicher zu machen. Für einen Entwickler bedeutet dies, dass er (im Idealfall) gesuchte Quellcode-Abschnitte schnell finden und ihre Funktionsweise verstehen kann (Analysierbarkeit) sowie erforderliche Änderungen immer konform zu getroffenen Designentscheidungen insbesondere auf der Architekturebene vornimmt.

Die Bedeutung guter Dokumentation ist den Entwicklern bewusst, es wird aber dennoch eher nachlässig dokumentiert (s. bspw. [PC86, S. 255], [Br95, S. 169]). Die Gründe sind vielfältig und nicht immer scharf zu benennen. Zunächst ist das Schreiben guter Dokumentation eine anspruchsvolle und zeitintensive Tätigkeit. Ist das Talent vorhanden, mangelt es oft an der notwendigen Zeit, besonders in Phasen der Fehlerbehebung. Ein weiterer Grund mag der Unwillen vieler Entwickler sein zu dokumentieren. Selbstdokumentierender Code sei doch ausreichend und dem Kunde ein lauffähiges System wichtiger als eine umfassende Dokumentation (siehe dazu bspw. [Ma09] und [Mc04]). Die derzeitig populären *Agilen Methoden* verstehen Dokumentation denn auch als notwendiges, aber zu minimierendes Übel, der man zudem nicht trauen kann [Am02, S. 165f]. Dass der Dokumentation nicht getraut werden kann, ist allerdings ein Resultat der mangelhaften Pflege derselben. Ein wesentliches Problem ist die fehlende Konsistenz von (insbesondere externer) Dokumentation und zugehörigen Quellcode-Bereichen. Ohne aktive Gegenmaßnahmen wird sie nach einiger Zeit der Wartung immer schlechter. Das hat zur Folge, dass bei Wartungsarbeiten Strukturentscheidungen, die nicht aus dem Code ersichtlich sind, nicht berücksichtigt werden können. Der Verlust dieses Zusatzwissens erschwert das Systemverstehen und damit die Wartbarkeit nachhaltig.

3.2 Literate Programming als integrierender Ansatz

Auch wenn der Quellcode weitgehend selbstdokumentierend ist, lassen sich allein schon aus syntaktischen Gründen nicht alle für die Entwicklung erforderlichen Informationen in dieser Form ausdrücken; eine Dokumentation über den Quellcode hinaus ist notwen-

dig. Ein erster Lösungsansatz besteht darin, die Trennung von Dokumentation und Quellcode schlicht aufzuheben und beides zusammen zu verfassen. Das, was sich gut in der gewählten Programmiersprache formulieren lässt, wird als Quellcode festgehalten, alles andere in Form von Prosa, Bildern etc. Denkt man diesen Ansatz noch einen Schritt weiter, dann werden die Erläuterungen nicht zum Quellcode, sondern der Quellcode den Erläuterungen beigefügt; das Dokument wird damit zum primären Medium. Aus diesem lassen sich dann eine oder mehrere Quellcode-Dateien extrahieren und weiter verarbeiten.

1984 schlug Knuth mit seinem „Literate Programming" ein solches System vor. Er verfolgte damit das Ziel, die Verständlichkeit von Programmen zu verbessern. Dazu entwarf er eine einfache Auszeichnungssprache namens `WEB`, die Dokumentation und Quellcode in sog. *Chunks* einteilt. Jeder Code Chunk wird mit einen besonders markierten Bezeichner eingeleitet und endet, wenn eine Zeile mit einem `@` beginnt:

```
<<ein Bezeichner>>=
def meine_funktion():
  ...

@ Hier steht keine Code mehr.
```

Nach dem `@` beginnt ein Documentation Chunk. Auf diese Weise können beliebig Chunks aneinander gefügt werden, unabhängig davon, ob es sich um Dokumentation oder Quellcode handelt. Wird ein Code Chunk mit einem schon vergebenen Bezeichner eingeleitet, werden sie bei der Extraktion (`tangle`) gemäß der Reihenfolge ihres Auftretens im Dokument zusammengeführt.

Innerhalb eines Code Chunks lassen sich auch andere Code Chunks referenzieren:

```
<<ein weiterer Bezeichner>>=
def eine_weitere_funktion():
  <<ein Bezeichner>>
  for line in file:
      ...
@
```

`<<ein Bezeichner>>` fungiert hier als Platzhalter und wird bei der Extraktion durch den benannten Code Chunk ersetzt. Wo sich dieser im Dokument befindet, ist unerheblich. Damit bleibt es dem Entwickler überlassen, ob er top-down, bottom-up oder auf andere Art sein Programm entwirft, losgelöst von den Vorgaben der verwendeten Programmiersprache.

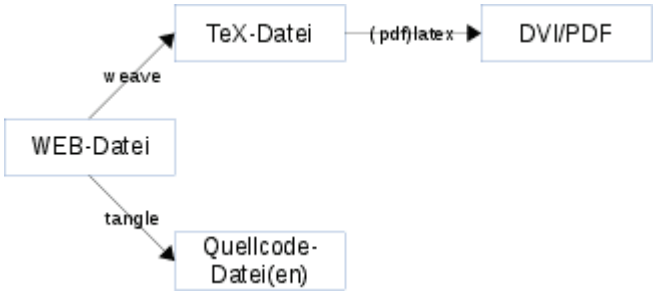

Abbildung 1: Verarbeitung einer WEB-Datei.

Ursprünglich hatte Knuth `WEB` für die Entwicklung von Pascal-Programmen entworfen und als Auszeichnungssprache für die Dokumentations-Chunks das von ihm entwickelte TeX-System vorgesehen; letzteres, weil seiner Ansicht nach Programme als Kunst angesehen werden sollten und daher eine angemessene Darstellung verdienen. Ramsey entwickelte 1989 mit `noweb` ebenfalls ein Werkzeug für das literate Programmieren, allerdings mit dem Ziel, einen möglichst einfachen, von Programmiersprachen unabhängigen Zugang zum LP zu bieten. Neben TeX unterstützt `noweb` auch LaTeX, HTML und `troff` als Backend-Formate. Diese Flexibilität war ausschlaggebend dafür, die literate Systemadministration auf Basis von `noweb` zu entwickeln.

Die Entwicklung von Software mit Hilfe dieses dokumentenzentrierten Ansatzes hat weitreichende Auswirkungen. Da Dokumentation und Quellcode eng zusammen stehen, kann ihre Konsistenz zueinander besser gewahrt werden [Th86, S. 204]. Ein Dokument kann zudem mehrere Quellcode-Dateien aufnehmen, wodurch eine geschlossene Darstellung von in den Dateien verteilter Zusammenhänge möglich ist [De92, S. 20]. Der Entwickler kann weiterhin unabhängig von der gewählten Programmiersprache seine Gedanken und Überlegungen in der ihm angemessen erscheinenden Reihenfolge ausdrücken [Kn92, S. 168].

Dass Literate Programming funktioniert, wurde und wird anhand vieler Projekte deutlich. Ramsey pflegt auf seiner Website [Ra14] eine Liste von Personen und Projekten, die noweb einsetzen, darunter auch das an der Fakultät entwickelte `relax` [Wo14].

4 Literate Programming in der Systemadministration

4.1 Erfahrungen mit LP bei der Softwareentwicklung

Die Entwicklung des *Fakultätsinformationssystems* (FakInfo; s. auch [ST09]) für die Fakultät erfolgte literat unter Verwendung von noweb. Bei der abschließenden Bewertung wiesen die Entwickler auf mehrere Probleme hin, dieses Vorgehen mit sich bringt:

- Als Auszeichnungssprache für die Dokumentation wurde LaTeX eingesetzt. Der Aufwand bei der Eingabe ist im Vergleich zu alternativen Auszeichnungs-

sprachen wie `Markdown` [Gr04] oder `reStructuredText` unverhältnismäßig hoch.

- Ebenso wie `WEB` sieht `noweb` vor, dass mit nur *einer* Datei gearbeitet wird, unabhängig von der Größe des zu entwickelnden Systems. (Auch die drei Grundbausteine für das umfangreiche Satzsystem bestehen jeweils aus nur einer Quelldatei: `tex.web` (TeX-Compiler) ~25000 Zeilen, `mf.web` (METAFONT) ~23000 und `bibtex.web` (BIBTeX) ~11600 Zeilen).) Heutige Editoren haben mit großen Dateien keine Schwierigkeiten und helfen dem Entwickler u.a. mit Outlining-Funktionen bei der Navigation. Sie beherrschen dabei jedoch nicht den Umgang mit der Code Chunk-Notation, so dass hier durch die fehlende Unterstützung mit zunehmendem Dateiumfang der Überblick verloren geht.
- Die Arbeitsteilung im Team wird durch die Verwendung nur einer Datei erschwert. Auch wenn moderne Versionsverwaltungssysteme leistungsfähige Merge-Mechanismen besitzen, ist die Wahrscheinlichkeit eines Konflikts bei nur einer Datei wesentlich höher als bei einer Verteilung der Dokumentation auf mehrere Dateien.
- Das von `noweb` durchgeführte Syntax Highlighting kann bestenfalls als rudimentär bezeichnet werden und ist aus heutiger Sicht völlig unzureichend.
- Die Bezeichner der zu extrahierenden Code Chunks müssen dem verwendeten Build-Werkzeug bekannt gemacht werden, bspw. in einem `Makefile` oder Shell-Skript. Hier kam es bei Refactoring-Maßnahmen immer wieder zu Fehlern, weil Änderungen von Bezeichnern nicht nachgeführt wurden.

Schon während der Entwicklung von FakInfo wurden Werkzeuge zur Behandlung einiger dieser Probleme geschrieben:

- Der `includer` definiert eine einfache Syntax, die das Zusammenfügen von Dateien erlaubt. Damit ließ sich die Quelldatei in eine Haupt- und mehrere Nebendateien aufspalten.
- Der `chunk-viewer` liefert eine Übersicht aller definierten Code Chunks sowie der zugehörigen Root Chunks einer Datei.
- Der `highlighter` ersetzt die Syntax Highlighting-Funktionalität von `noweb` durch die des LaTeX-Pakets `listings`.

Mit Hilfe dieser Werkzeuge konnte das Projekt erfolgreich abgeschlossen werden. Das Fakultätsinformationssystem ist seit 2008 produktiv im Einsatz.

4.2 Anpassungen für Verwendung von LP in der Systemadministration

Die Verwendung des `includers` kann zu einem unerwünschten Nebeneffekt führen. Da mehrere Dateien zu einer Gesamtdatei zusammengefasst werden, besteht das Risiko, dass es Code Chunks mit identischen Bezeichern gibt, die inhaltlich jedoch nichts miteinander zu tun haben. Der `tangle`-Prozess würde sie als zusammengehörig behandeln

und aneinander hängen. Bei der Beschreibung von Konfigurationen, die auf mehreren Systemen aktiv sein sollen, ist dieses Risiko aufgrund identischer Konfigurationsdateinamen besonders hoch. Aus diesem Grund wurde im Vorfeld die Verwendung des `includers` ausgeschlossen und Inklusionen von Dateien zwecks Wiederverwendung von Konfigurationsbausteinen an das Administrationswerkzeug delegiert.

Für den Einsatz von LP in der Systemadministration entschieden wir uns, ein Dokumentationssystem zu suchen, das statt LaTeX eine einfachere Markup-Sprache verwendet und die Verarbeitung mehrerer Dateien inkl. Querverweisen erlaubt. Die Erfahrungen mit Python, die während der Entwicklung von FakInfo gemacht wurden, führten uns zu Sphinx (s. [Br14]). Ursprünglich für die Dokumentation des Python- Interpreters und seiner Standardbibliothek entwickelt, unterstützt es die Dokumentation beliebiger Systeme über mehrere Dateien hinweg und beherrscht u. a. die Ausgabe von HTML und LaTeX/PDF. Die HTML-Ausgabe bietet außerdem eine Suchfunktion und einen Index. Als Markup-Sprache nutzt Sphinx `reStructuredText` (`reST`), eine vereinfachte Auszeichnungssprache, die auch in der reinen Textform gut lesbar ist. Über einen Plugin-Mechanismus lassen sich auch ASCII-Grafiken einbetten, die mit Hilfe von `aafig` automatisch zu Bilddateien konvertiert und in die Ausgabe eingefügt werden.

Damit die noweb-Syntax von Sphinx verarbeitet werden kann, ist eine Übersetzung der die Code Chunk begrenzenden Auszeichnungen in eine `reST`-konforme Syntax erforderlich. Hierfür wurde das Werkzeug `nw2rst` (s. [Te08]) entwickelt, das zwischen den Verarbeitungsprozess von `noweb`-Markup zum Dokument geschaltet wird. Die mit Sphinx möglichen Querverweise sind für die Arbeit mit mehreren Dateien ausreichend. Die Code Chunks hingegen lassen sich durch den Wegfall des `includers` nicht über Dateigrenzen hinweg definieren. Dieser Umstand muss bei der Erstellung der Dokumentation berücksichtigt werden, er wird aber durch die Möglichkeiten des Administrationswerkzeugs ausgeglichen.

Um die Abhängigkeit von einer externen Datei zu verhindern, die die Extraktion von Code Chunks bestimmt, wurde ein `build-script` nach einem Vorschlag des LyX-Projekts (s. dazu auch [Te09]) entworfen. In jeder Datei wird dazu ein Code Chunk namens `build-script` definiert, der die Extraktion in Form eines Shell-Skripts beschreibt. Die eigentliche Extraktion wird durch ein Shell-Skript angestoßen, das diesen eingebetteten Code Chunk ausführt. Dank dieser Kombination von Shell-Skripten muss der Entwickler die `noweb`-Datei nicht verlassen, wenn er einen zu extrahierenden Code Chunk löschen, umbenennen oder hinzufügen will.

4.3 Salt als Werkzeug zur Administration

Die Idee eines *ausführbaren* Dokuments zur Systemadministration lässt sich mit Hilfe von Konfigurationsmanagement-Werkzeugen wie `Cfengine`, `Chef`, `Puppet` oder `Salt` realisieren. Ihre textbasierten Konfigurationsbeschreibungen können dank Literate Programming wie der Quellcode eines Programms in die Dokumentation einbetten werden.

Nach dem Studium von Erfahrungsberichten und einer halbjährigen Testphase mit `Puppet` haben wir uns für `Salt` entschieden. `Salt` verwendet als Beschreibungssprache YAML (Abkürzung für *Yet Another Markup Language*), eine deklarative Sprache, die im Vergleich zur *Puppet Language* sehr einfach strukturiert und wesentlich leichter zu erlernen ist. Trotz dieser Einfachheit lassen sich damit auch komplizierte Konfigurationen formulieren. Darüber hinaus können über weitere sog. *Renderer* Template-Sprachen wie Jinja oder Mako verarbeiten werden, mit deren Hilfe sich Konditionalausdrücke und Schleifen formulieren lassen.

Die Konfiguration eines Systems erfolgt bei `Salt` als Zustandsbeschreibung in Form sog. *Salt States*. Ausgangspunkt ist die Datei `top.sls`, in der alle zu verwaltenden Systeme aufgelistet sind. Hier ein Beispiel mit nur einem System namens `atlas`:

```
# top.sls
base:
  'atlas':
    - core
    - ssh.server
    - users.comet
```

`atlas` nutzt u. a. Einstellungen aus `core`, die in der Datei `core.sls` genauer beschrieben sind. Die Aufgabe von `core.sls` besteht darin festzulegen, welche Softwarepakete auf einem System installiert sein sollten. In diesem Beispiel sind das eine Shell, ein Terminalmanager und ein Filemanager namens `ranger`. Dieser Filemanager wird allerdings auf FreeBSD-Systemen mit einem anderen Paketnamen installiert als auf Debian-basierten Systemen. Um bei der Konfiguration aber plattformunabhängig zu sein, wird hier mit Hilfe von Jinja eine Variable namens `ranger` auf den passenden Paktetnamen gesetzt und für die Installation verwendet:

```
# core.sls
{% if grains['os'] == 'FreeBSD' %}
{% set ranger = 'sysutils/py-ranger' %}
{% elif grains['os'] in ['Debian', 'Ubuntu'] %}
{% set ranger = 'ranger' %}
{% endif %}

core:
  pkg:
    - installed
    - names:
      - bash
      - tmux
      - {{ ranger }}
```

Mit dem Befehl `salt 'atlas' state.highstate`, ausgeführt auf dem *Salt Master*, wird diese Konfiguration dann auf dem System ausgerollt.

Prinzipiell ist es möglich, mit solchen Zustandsbeschreibungen quasi auf Knopfdruck eine vollständige Systemlandschaft einzurichten. Dank seiner auf Remote-Execution basierenden Architektur ist `Salt` außerdem auch sehr gut für die Ad hoc-Administration geeignet. Dazu ein Beispiel: Um auf allen verwalteten Systeme den Patch für die in diesem Jahr bekannt gewordene OpenSSL-Lücke (auch *Heartbleed Bug* genannt) einzuspielen, ist nur der folgende Befehl auszuführen:

```
salt '*' pkg.install openssl refresh=True
```

Diese kleinen Beispiele sollen demonstrieren, wie einfach eine Konfiguration mit `Salt` beschrieben und ausgerollt werden kann. Als Managementwerkzeug ist `Salt` sehr leistungsfähig und eine detaillierte Erläuterung der Konfigurationsmöglichkeiten würde den Rahmen sprengen. Weiterführende Informationen sind auf der Website des Projekts zu finden, die eine umfassende Dokumentation und viele Anwendungsbeispiele bietet.

5 Fazit

Eine umfassende Dokumentation der IT-Systeme ist angesichts der Vielfalt der Hardware, Betriebssysteme und Dienste notwendiger denn je. Dieser Beitrag soll demonstrieren, dass dazu nicht eine Vielzahl unterschiedlicher Werkzeuge notwendig ist, sondern für viele der zu dokumentierenden Aufgaben auch ein klassisches Textdokument ausreicht, angereichert mit Hilfe des Literate Programming-Ansatzes um ausführbare Konfigurationsbeschreibungen. Die hier beschriebene Vorgehensweise kommt seit zwei Jahren für die Verwaltung sämtlicher Server, Dienste und zentral verwalteter GNU/Linux-Desktops erfolgreich zum Einsatz.

Die literate Systemadministration stößt an ihre Grenzen, wenn stark formalisierte Prozesse dokumentiert werden sollen. Ein Beispiel sind Tickets eines Helpdesk-Systems. Diese als Textdokument umzusetzen und nachzuhalten, ist vom Aufwand her unverhältnismäßig und geht einher mit einem Funktionalitätsverlust (Generierung von Berichten, Nachhalten von Arbeitszeiten und deren Abrechung usw.). Derzeit wird an der Fakultät noch ein eigenständiges Ticket-System eingesetzt, das auch ein Wiki beinhaltet. Die Umstellung auf das Versionsverwaltungssystem Fossil, die vor einem Jahr vorgenommen wurde, bietet jedoch die Möglichkeit, Tickets und Wiki samt Quelldateien der Dokumentation in einer einzigen Repository-Datei zu vereinen. Damit wäre dann der letzte Schritt hin zu einer „umfassenden“ Dokumentation getan.

Literaturverzeichnis

[AL07] Adelstein, T.; Lubanovic, B.: Linux Schnellkurs für Administratoren. Beijing et al.: O’Reilly 2007.

[Am02] Ambler, S.: Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process. New York: John Wiley&Sons 2002.

[Ba99] Bass, L.; Clements, P.; Kazman, R.: Software Architecture in Practice. Reading, Mass. et al.: Addison-Wesley 1999.

[Ba05] Bauer, M.D.: Linux Server-Sicherheit. 2. Aufl. Beijing et al. : O'Reilly 2005.
[Br14] Brandl, G. (2014): Sphinx team: Sphinx document generator. http://sphinx-doc.org/
[Br95] Brooks, F.P.: The Mythical Man Month. Boston et al. : Addison-Wesley 1995.
[Bu04] Burgess, M.: Principles of Network and System Administration. 2. Aufl. The Atrium, Southern Gate, Chichester, West Sussex PO19 8SQ, England : John Wiley&Sons 2004.
[De12] Degenhard, T. ; Korff, M. ; Schäfermeier, U.: Einführung eines nachhaltigen IT-Service Managements an der FH Bielefeld. In: Brandt-Pook / Fleer / Spitta / Wattenberg [Hrsg.] Nachhaltiges Software Management. SWM 2012. Bielefeld (2012), S. 123–134.
[De92] Denert, E.: Software-Engineering. Berlin et al. : Springer 1992.
[Ga05] Gaus, W.: Dokumentations- und Ordnungslehre: Theorie und Praxis des Information Retrieval. 5. überarb. Aufl., Berlin, Heidelberg : Springer 2005.
[Gr14] Gruber, J. (2014): Markdown. http://daringfireball.net/projects/markdown/
[He99] Hegering, H-G.; Abeck, S.; Neumair, B.: Intergriertes Management vernetzter Systeme: Konzepte, Architekturen und deren betrieblicher Einsatz. Heidelberg : dpunkt 1999.
[KF09] Kiniry, J.R.; Fairmichael, F.: Ensuring Consistency between Designs, Documentation, Formal Specifications, and Implementations. In: Lewis / Poernomo / Hofmeister [Hrsg.] Component-Based Software Engineering. Berlin Heidelberg: Springer 2009. S. 242-261.
[Kn92] Knuth, D.E.: Literate Programming - Center for the Study of Language Information, 1992.
[Ko13] Kofler, M.: Linux: Das umfassende Handbuch. Galileo Computing 2013.
[Kr01] Krüll, J.: Literate Systemadministration. Münster: LIT 2001.
[La03] Langston, M.C.: Documentation Writing for System Administrators. Berkely: USENIX Association 2003.
[Ma09] Martin, R.C.: Clean Code: Refactoring, Patterns, Testen und Techniken für sauberen Code. Heidelberg et al. : mitp 2009.
[Mc04] McConnell, S.: Code Complete. 2. Aufl. Redmond, Washington, Microsoft Press 2004.
[PC86] Parnas, D. L.; Clements, P. C.: A Rational Design Process: How and Why to Fake it. IEEE Transactions on Software Engineering 12 (1986), Nr. 2, S. 251–256.
[PW12] Plötner, J.; Wendzel, S.: Linux: Das umfassende Handbuch. 5. akt. Aufl., Galileo Computing 2012.
[Po13] Pollei, R. P. (2013): Debian 7: System Administration Best Practices. Birmingham - Mumbai: Packt.
[Ra14] Ramsey, N. (2014): Projects using noweb. http://www.cs.tufts.edu/~nr/noweb/users
[ST09] Spitta, T.; Teßmer, M.: Informationen aus dem eKVV - Hilfe oder Bürokratie? In: H1 Bd. 2009, Universität Bielefeld, Nr. 2, S. 17 und: http://pub.uni-bielefeld.de/publication/2680573
[Te08] Teßmer, M. (2008): nw2rst. http://www.wiwi.uni-bielefeld.de/lehrbereiche/statoekoinf/comet/mtessmer/Software/nw2rst
[Te09] Teßmer, M. (2009): Literate Programming mit LyX. http://www.wiwi.uni-bielefeld.de/lehrbereiche/statoekoinf/comet/mtessmer/Downloads/literate_programming_with_lyx.pdf
[Th86] Thimbleby, H.: Experiences of Literate Programming using cweb (a variant of Knuth's WEB). The Computer Journal 29 (1986) 3, S. 201–211.
[Wo14] Wolf, P. (2014): R-package-relax. http://www.wiwi.uni-bielefeld.de/lehrbereiche/statoekoinf/comet/wolf/R_package_relax