2 Einführung in das Konfigurationsmanagement

»Die Theorie träumt, die Praxis belehrt.«

(Karl von Holtei, deutscher Schriftsteller, 1798–1880)

Dieses Kapitel vermittelt Ihnen die Grundlagen des in diesem Buch vorgestellten Konfigurationsmanagement-Prozesses. Ich habe diese rein theoretische Einführung bewusst an den Anfang gestellt, trotz des Risikos, auf den einen oder anderen Leser eventuell abschreckend zu wirken. Meiner Erfahrung nach werden an sich einfache Sachverhalte durch die technischen Eigenheiten und Einschränkungen eines Werkzeuges oft unnötig verkompliziert. Hat man hingegen die Grundidee erst einmal verstanden, fällt es viel leichter, die konkrete Umsetzung nachzuvollziehen. Daher spielen die drei Werkzeuge Subversion, Maven und Redmine in diesem Kapitel zunächst keine Rolle. Was allerdings nicht bedeutet, dass ich vorhabe, Sie auf den folgenden Seiten mit praxisferner Träumerei zu langweilen. Das obige Zitat gibt ziemlich genau meine Erfahrungen mit »abgehobenen« theoretischen Erläuterungen wieder und diente mir insbesondere im Einführungskapitel als Leitlinie.

Um den Einstieg zu erleichtern, habe ich die Einführung zudem in zwei Abschnitte unterteilt. Die meiner Ansicht nach für Softwarearchitekten und -entwickler wichtigsten Bausteine des *Kernprozesses* beschreibe ich nach einer allgemeinen Begriffsbestimmung in Abschnitt 2.2. Ich empfehle, diesen Teil der Einführung in jedem Fall zu lesen. Wer noch einen Schritt weiter gehen will, findet in Abschnitt 2.3 einige Anregungen zum Ausbau des Prozesses.

2.1 Was ist Konfigurationsmanagement?

Jedes Softwareentwicklungsprojekt, unabhängig davon, ob es nach einem umfangreichen Wasserfall-Vorgehensmodell oder einer der schlanken, agilen Methoden durchgeführt wird, lebt unterm Strich von

den erstellten Ergebnissen. Konfigurationsmanagement ist letztlich nichts anderes als der Versuch, diese Ergebnisse auch in der allgemeinen Hektik des Projektalltags sicher zu verwalten und den Teammitgliedern jederzeit kontrollierten Zugriff darauf zu gewähren. Ein KM-Prozess bildet das Fundament, auf dem ein Team erfolgreich und effizient zusammenarbeiten kann.

Verzichtet man auf dieses Fundament, führt dies zu Qualitätsproblemen, reduzierter Produktivität und unter Umständen zu einem Verlust der Kontrolle über das Projekt. Die genannten Schwierigkeiten entstehen, weil wir als Individuen nur schlecht auf die Zusammenarbeit im Team vorbereitet sind[1]. Um diesen Mangel zu kompensieren, benötigen wir Richtlinien und Vorschriften. Sie schränken unsere individuelle Freiheit im Projektalltag ein und stellen dadurch sicher, dass unsere Arbeitsergebnisse nicht mit denen anderer Teammitglieder kollidieren.

2.1.1 Ziele des Konfigurationsmanagements

Softwareprojekte basieren auf mehreren Regelwerken, angefangen beim eingesetzten Vorgehensmodell bis hin zum Projektplan mit den Aufgaben pro Teammitglied. Konfigurationsmanagement ist eine weitere Sammlung von Richtlinien. Diese beschreiben einen *Prozess,* der die Zusammenarbeit im Team regelt und optimiert. Der in diesem Buch beschriebene KM-Prozess verfolgt vier Ziele (siehe Abb. 2–1). Er hilft, Änderungen an den einzelnen Elementen eines Projektes unter Kontrolle zu behalten und die Qualität des erstellten Produktes zu gewährleisten. Ferner steigert er die Produktivität im Team und unterstützt über eine größere Transparenz das Management des Projektes.

Änderungen kontrollieren

Änderungen geraten außer Kontrolle, wenn die Kommunikation im Team nicht mehr funktioniert. Solange sich jeder vor Durchführung einer Änderung mit den anderen Beteiligten abstimmt, sind keine größeren Schwierigkeiten zu erwarten. Diese permanente Abstimmung funktioniert jedoch nur in sehr kleinen Teams mit maximal zwei bis drei Personen – und selbst dann meiner Erfahrung nach nur, wenn alle

1. Ein aufmerksamer Leser hat hierzu einmal angemerkt, dass wir von Natur aus »Herdentiere« sind, die früher nur im Team überleben konnten. Trotzdem funktionieren viele Teams zunächst schlecht, wir stehen also vor einem Widerspruch. Eine nicht ganz ernst zu nehmende Erklärung wäre, dass Softwareprojekte nicht primär dem eigenen Überleben dienen. Auch wenn das Management dies gerne genau so darstellt ...

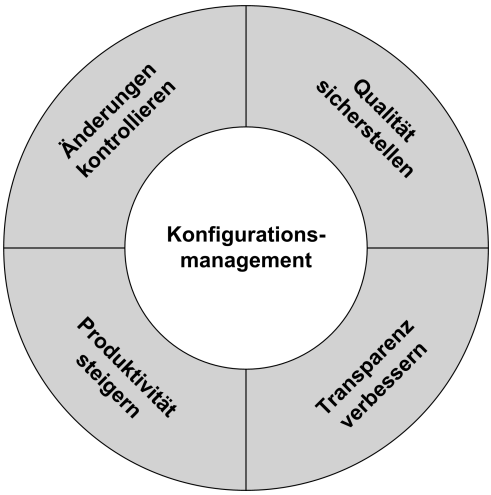


Abb. 2–1
Ziele des Konfigurations-managements

im selben Raum arbeiten. Es ist leicht nachvollziehbar, dass der Kommunikationsaufwand in einem Projekt steigt, je mehr Personen daran beteiligt sind. Unter der Annahme, dass jeder irgendwann mit jedem redet, wächst der Aufwand für Abstimmungen deutlich überproportional mit jedem neuen Teammitglied an (siehe Abb. 2–2). Die Folge ist, dass schon in Teams ab ca. vier Personen dringend notwendige Abstimmungen nicht oder nur noch unvollständig stattfinden können.

Vereinfachung der Kommunikation

Ein KM-Prozess verhindert diese Situation durch Vereinfachung der teaminternen Kommunikation. So wird beispielsweise Quelltext in einem zentralen *Repository* verwaltet, das über alle Änderungen Buch führt. Zusätzlich können die Änderungen mit kurzen Kommentaren versehen werden, die oft schon ausreichend sind, um direkte Rückfragen im Team zu vermeiden.

Das Repository wird ergänzt durch ein *Collaboration-Werkzeug*. Dieses umfasst typischerweise Module zum Änderungs- und Fehlermanagement und zur projektinternen Kommunikation (z. B. über ein Wiki). Gerade in größeren Teams helfen derartige Tools enorm, den Überblick zu behalten. Alle Aufgaben im Team werden darüber zentral erfasst und an die einzelnen Bearbeiter verteilt. Es ist also jederzeit klar, wer was macht. Und dies ohne zeitraubende und oft nutzlose Statusmeetings.

Neben der Vereinfachung der Kommunikation wird durch einen KM-Prozess sichergestellt, dass nur die »richtigen« Änderungen vorgenommen werden. Dies kann bedeuten, dass in bestimmten Projektphasen nur noch Bugfixes, aber keine funktionalen Erweiterungen zugelassen sind. Es finden also eine Auswahl und Priorisierung der durchzuführenden Änderungen statt.

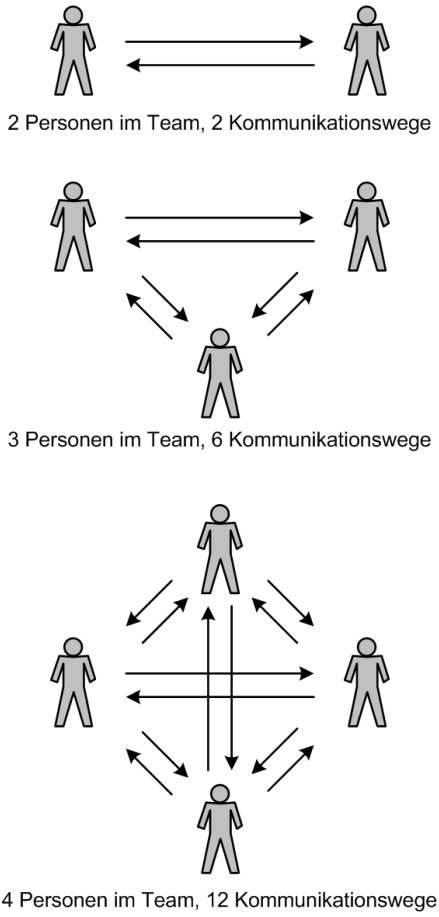


Abb. 2–2
Abstimmungsaufwand steigt überproportional (nach [Leon05]).

Etablierung einer Meritokratie

Ein weiteres Beispiel ist die Absicherung von Änderungen über die Etablierung einer *Meritokratie* im Projekt. In einer Meritokratie erhält jeder Einzelne umso mehr Einfluss und Rechte, je mehr er zum Ergebnis beiträgt. Bezogen auf die Softwareentwicklung bedeutet dies, dass ein erfahrener Entwickler in einem Projekt volle Schreibrechte auf alle Module bekommt, während ein Neueinsteiger zunächst nur einzelne, eher unkritische Quelltexte bearbeiten darf. In der Open-Source-Entwicklung sind meritokratisch organisierte Projekte weit verbreitet. Populäre Beispiele hierfür sind die Projekte der Apache Software Foundation [URL: ASFMeritokratie] und die Eclipse-Plattform. Technische Grundlage für die Umsetzung einer Meritokratie ist ein Repository als Teil des KM-Prozesses, mit dem die Rechte für einzelne Benutzer(-gruppen) unterschiedlich vergeben werden können. Wichtig ist zudem ein Collaboration-Werkzeug zur Verwaltung der anfallenden Aufgaben im Projekt. Auf diese Weise kann sich – ohne zentrale Steue-

rung – jeder interessierte Entwickler einer der offenen Aufgaben annehmen und so Schritt für Schritt mehr Bedeutung im Projekt gewinnen.

Nachvollziehbarkeit der Änderungen

Nicht zuletzt muss der KM-Prozess garantieren, dass *alle* am Produkt durchgeführten Änderungen auch in Zukunft nachvollzogen werden können. Insbesondere dürfen Änderungen auch unter sozusagen widrigen Umständen nicht verloren gehen. Dies betrifft beispielsweise parallele Änderungen am selben Element durch mehrere Bearbeiter. Ohne Kontrollmechanismen ist nicht vorhersagbar, in welchem Zustand das Element nach den Änderungen wirklich vorliegt. Ändern zwei Entwickler gleichzeitig dieselbe Quelltextdatei, ist unklar, welche Version erhalten bleibt und welche nicht. Abhängig von der Projektumgebung könnte dies z. B. nur die zuletzt gespeicherte Variante sein. In diesem Fall ginge die Arbeit desjenigen Entwicklers verloren, der zufällig als Erster auf *Speichern* geklickt hat. Diese Situation wird durch das Repository verhindert. Es erkennt parallele Änderungen und stellt sicher, dass keine Daten verloren gehen.

Qualität sicherstellen

Konfigurationsmanagement hilft uns, durch *Fehlervermeidung*, *Änderungsmanagement* und die *Projektautomatisierung* die Softwarequalität zu verbessern und das einmal erreichte Qualitätsniveau dauerhaft zu halten.

Fehlervermeidung

Die Fehlervermeidung umfasst rein präventive Maßnahmen. Durch eine Optimierung des Entwicklungsprozesses im Rahmen des Konfigurationsmanagements können rein prozessbedingte Fehler, die z. B. durch eine falsch konfigurierte Testumgebung verursacht werden, reduziert werden. Eine weitere Möglichkeit ist der Einsatz von Werkzeugen zur statischen Quelltextanalyse. Diese erzeugen Metriken, die Hinweise auf besonders fehlerträchtige Module liefern können (mehr dazu in Abschnitt 2.3.2).

Durchführung von Modultests

Die Durchführung automatisierter Modultests ist sicherlich eine der wirksamsten Maßnahmen zur Fehlervermeidung. Nur um Missverständnissen vorzubeugen: Die Erstellung der Modultests ist eine Kunst für sich[2] und keinesfalls Teil eines Konfigurationsmanagement-Prozesses. Das Konfigurationsmanagement sorgt lediglich dafür, dass die vorhandenen Tests regelmäßig ausgeführt und die Testergebnisse automatisch ausgewertet werden.

2. Zum Thema Unit-Tests und testgetriebene Entwicklung gibt es sehr gute Literatur, z. B. [Beck02]. Speziell für die Programmiersprache Java ist auch [Link05] zu empfehlen.

Änderungsmanagement

Trotz aller Sorgfalt lassen sich Fehler und nachträgliche Änderungen in der erstellten Software nicht vermeiden. Das Änderungsmanagement hat daher die Aufgabe, bekannte Fehler und Änderungsanforderungen zu dokumentieren, zu bewerten und zu priorisieren. Ohne ein funktionierendes Änderungsmanagement hat man keinen Überblick, wie der aktuelle Status des Projektes ist. Zudem entfällt in diesem Fall die Filterfunktion des Änderungsmanagements, d. h., jede Änderungsanforderung schlägt schlimmstenfalls direkt zum Entwicklungsteam durch. Erfahrungsgemäß sind viele Änderungsanforderungen und Fehlermeldungen unvollständig bzw. schon bekannt. Ein KM-Prozess schützt ein Team von Entwicklern vor der unnötigen Mehrbelastung, sich mit den Erstellern ungültiger Tickets selbst auseinanderzusetzen.

Projektautomatisierung

Ein weiterer Aspekt der dauerhaften Qualitätssicherung ist die Möglichkeit, das Softwareprodukt jederzeit zuverlässig und wiederholbar neu zu erstellen. Voraussetzung hierfür ist eine leistungsfähige Projektautomatisierung als Teil des KM-Prozesses. Beispielsweise stellen Build-Skripte sicher, dass das Produkt immer auf dieselbe Weise aus den Quelltexten neu kompiliert und zur Auslieferung vorbereitet wird.

Produktivität steigern

Eine höhere Produktivität erreicht man am einfachsten, indem die Teammitglieder sich wirklich auf ihre jeweiligen Aufgaben konzentrieren können. Alle anderen Tätigkeiten müssen demzufolge so weit wie möglich reduziert werden.

Je nach Rolle in einem Projekt unterstützt ein KM-Prozess dieses Vorhaben auf unterschiedliche Weise. So kann Analysten z. B. einfach dadurch geholfen werden, dass zusammengehörige Anforderungsdokumente schon am Dateinamen zu erkennen sind. Dies erspart das wiederholte, zeitintensive Durchstöbern der Projektstruktur.

Entwickler profitieren insbesondere von den zur Umsetzung des KM-Prozesses verwendeten Werkzeugen, wie beispielsweise dem Repository zur Versionsverwaltung. Dieses erlaubt den schnellen Vergleich zweier Versionen einer Datei und verhindert dadurch zeitraubende Rückfragen im Team.

Transparenz verbessern

Insbesondere das Management leidet oft unter der Tatsache, dass Software ein »unsichtbares Produkt« ist. De facto liegt in vielen Projekten die Wahrheit über ein System ausschließlich im Code – und den kennen meist nur die Entwickler. Vielen Projektleitern ist daher nicht klar, wo sie wirklich stehen, da sie nur sehr vage Aussagen von den Entwicklern erhalten (»Bin beinahe fertig«).

Dieser Zustand ist eigentlich nicht zu akzeptieren, denn ohne effektive Kontrolle geht es auch in der Softwareentwicklung nicht. Das Konfigurationsmanagement kann hier zumindest einen Teil der notwendigen Transparenz herstellen. Eine wichtige Rolle spielt hier insbesondere das Werkzeug für das Änderungsmanagement. Mit Hilfe von Berichten kann sich die Projektleitung jederzeit über die noch offenen, aktuell bearbeiteten und bereits abgeschlossenen Änderungsanforderungen bzw. Fehler informieren.

Projekt-Homepages

Eine weitere Hilfestellung zur Verbesserung der Transparenz sind sogenannte Projekt-Homepages. Diese erlauben beispielsweise den Einblick in die automatisch ermittelten Metriken und Ergebnisse der Testläufe.

2.1.2 Argumente für den Einsatz im Projekt

Nachdem geklärt ist, was unter dem Begriff Konfigurationsmanagement zu verstehen ist, stellt sich die Frage, wer Konfigurationsmanagement warum einsetzen sollte. Die Antwort könnte nach der Lektüre der obigen Abschnitte lauten: jeder, der mit mindestens einer weiteren Person im Team zusammenarbeitet. In der Praxis kommt man mit dieser zwar simplen, aber durchaus korrekten Argumentation allerdings nicht weit. Denn leider fristet das Thema Konfigurationsmanagement in vielen Softwareentwicklungsprojekten ein ausgesprochenes Schattendasein.

Der Kampf um Zeit und Geld

Bei genauer Betrachtung geht es um die Frage, wie bedeutend ein KM-Prozess für den Projekterfolg ist. Nach meiner Erfahrung herrscht oft die Meinung vor, dass die kontrollierte Durchführung von Änderungen, automatisierte Skripte und Fehlermanagement letztlich Details sind, die, wie viele andere Aufgaben auch, quasi nebenher erledigt werden. In nahezu allen Plänen, mit denen ich als Architekt bisher bei Start eines Projekts konfrontiert wurde, tauchten diese Punkte nicht oder nur in minimalem Umfang auf. Projektstrukturen und Build-Skripte entstehen aber nicht »einfach so«, es ist durchaus angebracht, hierfür etwas Zeit und Geld einzufordern. Will man Konfigurations-

management in einem Projekt einsetzen, ist also in der Regel Überzeugungsarbeit zu leisten.

Vorteile des Konfigurationsmanagements

In dieser Situation sprechen die oben beschriebenen Ziele des Konfigurationsmanagements zunächst für sich. Niemand wird die Notwendigkeit eines Repositorys als Grundlage für die Arbeit eines Projektteams wirklich in Frage stellen, wenn er mit den potenziellen negativen Auswirkungen im Falle eines Verzichts darauf konfrontiert wird. Wichtig ist es, in diesem Fall darauf hinzuweisen, dass es keinesfalls nur auf die Auswahl eines geeigneten Werkzeuges ankommt. Das Tool bildet nur die Grundlage für einen Teil des KM-Prozesses. Zusätzlich sind Richtlinien für den Umgang mit dem Repository im Projekt notwendig. Wie wir später noch sehen werden, ist es beispielsweise sinnvoll, in manchen Bereichen eines Repositorys nur lesende Zugriffe zu erlauben.

Die Steigerung der Produktivität sollte der Leitung eines Projektes selbst am Herzen liegen, daher ist dieser Aspekt eines KM-Prozesses meist relativ einfach zu vermitteln. Auch dass hierfür Maßnahmen wie automatisierte Skripte notwendig sind, ist einsichtig.

Frühe Etablierung des Änderungsmanagements

Problematischer sind meiner Erfahrung nach oft die Teilbereiche des KM-Prozesses, die auf die Verbesserung der Qualität und der Transparenz abzielen. Fehlervermeidung und Änderungsmanagement scheinen beim Start eines Projekts noch sehr weit entfernt zu sein. Schließlich will man sich nicht schon über Änderungsanforderungen Gedanken machen, bevor die erste Zeile eines Use Cases geschrieben worden ist. Tatsächlich sollten allerdings sowohl die Fehlervermeidung als auch das Änderungsmanagement so früh wie möglich im Projekt etabliert werden. Setzt man beispielsweise automatisiert ermittelte Metriken zur Fehlervermeidung ein, müssen die Ergebnisse unbedingt von der ersten Quelltextzeile an im Team verteilt werden. Lässt man Metriken auf ein quasi fertiges Produkt los, erntet man unter Garantie Hunderte von Fehlermeldungen. Die Aufregung und der Ärger im Projekt sind dann groß, und eine inhaltliche Diskussion der Messergebnisse wird entsprechend schwierig. Beim Änderungsmanagement spricht eher der direkte Nutzen im Projektalltag für eine frühe Einführung. Neben Fehlern und Änderungsanforderungen können, wie bereits erwähnt, auch die Aufgabenpakete für das Entwicklerteam prima in den entsprechenden Werkzeugen verwaltet werden.

Zahlen, Zahlen, Zahlen ...

Oft kann man Argument an Argument reihen, so richtig kommt das alles erst an, wenn konkrete Zahlen genannt werden. So verhält es sich auch beim Thema Softwarefehler. Dankenswerterweise haben Barry Boehm und Victor Basili in [Boehm01] eine Top-10-Liste der wichtigsten Aussagen und Metriken zum Thema Softwarefehler veröffentlicht. Unter anderem stellen die Autoren Folgendes fest:

- Softwarefehler, die erst spät im Lebenszyklus einer Software – also z. B. nach der Auslieferung – gefunden werden, verursachen bis zu 100-mal mehr Kosten als solche, die frühzeitig entdeckt werden. Dies spricht für den konsequenten Einsatz von Techniken zur Fehlervermeidung, insbesondere auch in den frühen Phasen eines Projektes.
- 40 bis 50 Prozent des Gesamtaufwandes eines Softwareprojekts entstehen durch eigentlich vermeidbare Nacharbeit. Eine deutliche Reduktion dieses Aufwandes kann vor allem durch Verbesserungen in den Bereichen Entwicklungsprozess, Softwarearchitektur und Risikomanagement erreicht werden.
- 80 % der vermeidbaren Nacharbeit werden von nur 20 % der Fehler verursacht. Fehlermanagement kann helfen, die kritischen Bereiche einer Anwendung zu identifizieren. Diese können dann einem Refactoring unterzogen werden.
- 80 % der Fehler treten in nur 20 % der Module oder Komponenten auf. Über die Hälfte aller Module und Komponenten ist fehlerfrei. Dieser Tatsache sollte man z. B. durch automatisierte Codeanalysen Rechnung tragen. Diese Analysen geben schon früh im Lebenszyklus eines Projektes wertvolle Hinweise auf potenziell fehleranfällige Module.
- 90 % aller Ausfälle eines Systems werden von nur 10 % der Fehler verursacht. Es ist offensichtlich, dass der komplette Stillstand eines produktiven Systems zu den schwerwiegendsten Fehlerszenarien gehört. Jeder einzelne frühzeitig identifizierte oder vermiedene Fehler aus diesen kritischen 10 % rechtfertigt schon für sich die Maßnahmen zur Fehlervermeidung in einem Projekt.

Schrittweise Verbesserung der Transparenz

Die Herstellung einer gewissen Transparenz im Projekt mit Hilfe eines Collaboration-Werkzeuges und eventuell noch einer Projekt-Homepage ist in den letzten Jahren erfreulich einfach geworden. Viele Projektleiter haben mittlerweile Erfahrungen mit derartigen Werkzeugen gesammelt und die Vorzüge schätzen gelernt. Problematisch ist eventuell nur die Auswahl des richtigen Tools. Viele Hersteller buhlen um den Kuchen, und die oft hohen Lizenzkosten professioneller Tools erlauben den Einsatz ganzer Heerscharen von Vertriebsmitarbeitern. Lassen Sie sich davon nicht verunsichern. Entscheidend ist auch bei einem Collaboration-Werkzeug ausschließlich, ob es den – meist sehr überschaubaren – Anforderungen des Projektes genügt. Beziehen Sie daher unbedingt sowohl Open-Source-Werkzeuge als auch Produkte von spezialisierten Herstellern in die Auswahl mit ein.

Konfigurationsmanagement ist kein zusätzlicher Aufwand.

Zusammenfassend lässt sich feststellen, dass Konfigurationsmanagement letztendlich keinen zusätzlichen Aufwand für ein Projekt darstellt. Nahezu alle Tätigkeiten im Rahmen eines KM-Prozesses müssen früher oder später sowieso durchgeführt werden. Niemand kommt darum herum, die wesentlichen Elemente eines Softwareprojektes in einer passenden Struktur anzuordnen und diese in einem Repository zu verwalten. Ähnlich verhält es sich mit der Erstellung von Build-Skripten und der Verwaltung von Aufgaben, Änderungen und Fehlerberichten. Vielmehr spart man Zeit und Geld, wenn Diskussionen zu diesen Themen rechtzeitig und sorgfältig geführt werden.

2.1.3 Normen und Standards

Die Grundzüge des Konfigurationsmanagements wurden und werden von diversen nationalen und internationalen Organisationen in Standards festgehalten. Der erste Konfigurationsmanagement-Standard AFSCM-375-1 wurde bereits 1963 vom US-Militär entwickelt. Dieser Standard bezog sich ausschließlich auf die Entwicklung von Hardware. Erst 1971 wurde auch Software explizit in den – ebenfalls vom US-Militär initiierten – Standard MIL-STD-483 aufgenommen. Laut [Berczuk02] sind mittlerweile über 200 KM-Standards veröffentlicht worden.

Für uns sind Normen und Standards nur von eingeschränktem Wert. Es liegt in der Natur der Sache, dass sich Standards meist auf einem hohen Abstraktionsniveau bewegen, schließlich sollen möglichst viele Einsatzszenarien berücksichtigt werden. Hinzu kommt, dass Standards aus demselben Grund oft umständlich beschrieben und schwer verständlich sind.

Im Folgenden habe ich drei international anerkannte Standards zum Thema Konfigurationsmanagement aufgezählt, die trotz der genannten Einschränkungen für Sie interessant sein könnten. Dies trifft insbesondere dann zu, wenn Konfigurationsmanagement für Sie nicht nur im Rahmen eines konkreten Projektes relevant ist, sondern auch auf einer übergeordneten, strategischen Ebene. Beispielsweise ist ein etablierter KM-Prozess im Unternehmen Voraussetzung zur Erreichung des CMMI[3] Reifegrads 2. Zusätzlich zu den Standards sollten Sie in diesem Fall allerdings auch weiterführende Literatur zurate ziehen (z. B. [Berczuk02]).

---

3. Das *Capability Maturity Model for Development* [URL: CMMI] ist ein Referenzmodell für Entwicklungsprozesse. Zudem erlaubt es die Beurteilung des Reifegrades existierender Prozesse. Der höchstmögliche Reifegrad ist 5.

- *ISO 10007:2003* [ISO-10007-2003]
  Diese ISO-Norm zum Thema Konfigurationsmanagement ist nicht auf die Softwareerstellung beschränkt, sondern umfasst jegliche Produktentwicklung. Der Aspekt des Managements eines Entwicklungsprozesses wird in der Norm in den Vordergrund gestellt. KM hat demzufolge sicherzustellen, dass der Status und die Konfiguration des entwickelten Produktes jederzeit vollständig dokumentiert und einsehbar sind. Zudem muss durch das KM gewährleistet werden, dass die Anforderungen an das Produkt vollständig erfüllt werden. Die Norm bietet einen guten Überblick, was ganz allgemein unter Konfigurationsmanagement zu verstehen ist.
- *ISO 12207:1995* [ISO-12207-1995]
  Die ISO 12207 definiert einen sogenannten *Software-Lifecycle-Prozess*. Prozesse dieser Art beschreiben den Werdegang eines Softwareproduktes, vom Entwurf über die Umsetzung bis hin zur Wartung. Die Norm kann nicht direkt in einem Projekt umgesetzt werden, sie legt aber einen allgemein akzeptierten Rahmen für derartige Prozesse fest. Insgesamt beinhaltet dieser Rahmen fünf Hauptprozesse (z. B. Entwicklung und Betrieb), acht Hilfsprozesse und vier organisatorische Prozesse (z. B. Management und Training). Konfigurationsmanagement ist einer der acht Hilfsprozesse. Interessant an der Norm ist die Einbindung des Konfigurationsmanagements in den gesamten Lebenszyklus eines Produktes. Für konkrete Projekte ist es jedoch hilfreicher, wenn man beispielsweise ein Vorgehensmodell einsetzt, das die Vorgaben der Norm praxisgerecht umsetzt.
- *IEEE Std 828-2005* [IEEE-828-2005]
  In dieser IEEE-Norm werden der Aufbau und Inhalt eines *Konfigurationsmanagement-Plans* beschrieben. Im Gegensatz zu den bisher genannten Standards kann man hier direkte Anregungen für ein Projekt finden. Auf diesen Standard werde ich in Abschnitt 2.2.2 bei der Erläuterung des Konfigurationsmanagement-Handbuchs zurückkommen.

2.2 Aufgaben und Verfahren des Kernprozesses

2.2.1 Auswahl der Konfigurationselemente

Die Auswahl der Konfigurationselemente ist der erste Schritt zur Einführung eines KM-Prozesses in einem Projekt. Die IEEE definiert ein Konfigurationselement wie folgt: »*A software configuration item (SCI) is an aggregation of software designated for configuration mana-*

gement and is treated as a single entity in the SCM process« [URL: SWEBOK]. Dieser Satz ist ein schönes Beispiel dafür, dass ab einem gewissen Abstraktionsniveau jeder Inhalt verloren geht. Folgen wir der Definition, kann ein Konfigurationselement vom kompletten Subsystem bis hin zur einzelnen Datei alles umfassen.

Konfigurationselement als Klasse

Damit kommen wir nicht weiter, daher verwende ich im Folgenden den Begriff *Konfigurationselement* analog zum Konzept der *Klasse* in objektorientierten Sprachen. Das Element legt demnach Typ und Eigenschaften einer zusammengehörenden Gruppe von Artefakten in einem Projekt fest. So würde z. B. das Element *Quelltext* alle einzelnen Quelltextdateien umfassen. Wie wir später noch sehen werden, ist insbesondere das zu verwendende Namensschema eine wichtige Eigenschaft jedes Konfigurationselements. Für die einzelnen Dateien des Elementes *Quelltext* könnte man z. B. unterschiedliche Pre- oder Postfixes für Interfaces und Klassen definieren.

Produkt

Alle Instanzen der Konfigurationselemente zusammen ergeben das *Produkt*, das man im Zuge des Projektes erstellen will. Anders ausgedrückt sind alle Bestandteile eines Projektes, die nicht dem Produkt zugeordnet werden können, keine Konfigurationselemente. Beispiele hierfür sind z. B. Listen und Pläne, die nur als Hilfsmittel zur Durchführung des Projektes an sich dienen.

Durch den Auswahlprozess wird festgelegt, welche Dokumente, Dateien und sonstige Bestandteile eines Projekts unter das Konfigurationsmanagement fallen sollen und welche nicht. Gleichzeitig wird versucht, die ausgewählten Elemente sinnvoll zu strukturieren. Vermutlich wird man bei der erstmaligen Auswahl der Elemente zum Start eines Projektes nur einen ersten Wurf hinbekommen. Daher sollte die Auswahl nicht als eine einmalige Tätigkeit betrachtet werden. Vielmehr gilt es, bei jeder Einführung neuer Elemente in das laufende Projekt zu überprüfen, ob diese für das Konfigurationsmanagement relevant sind oder nicht.

Sorgfältige Auswahl der Elemente

Man kann die Wichtigkeit einer sorgfältigen Auswahl der Konfigurationselemente nicht überbetonen. Vergessen wir, wichtige Teile des Produktes mit ins Konfigurationsmanagement aufzunehmen, kann später eine vollständige Auslieferung desselben nicht garantiert werden. Ein beliebtes Beispiel hierfür sind Build-Skripte. Wird zwar der Quelltext als Konfigurationselement identifiziert, nicht aber die Build-Skripte, bekommt man früher oder später ein Problem. Dann nämlich, wenn eine ältere Version des Systems erstellt werden soll, das hierfür notwendige Skript aber nicht mehr verfügbar ist.

Auf der anderen Seite wollen wir auch nicht überflüssige oder redundante Informationen in den KM-Prozess aufnehmen. Zwar scha-

den zu viele Elemente nicht in dem Maße wie zu wenige, aber Redundanz erzeugt potenziell unnötigen Aufwand. Und genau den wollen wir mit einem KM-Prozess ja eigentlich vermeiden. Die Entscheidung, welche Elemente wirklich aus dem Konfigurationsmanagement ausgeschlossen werden sollen, ist jedoch schwierig.

Ein Beispiel für ein fragwürdiges KM-Element ist die (hoffentlich) generierte Quelltextdokumentation. Im Java-Umfeld verwendet man zu diesem Zweck üblicherweise das JavaDoc-Tool. Dieses erstellt aus den Kommentaren in den Quelltextdateien gut lesbare HTML- oder PDF-Dokumente. Nimmt man diese Dokumente in die Liste der Konfigurationselemente auf, wird eine unnötige Redundanz verursacht. Denn die aktuellste Version der Dokumentation kann jederzeit aus dem Quelltext neu erzeugt werden.

Der Preis ist allerdings eine geringere Verfügbarkeit, denn die Generierung kostet schließlich Zeit. In großen Projekten mit mehreren tausend Quelltextdateien kann die Generierung sogar so lange dauern, dass die fertige Dokumentation zwingend als Konfigurationselement ausgewählt werden muss. Eine Ad-hoc-Erzeugung durch jedes einzelne Teammitglied wäre in diesem Szenario nicht mehr praktikabel. Allerdings muss man sich in der Folge Gedanken machen, wie die Aktualität der Dokumentation sichergestellt werden kann. Theoretisch müsste der KM-Prozess dafür sorgen, dass nach jeder Änderung am Quelltext die Dokumentation neu erzeugt und dem Team zur Verfügung gestellt wird. In der Praxis ist dieses Vorhaben schwierig umzusetzen, schließlich wird in größeren Teams andauernd am Quelltext gearbeitet. Ein möglicher Mittelweg ist, die Dokumentation *näherungsweise* aktuell zu halten. Hierzu kann die Generierung z. B. im Rahmen eines automatisierten Integrations-Builds erfolgen.

Eindeutige Konfigurationselemente

Nicht alle Ergebnistypen in einem Projekt machen uns die Entscheidung für die Aufnahme in die Liste der Konfigurationselemente so schwer wie die Quelltextdokumentation im obigen Beispiel. Die folgende Liste gibt einen Überblick über Konfigurationselemente, die für die Entwicklung von Software eigentlich fast immer notwendig sind:

- Quelltext
- Anforderungsdokumente (z. B. Use Cases)
- Architektur- und Designdokumente
- Konfigurationsmanagement-Handbuch
- Schnittstellenverträge
- Testspezifikationen und Testdaten

- Build-Skripte
- Meta- und Konfigurationsdaten
- Benutzerdokumentation
- Installationsanleitung, Release-Notes etc.

Weitere Detaillierung der Elemente

Diese Aufzählung der eindeutigen Konfigurationselemente ist zugegebenermaßen recht grobgranular. Will man mehr ins Detail gehen, könnte z. B. das Element *Quelltext* weiter in *Komponente* und *Schnittstelle* unterteilt werden. Dadurch ist es später möglich, die Abhängigkeiten zwischen Komponenten über öffentliche Schnittstellen explizit im Konfigurationsmanagement abzubilden. Zusätzlich können unterschiedliche Regeln für Änderungen an Komponenten und deren öffentlichen Schnittstellen im Projekt etabliert werden.

Wie detailliert die Konfigurationselemente ausgewählt werden, ist aus meiner Sicht wesentlich vom verwendeten Vorgehensmodell und den Vorgaben der Softwarearchitektur abhängig. Existiert beispielsweise schon beim Start des Projektes eine Standardarchitektur, welche die grobe Struktur des Systems und die wichtigsten Komponententypen festlegt, wird man diese auch in der Liste der Konfigurationselemente abbilden. In diesem Fall wäre jeder Komponententyp ein eigenes Konfigurationselement. Ähnliches gilt für die laut Vorgehensmodell zu liefernden Ergebnisse. Schreibt das Vorgehensmodell z. B. die Erstellung von Use Cases und GUI-Prototypen vor, sollten auch entsprechende Konfigurationselemente festgelegt werden.

Mögliche Konfigurationselemente

Die folgenden Beispiele sind typische »Wackelkandidaten«, die erst nach einer sorgfältigen Abwägung als Konfigurationselemente festgelegt werden sollten:

- Werkzeuge, wie z. B. Entwicklungsumgebung, Compiler und Build-Tools
- Bibliotheken und Frameworks (speziell hierfür bietet übrigens das Tool *Maven* eine sehr elegante Unterstützung, die wir in Kapitel 5 näher kennenlernen werden)
- bestimmte generierte Artefakte

Gründe für den Ausschluss von Elementen

Gegen die obigen Kandidaten spricht entweder die physikalische Größe der Elemente oder die Einführung von Redundanz. Letzteres habe ich weiter oben schon am Beispiel der Quelltextdokumentation erläutert. Es bleibt uns noch, die Größe zu diskutieren. Und diese ist in der Tat ernst zu nehmen. Installationspakete gängiger Entwicklungsumgebungen erreichen locker mehrere 100 MB Umfang.

Werden Entwicklungswerkzeuge wegen der Größe nicht als Konfigurationselement definiert und aus dem Repository ausgeschlossen, sollte man darüber nachdenken, einen Spezialfall einzuführen. Die eingesetzten Entwicklungswerkzeuge werden im Laufe eines Projektes nur wenige Male erneuert oder ausgetauscht, sie unterscheiden sich also nicht nur in puncto Größe von anderen Konfigurationselementen, sondern auch in der Änderungshäufigkeit.

Manuelle Verwaltung von Konfigurationselementen

Ein praktikabler Kompromiss ist die manuelle Verwaltung der Werkzeuge. Geeignet hierfür ist jedes Netzwerklaufwerk, die Versionierung der Werkzeuge wird über eine entsprechende Ordnerstruktur sichergestellt. Auch eine Archivierung auf Blu-Ray-Disks ist denkbar, z. B. wenn aus lizenzrechtlichen Gründen eine zentrale Ablage nicht in Frage kommt. Ein Problem der manuellen Vorgehensweise ist allerdings, dass die Werkzeuge nicht mehr automatisch über das Versionskontrollsystem in ein Release aufgenommen werden können. Abhilfe schafft ein expliziter Verweis auf die eingesetzten Werkzeuge und deren Version in der Dokumentation jedes Release.

Die Frage, ob die oben genannten Konfigurationselemente sinnvoll sind, hängt vom Projekt und der Umgebung ab. Sind in einem Unternehmen z. B. die erlaubten Bibliotheken und Frameworks standardisiert, werden diese in der Regel sowieso außerhalb des Projektes archiviert. In diesem Fall genügt es, auf diese zentrale Ablage zu verweisen.

Keine Konfigurationselemente

Eine Reihe von Artefakten im Projekt sind keine Konfigurationselemente, auch wenn sie für die Projektabwicklung an sich durchaus wichtig sind. Darunter fallen z. B.:

- Protokolle von Meetings
- binäre Auslieferungsdateien
- Generierte Dateien, insbesondere die kompilierten Quelltexte. Ausnahmen sind hier möglich, wie beispielsweise die bereits erwähnte JavaDoc-Dokumentation.
- Projektpläne
- Liste offener Punkte, Risikolisten etc.

Konfigurationsmanagement umfasst alle Elemente, die zur Erstellung des Softwareproduktes notwendig sind oder dieses Produkt beschreiben. Die Projektablage enthält hingegen zusätzlich alle Dateien, die zur Steuerung und Durchführung des Projektes nötig sind. Ein Projektplan z. B. ist, wenn er denn funktionieren soll, ein lebendes, häufig geändertes Dokument. Es hat nicht wirklich Sinn, einen alten Stand des Planes wiederherzustellen – außer vielleicht man betreibt Vergangenheitsbe-

wältigung in einem schwierigen Projekt. Ähnlich verhält es sich mit Protokollen zu Meetings. Fallen in einem Meeting wichtige Entscheidungen, müssen diese sowieso in die entsprechenden Anforderungs- und Designdokumente eingearbeitet werden.

Umgang mit binären Auslieferungsdateien

Ich habe auch die binären Auslieferungsdateien aus den Konfigurationselementen ausgeschlossen, obwohl dies nicht immer auf Zustimmung stoßen dürfte. Tatsächlich wird man die Auslieferungsdateien in der Regel zumindest eine Zeit lang archivieren. Dies ist alleine schon deshalb sinnvoll, falls nach einer neuen Auslieferung wirklich der GAU eintreten sollte und nichts mehr funktioniert. Es ist in diesem Fall sehr beruhigend, wenn zumindest das alte Release schnell und unkompliziert zur Hand ist. Diese Archivierung ist aber nur temporär notwendig. In der Regel kann man die fertigen Installationspakete alter Releases nach einer Übergangszeit gefahrlos verwerfen. Schließlich können sie dank des Konfigurationsmanagements bei Bedarf wieder neu generiert werden.

2.2.2 Erstellung des Konfigurationsmanagement-Handbuches

Sobald die Liste der Konfigurationselemente festgelegt ist, gilt es, diese zu dokumentieren. An dieser Stelle kommt der *Konfigurationsmanagement-Plan* ins Spiel. Alle Entscheidungen, die den KM-Prozess in einem Projekt betreffen, werden in diesem Plan dokumentiert. Laut IEEE [IEEE-828-1998] sollte ein KM-Plan aus den folgenden sechs Abschnitten bestehen:

- *Einleitung*
  Erläutert die Ziele des Plans und der wichtigsten Begriffe
- *Management*
  Beschreibt, wer für welche KM-Aktivitäten verantwortlich ist und wie die Einbindung anderer Organisationseinheiten erfolgt
- *Aktivitäten*
  Dokumentiert die zentralen Aufgaben und Verfahren des Konfigurationsmanagements
- *Zeitplanung*
  Legt die zeitliche Abfolge der KM-Aktivitäten fest
- *Ressourcen*
  Dokumentiert den Bedarf an Werkzeugen und Personal
- *Pflege*
  Beschreibt, in welcher Form der Plan während eines laufenden Projektes geändert werden kann, z. B. um auf geänderte Anforderungen zu reagieren

KM-Handbuch als Teilmenge des KM-Plans

Für den von uns verfolgten, leichtgewichtigen KM-Ansatz ist nur ein Teil der genannten Punkte interessant. Konkret werden wir lediglich den Abschnitt *Aktivitäten* und Teile des Abschnitts *Ressourcen* benötigen. Um Missverständnisse zu vermeiden, habe ich daher entschieden, den Begriff Konfigurationsmanagement-Plan im weiteren Verlauf des Buches nicht zu verwenden, und das zu erstellende Dokument *Konfigurationsmanagement-Handbuch* genannt.

Notwendigkeit des KM-Handbuches

Die Einschränkung des KM-Handbuchs auf nur einen Teil der möglichen Inhalte wirft die Frage auf, ob man nicht auch ganz auf ein solches Dokument verzichten könnte. Schließlich sind die dem KM-Prozess zugrunde liegenden Werkzeuge gut dokumentiert. Darüber hinaus ist es unser erklärtes Ziel, Formalismen weitgehend zu vermeiden.

Trotzdem ist nach meiner Auffassung ein KM-Handbuch unverzichtbar. Tatsächlich wird in den meisten Projekten ein Sammelsurium an Beschreibungen existieren, die zusammengenommen nichts anderes als das KM-Handbuch darstellen. Dies umfasst z. B. Namenskonventionen, Richtlinien zur Codeformatierung oder Installationsanleitungen für die verwendeten Tools. Hinzu kommt, dass der Aufwand für die Erstellung des Handbuchs nur im ersten Projekt vollständig anfällt. Große Teile des KM-Handbuchs kann und sollte man in weiteren Projekten wiederverwenden. Dies erleichtert auch die Einarbeitung eines Teams in einem neuen Projektumfeld.

Gliederung des KM-Handbuches

Die Tabelle 2–1 enthält eine meines Erachtens sinnvolle Gliederung für ein Konfigurationsmanagement-Handbuch. Das Handbuch umfasst wesentlich mehr Punkte als die reine Dokumentation der Konfigurationselemente. Daher verwende ich in der Tabelle eine Reihe von Begriffen, die erst in den noch folgenden Kapiteln erläutert werden. Da ich in der Tabelle aus Platzgründen jeden Gliederungspunkt nur in aller Kürze erläutern kann, finden Sie auf der Webseite zum Buch eine ausführlich kommentierte Beispielgliederung als Word-Datei.

Kapitel		Beschreibung
1	**Einleitung**	
1.1	Inhalt dieses Dokuments	Kurze Zusammenfassung des Inhalts. In vielen Projekten wird eine derartige »Management Summary« am Beginn jedes Dokuments gefordert, im Prinzip kann man aber auch darauf verzichten
1.2	Leserkreis	Wer sollte welche Kapitel lesen?
1.3	Projekt-Homepage	Verweis auf die Projekt-Homepage mit zusätzlichen Informationen. Wenn auf der Projekt-Homepage kein Verzeichnis weiterer wichtiger Dokumente im Projekt existiert, sollte an dieser Stelle eine entsprechende Tabelle eingefügt werden.
2	**Konfigurationselemente**	
2.1	Übersicht	Tabellarische Übersicht aller Konfigurationselemente
2.2	KE: *<Name>*	Pro Element ein Kapitel mit der detaillierten Beschreibung (Näheres hierzu gleich im Anschluss an die Tabelle)
3	**Projektumgebung**	
3.1	Repository-Struktur	Dokumentation der Repository-Struktur
3.2	Verzeichnisstruktur des Arbeitsbereiches	Beschreibung des lokalen Arbeitsbereiches
3.3	Werkzeuge	Tabellarische Übersicht der im Projekt verwendeten Werkzeuge. Anschließend ein Unterkapitel pro Werkzeug mit Anleitungen zur Installation und Konfiguration.
3.4	Externe Komponenten	Wie 3.3, diesmal werden jedoch die externen Komponenten beschrieben. Hierunter fallen beispielsweise zugekaufte Bibliotheken zur Realisierung bestimmter Funktionen im Produkt.
4	**Verwaltung der Konfigurationselemente**	
4.1	Erstmaliger Check-out des Arbeitsbereiches	Beschreibt die Schritte zum erstmaligen Check-out des Arbeitsbereiches aus dem Repository
4.2	Hinzufügen, Ändern und Löschen von Dateien	Dokumentiert den Änderungszyklus im Projekt
4.3	Durchführung von strukturellen Änderungen	Legt fest, unter welchen Bedingungen die Projektstruktur geändert werden darf. Dies umfasst beispielsweise auch das Umbenennen oder Verschieben von Dateien.
4.4	Sperren von Dateien	Beschreibt den Umgang mit dem Sperrmechanismus des Repositorys. Hierunter fällt zum Beispiel eine Liste aller Dateitypen, die nur nach dem Setzen einer Sperre bearbeitet werden dürfen. Zudem sollten die projektspezifischen Regelungen beim Einsatz von Sperren festgehalten werden (z. B. der Umgang mit Sperren, wenn Branches aktiv sind, oder die zwingende Freigabe von Sperren vor dem Wochenende bzw. dem Urlaubsbeginn).
4.5	Erstellung von Tags	Tags dürfen in der Regel nur unter bestimmten Bedingungen und von einem eingeschränkten Personenkreis erstellt werden. Hier wird beschrieben, welche Einschränkungen im Projekt gelten.
4.6	Erstellung von Branches	Wie oben, nur für die Erstellung und Verwaltung von Branches

→

Kapitel		Beschreibung
5	**Änderungs- und Fehlermanagement**	
5.1	Rollen, Rechte und Pflichten	Für das Änderungs- und Fehlermanagement müssen meist spezielle Rollen im Projekt eingerichtet werden (z. B. ein Änderungsmanager).
5.2	Aufbau des CCB	Beschreibt die Zusammensetzung und Aufgaben des CCB (Change Control Board)
5.3	Erfassung, Bewertung und Bearbeitung von Tickets	Beschreibt den Prozess zum Umgang mit Tickets
6	**Releasemanagement**	
6.1	Releaseplan	Tabellarische und grafische Darstellung des Releaseplans
6.2	Auslieferung eines regulären Release	Beschreibt den Prozess zur Auslieferung eines regulären Release. In diesem Kapitel wird sowohl die Vorbereitungsphase als auch die Freigabe und Erstellung des Release dokumentiert.
6.3	Auslieferung eines Patches	Beschreibt den »Notfall-Prozess« zur Auslieferung eines Patches für kritische Fehler
7	**Audits, Metriken und Berichte**	
7.1	Audit-Plan	Übersicht der manuellen und automatisierten Audits im Projekt. Pro Audit müssen minimal der Teilnehmerkreis, der Zeitpunkt (z. B. einmal pro Monat) und das genaue Ziel des Audits festgelegt werden. Notwendig ist außerdem ein Plan, wie mit fehlgeschlagenen Audits umgegangen wird.
7.2	Manuelle Metriken	Beschreibt die manuell ermittelten Metriken im Projekt
7.3	Automatisierte Metriken	Dokumentiert die automatisiert ermittelten Metriken. Speziell für die automatisierten Messungen werden auch die Ausnahmen festgehalten. (Welche Module werden aus welchen Gründen von der Messung ausgeschlossen?)
7.4	Veröffentlichte Berichte	Tabellarische Übersicht der auf der Projekt-Homepage veröffentlichten Berichte. Hierzu gehört auch eine Interpretationshilfe (z. B.: Welche Bedeutung haben fehlgeschlagene Modultests?).

Tab. 2–1
Gliederung des KM-Handbuchs

2.2.3 Beschreibung der Konfigurationselemente

Die Dokumentation der identifizierten Konfigurationselemente im KM-Handbuch besteht pro Element aus minimal zwei Angaben: aus einer kurzen inhaltlichen Beschreibung und den Namenstemplates. Wie wir gleich sehen werden, ist es durchaus sinnvoll, noch weitere Eigenschaften eines Elements festzuhalten, doch beginnen wir zunächst mit den unverzichtbaren Bestandteilen der Dokumentation.

Die inhaltliche Beschreibung sollte nur aus ein bis zwei Sätzen bestehen. Ziel ist es, dass jedes Teammitglied den Sinn und Zweck eines Konfigurationselements auf einen Blick erfassen kann. Die Namenstemplates legen die möglichen Dateinamen für die Instanzen eines Konfigurationselements fest. Oft werden für ein Konfigurationselement mehrere Templates definiert. Beispielsweise können für das

Konfigurationselement *Quelltext* unterschiedliche Templates für Klassen und Schnittstellen eingeführt werden. Namenstemplates müssen so gewählt werden, dass die folgenden drei Anforderungen erfüllt sind:

Anforderungen an Namenstemplates

- Jede Instanz des Konfigurationselements muss anhand ihres Namens eindeutig identifiziert werden können. Etwas einschränkend kann man hinzufügen, anhand ihres Namens *und* ihrer Einordnung in der Projektstruktur, auf die ich etwas später noch eingehen werde. Für Java gilt beispielsweise, dass der Name einer Quelltextdatei nicht eindeutig sein muss, da hier die Eindeutigkeit logisch über Packages und physikalisch über entsprechende Verzeichnisstrukturen garantiert wird. Für andere Elemente, z. B. Designdokumente, existieren derlei Vorgaben von Haus aus nicht, hier muss also wirklich über die Namenstemplates die Eindeutigkeit sichergestellt werden. In größeren Projekten bietet sich zu diesem Zweck beispielsweise die Einführung einer laufenden Nummer für bestimmte Konfigurationselemente an.
- Der Name einer Datei sollte direkt auf das übergeordnete Konfigurationselement hinweisen. Dadurch kann bei Bedarf die Dokumentation des Elements ohne lange Sucherei im KM-Handbuch nachgeschlagen werden.

Und nicht zuletzt ist es wünschenswert, dass die Beziehungen zwischen Konfigurationselementen direkt aus den Dateinamen ersichtlich sind.

Ablage von Konfigurationselementen in Wikis

Viele Entwickler schwören auf Wikis zur Erstellung der Projektdokumentation. Auch Redmine besitzt eine integrierte Wiki-Funktionalität, die wir in Abschnitt 6.3.4 näher betrachten werden. So praktisch Wikis auch sein mögen, es stellt sich die Frage, ob sie auch den Ansprüchen eines KM-Prozesses gewachsen sind.

Legt man Konfigurationselemente, also z.B. eine Use-Case-Beschreibung, in einem Wiki ab, müssen zunächst die weiter oben genannten Anforderungen an die Namensgebung erfüllt sein. Zumindest dieser Punkt macht keine Probleme, denn die Eindeutigkeit ist durch Vergabe einer URL für das Dokument gewährleistet. Die restlichen Kriterien, wie beispielsweise die Vergabe von Namen, die einen Rückschluss auf verbundene Dokumente erlauben, könnte man auch in Wikis problemlos umsetzen. Tatsächlich ist dies aber gar nicht notwendig, denn schließlich ist das Verlinken auf andere Dokumente in Wikis sozusagen Pflicht. Statt einem indirekten Verweis auf das zugehörige Designdokument über den Dokumentnamen fügt man also in Wikis besser gleich einen entsprechenden Link ein.

→

Schwieriger zu erfüllen ist die Forderung der Nachvollziehbarkeit von Änderungen. Sobald ein Wiki zur Ablage von Konfigurationselementen verwendet wird, muss dieses die Rolle eines Repositorys übernehmen (mehr dazu in Abschnitt 2.2.5). Ein Wiki müsste demzufolge beispielsweise jede Änderung an einem Konfigurationselement protokollieren und über alle geänderten Versionen Buch führen. Viele Wikis erfüllen diese Anforderungen zum großen Teil, aber meist nicht komplett. So führt Redmine beispielsweise eine Versionshistorie für jeden Eintrag im Wiki, verhindert aber nicht, dass Elemente komplett und unwiderruflich gelöscht werden. Rein formal scheidet es damit als Repository aus, da die Nachvollziehbarkeit von Änderungen nicht 100%ig garantiert werden kann. Ob man das in der Praxis dann so eng sieht, steht auf einem anderen Blatt, denn Wikis sind einfach zu praktisch …

Beispiel für Namenstemplates

Abbildung 2–3 verdeutlicht diese Anforderungen an Namenstemplates mit Hilfe eines Beispiels. Im Beispiel wird davon ausgegangen, dass das Konfigurationselement *Entwurfsdokument* vom Konfigurationselement *UseCase* abhängig ist. Jeder Anwendungsfall besitzt einen eindeutigen Namen und wird in einem entsprechend benannten Use-Case-Dokument beschrieben. Der technische Entwurf des Use Case erfolgt in einem Designdokument. Im Projektalltag ist es nun äußerst hilfreich, wenn alleine aus dem Dateinamen eines Designdokuments der zugehörige Use Case klar wird. Die im Beispiel verwendeten Namenstemplates stellen dies durch die Verwendung des Use-Case-Namens in beiden Konfigurationselementen sicher. Eine Unterscheidung der Elemente erfolgt dann über das Präfix *UC* für Use Cases und *DES* für Designdokumente.

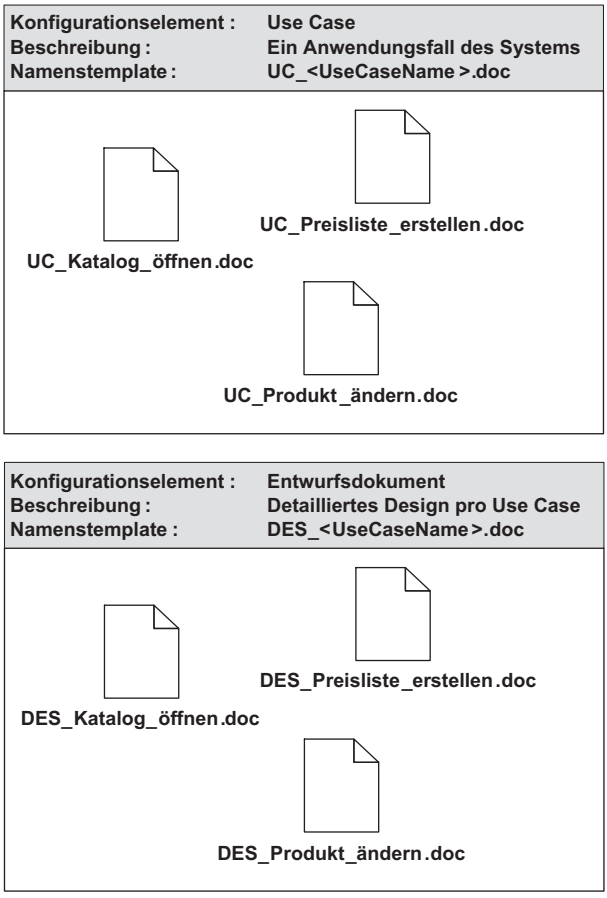


Abb. 2–3
Offenlegung der Zusammenhänge im Projekt über Namenstemplates

Wie ich weiter oben schon erwähnt habe, sollten zusätzlich zur inhaltlichen Beschreibung und zu den Namenstemplates weitere Eigenschaften und Richtlinien für die Konfigurationselemente festgehalten werden. Tabelle 2–2 gibt diesbezüglich einige Anregungen.

Element	Sinnvolle zusätzliche Dokumentation
Anforderungsdokument (z. B. Use Case) **Architektur- und Designdokument** **Benutzerdokumentation**	■ Bezugsquelle und Beschreibung der verwendeten Dokumentvorlage ■ Verweis auf Musterdokumente, die den korrekten Einsatz der Vorlage demonstrieren ■ Richtlinien für die Erstellung von Diagrammen. Dies umfasst sowohl die zu verwendenden Tools als auch Hinweise zum Einfügen der Grafiken in die Dokumente
Build-Skript	■ allgemeine Beschreibung des Build-Prozesses ■ Dokumentation der Voraussetzungen zur Ausführung des Skriptes ■ Beschreibung der möglichen Kommandozeilenparameter
Meta- und Konfigurationsdaten	■ Beschreibung, wer die Daten wann und wie ändern darf ■ Nicht beschrieben werden sollten hingegen die Inhalte der Dateien, dies ist entweder Teil der Benutzer- oder der Betriebsdokumentation!
Quelltext	■ Coding-Standards ■ Richtlinien zur Formatierung. Werden Tools zur automatischen Formatierung eingesetzt, sollten die Konfiguration und der Einsatz der Tools beschrieben werden. ■ Richtlinien zur Dokumentation des Quelltextes
Werkzeug	■ eingesetzte Version und Bezugsquelle, am besten in Form von Links auf das Installationspaket ■ Installations- und Konfigurationsanleitung

Tab. 2–2
Beispiele für die Beschreibung der Konfigurationselemente

2.2.4 Festlegung der Projektstruktur

Sobald die Konfigurationselemente beschrieben sind, ist es Zeit, sich über die *Projektstruktur* Gedanken zu machen. Für uns ist die Projektstruktur immer gleichbedeutend mit der Hierarchie der Verzeichnisse im Projekt. Um diese Struktur einigermaßen übersichtlich zu halten, empfiehlt es sich, für jedes Konfigurationselement einen separaten Ast im Verzeichnisbaum anzulegen. Im Ergebnis entsteht eine Projektstruktur wie beispielhaft in Abbildung 2–4 dargestellt.

Generierte Ordner und Dateien

Aus der Abbildung geht hervor, dass die Projektstruktur auch Ordner enthalten kann, die nicht auf ein Konfigurationselement zurückzuführen sind. Man muss an dieser Stelle sicherstellen, dass alle in derartigen Verzeichnissen abgelegten Dateien einen temporären Charakter haben. Anders ausgedrückt: Sollte jemand im obigen Beispiel das Verzeichnis *Target* löschen, darf dies keinen Einfluss auf das erstellte Produkt haben. Im Beispiel ist dies gewährleistet, da die beiden Unterverzeichnisse *Classes* und *Jar* ausschließlich generierte Dateien enthalten, die jederzeit wieder aus dem Quelltext neu erzeugt werden können.

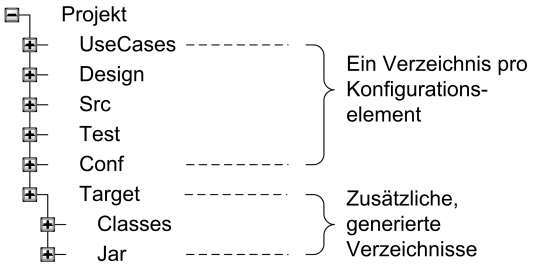


Abb. 2–4 *Aus den Konfigurationselementen abgeleitete Projektstruktur*

Einfluss der Projektorganisation

Laut Conway's Law (s. Kasten auf Seite 33) entwickelt sich die Struktur eines Systems entsprechend der Projektorganisation. Diese Tatsache müssen wir auch bei der Gestaltung der Projektstruktur in Rechnung stellen. Allerdings gilt dies meiner Ansicht nach nur für große Projekte mit räumlich verteilter Entwicklung. In diesem Fall wird in der Projektstruktur pro Teilprojekt ein eigener Ast eingerichtet – und zwar oberhalb der Ebene der Konfigurationselemente (s. Abb. 2–5).

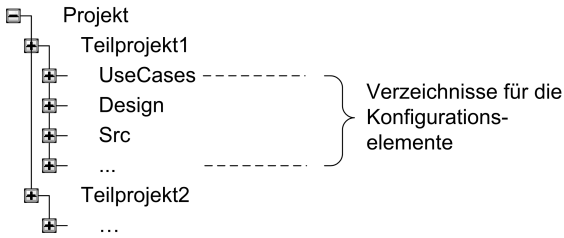


Abb. 2–5 *Projektstruktur mit Teilprojekten*

Auswirkungen auf den Systementwurf

Der Vorteil dieser Strukturierung ist, dass jedes Teilteam nur in dem für ihn vorgesehenen Ast des Projektes arbeitet und die anderen Teilprojekte nicht »sehen muss«. Allerdings hat diese Annehmlichkeit auch einen Preis, wenn auch nicht unbedingt aus Sicht des Konfigurationsmanagements. Problematisch ist die starke Betonung der organisatorischen Rahmenbedingungen in der Projektstruktur eher für den Systementwurf und die Umsetzung. Denn Komponenten und Subsysteme können nur schlecht quer über die einzelnen Teilprojekte hinweg entworfen und entwickelt werden. Die Architektur wird in der Folge der organisatorisch bedingten Struktur unterworfen – und dies ist natürlich nicht immer sinnvoll.

Conway's Law

Melvin Conway [Conway68] hat schon 1968 festgestellt, dass sich die Struktur eines Systems bevorzugt analog zur Organisation des Projektes entwickelt. Dies lässt sich wie folgt begründen: Sobald ein Projekt in mehrere Verantwortungsbereiche aufgeteilt wird, versuchen sich die Beteiligten voneinander abzugrenzen. Jeder will den eigenen Bereich so gut wie möglich vor dem Einfluss der anderen Teilprojekte abschirmen. In der Folge werden Subsysteme, Komponenten und Schnittstellen gebildet, die sich primär nicht an der Funktionalität oder an den Erfordernissen einer sauberen Architektur, sondern an der Projektorganisation orientieren. Sind an einem Projekt beispielsweise drei Parteien (Abteilungen, Dienstleister etc.) beteiligt, findet sich diese organisatorische Randbedingung des Projektes in der Architektur unter Umständen in Form von drei »künstlich« zurechtgeschnittenen Subsystemen wieder. Aus Architektursicht wären vielleicht zwei Subsysteme die bessere Wahl, doch dies lässt sich gegenüber den Beteiligten nicht durchsetzen. Eine wirklich konsistente, vielleicht sogar elegante Softwarearchitektur ist dann nicht mehr machbar. Abhilfe können hier nur umfassende organisatorische Maßnahmen im Projekt schaffen, welche die Verantwortlichkeiten an den Inhalten und nicht am Umfeld ausrichten (siehe [Dill05]).

Einfluss der Softwarearchitektur

In großen Projekten ist eine weitere Detaillierung der Projektstruktur auf Basis der Softwarearchitektur sinnvoll. Maßgeblich für diese Detailstruktur sind die im Rahmen der Analyse und des groben Entwurfes identifizierten Subsysteme oder Komponenten. Die entsprechenden Knoten werden unterhalb der Ebene der Konfigurationselemente in die Struktur eingefügt (siehe Abb. 2–6).

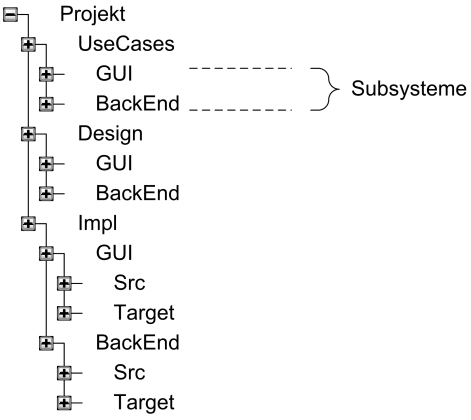


Abb. 2–6
Projektstruktur mit Subsystemen

Je nach verwendetem Werkzeug zur Projektautomatisierung ist es zudem eventuell notwendig, einen übergeordneten Ordner zu erstellen, der alle für die Implementierung relevanten Konfigurationselemente gruppiert (siehe Ordner *Impl* in Abb. 2–6). Maven beispielsweise setzt eine derartige Struktur für die Umsetzung eines modulübergreifenden Build-Prozesses voraus.

Technische Einflussfaktoren

Neben den bisher genannten Kriterien müssen auch technische Faktoren bei der Wahl der Projektstruktur berücksichtigt werden. Subversion beispielsweise setzt einige der im nächsten Abschnitt besprochenen Konzepte, wie z. B. das *Branching*, in Form von Verzeichnissen um. Diese müssen von Anfang an in die Projektstruktur integriert werden. In Kapitel 4 werden wir daher nochmals auf die Erstellung der Projektstruktur zurückkommen und die speziell für Subversion notwendigen Erweiterungen besprechen.

2.2.5 Verwaltung der Konfigurationselemente

Die Verwaltung der Konfigurationselemente spielt eine zentrale Rolle in jedem KM-Prozess. Sie unterstützt die kontrollierte Durchführung von Änderungen und hilft, die Produktivität im Projekt zu steigern. Zudem garantiert sie die Nachvollziehbarkeit aller Änderungen und die Wiederholbarkeit der Produktauslieferungen. Möglich wird dies durch die in diesem Abschnitt vorgestellten Konzepte und deren technische Umsetzung in Form von KM-Werkzeugen.

Repository

Im Repository (Softwarebibliothek) werden die Instanzen der Konfigurationselemente gespeichert, in unserem Fall also die entsprechenden Dateien. Es dient als zentrale Ablage im Projekt, auf die alle Teammitglieder zugreifen können. Man kann Repositorys auch als spezialisierte Datenbanken betrachten. Der Aufbau eines Repositorys entspricht der Projektstruktur, allerdings ohne die in Abschnitt 2.2.4 erwähnten temporären Ordner.

Ein Repository muss alle identifizierten Konfigurationselemente verwalten können, also z. B. sowohl Text- als auch Binärdateien. Insbesondere in Projekten mit einem modellbasierten Ansatz ist diese Voraussetzung nicht immer einfach zu erfüllen (siehe Kasten).

Modellbasierte Werkzeuge und das Repository

Beim Einsatz modellbasierter Entwicklungsansätze steigt die Wahrscheinlichkeit, dass einzelne Konfigurationselemente in einem Projekt nicht mehr in Form von Dateien vorliegen. Zugleich nimmt die Bedeutung der verwendeten Entwicklungswerkzeuge zu, da das Modell in der Regel in einem herstellerspezifischen Format vorliegt und ausschließlich mit den Werkzeugen dieses Herstellers bearbeitet werden kann. Eine effiziente Arbeit im Projekt ist unter diesen Voraussetzungen nur möglich, wenn das Repository über eine Schnittstelle direkt in die Werkzeuge integriert werden kann. Zusätzlich muss das Repository natürlich prinzipiell zu den Werkzeugen passen. So ist meiner Erfahrung nach die Kombination von dateibasierten Repositorys, wie z. B. Subversion, und modellbasierten Entwicklungswerkzeugen schwierig bis unmöglich. Am ehesten besteht Aussicht auf Erfolg, wenn das Entwicklungswerkzeug die einzelnen Elemente eines Modells als separate Dateien ablegt. Eine mögliche Alternative sind vollintegrierte Werkzeuge, die über ein eingebautes Repository verfügen. Man sollte vor dem Einsatz eines solchen Werkzeuges allerdings sorgfältig prüfen, ob es dem geplanten KM-Prozess wirklich gewachsen ist.

Eigenschaften des Repositorys

In einem KM-Prozess gewährleistet das Repository die Datensicherheit der Instanzen der Konfigurationselemente. Dies bedeutet konkret:

- Sicherstellung der Verfügbarkeit der Dateien
- Gewährleistung der Integrität, insbesondere auch bei gleichzeitigen Änderungen an einer Datei durch verschiedene Nutzer
- Verhinderung unberechtigter Zugriffe
- Sicherstellung der Nachvollziehbarkeit aller durchgeführten Änderungen. Dies beinhaltet auch die Möglichkeit, ungewollte Änderungen wieder zurückzunehmen.

Diese Liste kann man noch durch eine Reihe von technischen Anforderungen an ein Repository ergänzen. So muss dieses z. B. mit steigender Nutzerzahl skalieren und auch sehr große Datenmengen problemlos verwalten können.

Repository-Werkzeuge

Umgesetzt werden diese Anforderungen durch spezialisierte Werkzeuge. Für ein Projekt ist die Auswahl eines geeigneten Repository-Werkzeuges eine der wichtigsten Entscheidungen im Rahmen des Konfigurationsmanagements. Das Repository ist das Rückgrat des KM-Prozesses. Hält es den Anforderungen nicht stand, bricht der gesamte Prozess in sich zusammen.

Versionen und Deltas

Repositorys setzen intern die *Versionierung* ein, um die Nachvollziehbarkeit aller Änderungen und die Datenintegrität zu gewährleisten. Dieser Mechanismus stellt sicher, dass nach jeder Änderung an einer Datei von dieser eine neue Version im Repository abgelegt wird. Nach und nach entsteht auf diese Weise eine *Versionshistorie*, die alle jemals an einer Datei vorgenommenen Veränderungen dokumentiert. Selbst wenn eine Datei gelöscht wird, geht die Versionshistorie nicht verloren. Repositorys halten das Löschen einer Datei lediglich als Spezialfall einer Änderung fest.

Versionskontrolle und Versionskontrollsysteme

Da Versionen eine zentrale Rolle bei der Verwaltung der Elemente in einem Repository spielen, bezeichnet man den gesamten Prozess auch als *Versionskontrolle* und die Repository-Werkzeuge als *Versionskontrollsysteme*.

Vorteile der Versionierung

Die Versionierung hat in der täglichen Praxis eine ganze Reihe von Vorzügen. Sie ermöglicht z. B.:

- das Wiederherstellen eines alten Standes der Datei, z. B. weil man sich bei einem Refactoring verheddert hat und alle Änderungen gerne wieder rückgängig machen würde.
- das Wiederherstellen irrtümlich gelöschter Dateien.
- den Vergleich zweier Versionen einer Datei, z. B. um festzustellen, welche Änderungen ein Kollege seit letzter Woche durchgeführt hat.
- die parallele Arbeit an zwei unterschiedlichen Versionen derselben Datei, z. B. für das aktuelle und ein altes Release des Produktes.

Versionsnummern

Um die einzelnen Versionen einer Datei eindeutig zu identifizieren, werden diese vom Repository durchnummeriert. Das Format dieser *Versionsnummer* ist von Repository zu Repository unterschiedlich. Subversion verwendet z. B. eine automatisch vergebene, sequenziell hochgezählte Nummer. Andere Repositorys erlauben auch untergliederte oder manuell vergebene Versionsnummern.

Deltabildung

Zwei Versionen einer Datei unterscheiden sich durch ein *Delta*. Meistens umfasst dieses Delta nur einen kleinen Teil des gesamten Dateiumfanges, also z. B. nur wenige geänderte Zeilen. Diese Tatsache machen sich Versionskontrollsysteme zunutze, um Plattenplatz zu sparen. Statt jede Version einer Datei komplett zu archivieren, ermitteln die Tools das Delta zwischen zwei Versionen und legen nur dieses im Repository ab.

Als »Anker« für den Deltamechanismus benötigt das Repository eine komplette Version jeder Datei. Alle anderen Versionen können durch Anwendung der Deltas aus dem Anker abgeleitet werden. Abhängig davon, ob die allererste oder die neueste Version komplett

gespeichert wird, spricht man von *Vorwärts-* oder *Rückwärtsdeltas* (siehe Abb. 2–7).

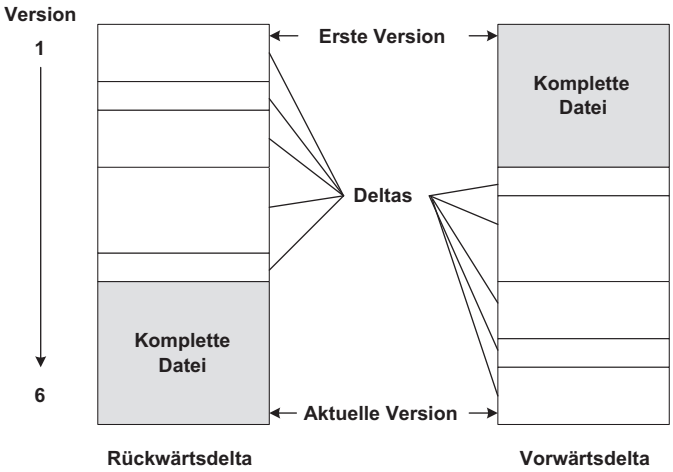


Abb. 2–7
Rückwärts- und Vorwärtsdelta (nach [Leon05])

Rückwärtsdeltas

Beide Varianten haben ihre Vor- und Nachteile. Basiert ein Repository auf Rückwärtsdeltas, steht die aktuelle Version einer Datei jederzeit zur Verfügung. Nur wenn ältere Stände abgefragt werden, muss das Repository diese anhand der Deltas wieder rekonstruieren. Nachteilig ist allerdings der höhere Aufwand für Änderungen. Wird eine neue Version im Repository gespeichert, muss zunächst die derzeit aktuellste Version entfernt und durch ein Delta zur neu hinzugefügten Version ersetzt werden. Anschließend wird diese als komplette Datei ins Repository geschrieben.

Vorwärtsdeltas

Beim Vorwärtsdelta verhält es sich im Prinzip genau andersherum. Hier fällt der Aufwand nicht beim Speichern neuer, sondern beim Abrufen der aktuellen Versionen aus dem Repository an. Zumindest für Entwicklungsprojekte ist der Theorie nach das Rückwärtsdelta daher effizienter, da die aktuellen Dateiversionen wesentlich häufiger aus dem Repository abgerufen als geändert werden. In der Praxis sollte der Deltamechanismus nicht überbewertet werden. Für die Gesamtperformance eines Tools spielen andere Aspekte, wie z. B. der Persistenzmechanismus des Repositorys, eine größere Rolle. Zudem können die prinzipiellen Nachteile beider Ansätze durch effiziente Algorithmen teilweise wieder ausgeglichen werden.

Deltabildung für Text- und Binärdateien

Unabhängig von der Richtung des Deltamechanismus bleibt noch die Frage zu klären, wie die Unterschiede zwischen zwei Versionen ermittelt werden. Uns interessiert an dieser Stelle nicht der konkrete Algorithmus, sondern lediglich dessen prinzipielle Arbeitsweise. Ich

habe weiter oben schon erwähnt, dass ein Repository alle Typen von Konfigurationselementen in einem Projekt verwalten muss. Dies betrifft nahezu immer Text- und Binärdateien. Hier trennt sich bei den Versionskontrollsystemen die Spreu vom Weizen. Denn gerade ältere Tools, wie z. B. CVS, benutzen einen auf Textdateien spezialisierten Deltamechanismus. Die Deltas werden in diesem Fall ausschließlich auf Basis von Zeilen ermittelt. Für Binärdateien funktioniert dieser Ansatz nicht, daher werden sie von der Deltabildung ausgenommen. Im Endeffekt landet jede Binärdatei komplett im Repository und sorgt dort für einen immensen Platzverbrauch. Moderne Repositorys, wie z. B. das von Subversion, können hingegen mit Text- und Binärdateien gleich gut umgehen, da der Deltaalgorithmus auf Byte-Ebene arbeitet.

Check-out und Check-in

Die Versionierung und Deltabildung kann nur funktionieren, wenn das Repository über alle Änderungen an einer Datei informiert wird. Im Projekt arbeitet man daher nicht direkt mit den Dateien im Repository, sondern mit Arbeitskopien. Das Repository liefert auf Anforderung eine solche Arbeitskopie (*Check-out*), aktualisiert diese bei Bedarf (*Update*) und liest sie nach Durchführung der Änderungen wieder ein (*Check-in* oder *Commit*). Während des Check-ins überprüft das Repository, welche Teile der Datei sich geändert haben, erzeugt das passende Delta und speichert dieses schließlich als neue Version der Datei. Der gesamte Vorgang ist in Abbildung 2–8 schematisch dargestellt.

Lokaler Arbeitsbereich

Die durch den Check-out erzeugten Arbeitskopien werden von den Teammitgliedern in einem lokalen *Arbeitsbereich* bearbeitet. Dieser wird, ebenso wie das Repository selbst, von dem eingesetzten Versionskontrollsystem verwaltet. Der Aufbau des Arbeitsbereichs im Dateisystem entspricht der Projektstruktur im Repository. In der Regel verwendet das Versionskontrollsystem zur Erstellung und Aktualisierung des Arbeitsbereichs die jeweils neueste Version der Dateien. Alternativ kann jedoch auch ein anderer Zustand angefordert werden, z. B. mit allen Dateiversionen eines älteren Release des Produktes.

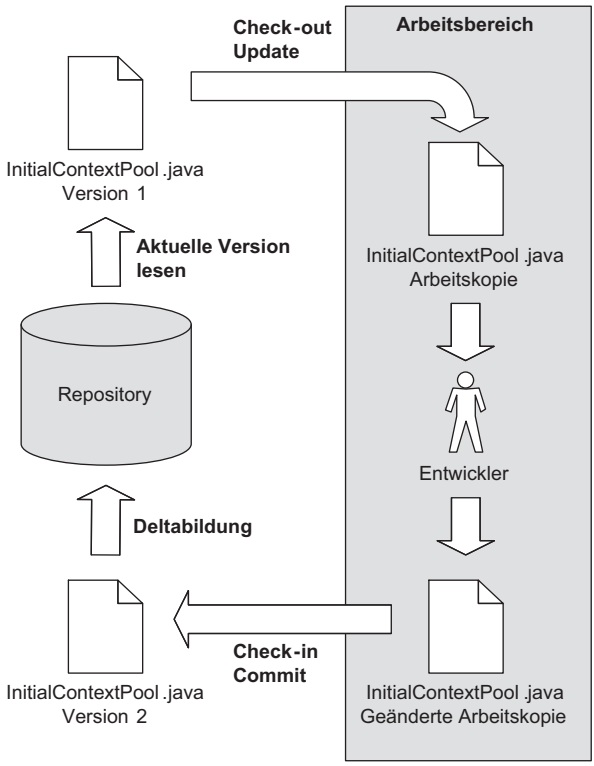


Abb. 2–8
Check-out und Check-in einer Datei

Parallele Änderungen an Dateien

Über den Check-out- bzw. den Check-in-Vorgang hat das Repository eine weitere Kontrollmöglichkeit. Es kann feststellen, ob mehrere Teammitglieder parallel an derselben Datei arbeiten wollen oder gearbeitet haben. Je nachdem, ob diese Überprüfung vor oder nach der Durchführung von Änderungen erfolgt, spricht man vom *Lock-Modify-Unlock*- oder *Copy-Modify-Merge*-Ansatz.

Lock-Modify-Unlock

Die erste Alternative verhindert parallele Veränderungen an einer Datei mit Hilfe eines Sperrmechanismus. Einige Repositorys kombinieren diese Sperre mit dem Check-out. Andere, wie z. B. Subversion, verwenden den Check-out nur zur Erstellung der Arbeitskopien und sperren Dateien mit einem separaten *Lock*-Befehl. Sobald eine Datei im Repository gesperrt ist, werden weitere Lock-Befehle mit einer Fehlermeldung quittiert. Die Freigabe erfolgt entweder mit einem separaten Befehl (*Unlock*) oder beim Check-in.

Copy-Modify-Merge

Die Copy-Modify-Merge-Methode verfolgt eine andere Philosophie. Hier dürfen die Dateien im Arbeitsbereich nach Belieben verändert werden. Erst beim Check-in überprüft das Repository, ob eine Datei

zwischenzeitlich schon von einem anderen Benutzer verändert wurde. Ist dies der Fall, tritt ein *Konflikt* auf. Über diesen wird der Anwender, der den Check-in durchführt, informiert. Er muss nun entscheiden, ob seine Änderungen mit der Version der Datei im Repository zusammenpassen. Wenn ja, führt er einen *Merge*-Vorgang durch, d. h., er führt seine Dateiversion aus dem Arbeitsbereich mit derjenigen im Repository zusammen. Und schließlich wiederholt er den Check-in, um die neue, zusammengeführte Datei im Repository zu speichern.

Damit ein Merge-Vorgang überhaupt gelingen kann, muss das Versionskontrollsystem gewisse Annahmen über den Aufbau einer Datei machen. Die Unterstützung für Merges in den Werkzeugen funktioniert daher grundsätzlich nur für zeilenbasierte Textdateien. Binäre Dateien, wie z. B. Textverarbeitungsdokumente, müssen vom Anwender manuell verglichen und dann in eine konsolidierte Datei zusammengeführt werden. In der Praxis ist dieser Ansatz meiner Erfahrung nach nicht umsetzbar. Dies beschränkt den Einsatz von Copy-Modify-Merge auf Textdateien, wie z. B. Quelltext, Build-Skripte und Konfigurationsdateien.

Vor- und Nachteile beider Ansätze

Unter Herstellern und Anhängern der diversen Versionskontrollwerkzeuge wird seit Jahren eine kontroverse Debatte geführt, ob die Copy-Modify-Merge- oder die Lock-Modify-Unlock-Methode die bessere ist. In letzter Zeit ist der Copy-Modify-Merge-Ansatz sehr populär geworden, da dieser quasi flächendeckend in Open-Source-Projekten zum Einsatz kommt. Der unbestreitbare Vorteil dieser Methode ist die Tatsache, dass niemand im Team durch jemand anderen blockiert werden kann. Alle gewünschten Änderungen können zunächst durchgeführt werden, erst beim Merge muss Zeit investiert werden, um Konflikte aufzulösen. Das Kalkül ist nun, dass für die Konfliktlösung wesentlich weniger Zeit aufgebracht werden muss, als durch das Warten auf gesperrte Dateien anfallen würde. Tatsächlich funktioniert Copy-Modify-Merge in der Praxis hervorragend, die erwähnten Open-Source-Projekte beweisen dies eindrucksvoll.

Demgegenüber punktet Lock-Modify-Unlock gerade mit der Vermeidung von Konflikten. Diese können nicht auftreten, da Änderungen an einer Datei nur sequenziell durchgeführt werden. Berechenbarkeit und Sicherheit sind daher auch die Argumente für diesen Ansatz.

Aus Sicht des Konfigurationsmanagements ist der ausschließliche Einsatz von Copy-Modify-Merge in einem Repository ausgeschlossen, da dieser Ansatz für binäre Dateien nicht geeignet ist. Für Textdateien ist er jedoch in vielen Projekten sicherlich eine gute Wahl. Subversion beherrscht sowohl Copy-Modify-Merge als auch Lock-Modify-Unlock, man kann hier also pro Konfigurationselement entscheiden, welche Methode besser geeignet ist.

Tags und Baselines

Sobald im Projekt intensiv mit dem Repository gearbeitet wird, wächst der Wunsch, ab und an eine Art Schnappschuss anzufertigen. Beispielsweise sollte für jedes freigegebene Release der Zustand des Produktes im Repository festgehalten werden.

Erstellung von Tags

Versionsnummern alleine sind schlecht zu merken und für diesen Zweck nicht praktikabel. Daher bieten Repositorys *Tags* (Bezeichner) an. Ein Tag markiert die zu einem bestimmten Zeitpunkt gültige Version aller Dateien im Repository mit einer frei wählbaren Bezeichnung. Man kann sich einen Tag auch als Querschnitt durch das Repository vorstellen.

Abbildung 2–9 verdeutlicht dieses Konzept mit einem Beispiel. Das abgebildete Repository umfasst zum Projektstart nur die Datei `UC_Preisliste_erstellen.doc` in der Versionsnummer 1. Dieser Zustand des Repositorys wird durch den Tag *Projektstart* festgehalten. Zum nächsten Meilenstein folgt ein weiterer Tag *Iteration #1*. Dieser Bezeichner umfasst nun Versionsnummer 2 des Use-Case-Dokuments und zusätzlich das Designdokument `DES_Preisliste_erstellen.doc` (Versionsnummer 2) sowie das Quelltextmodul `Preisliste.java` (Versionsnummer 1). Zu einem späteren Zeitpunkt kann der genaue Zustand des Repositorys zum Abschluss von Iteration #1 einfach durch Angabe des Tags abgerufen werden.

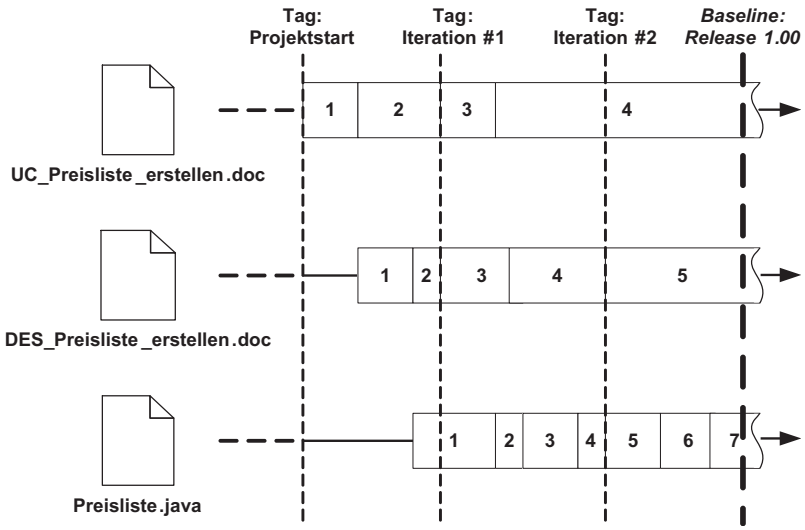


Abb. 2–9
Tags und Baselines markieren Meilensteine im Repository.

Erstellung von Baselines

Die Kennzeichnung eines wirklich bedeutsamen Ereignisses im Repository bezeichnet man auch als *Baseline* (Bezugskonfiguration). Eine Baseline ist »ein ausgewähltes, gesichertes und freigegebenes (Zwi-

schen-)Ergebnis des Systems« [Balzert97]. Technisch entspricht eine Baseline einem Tag. Im Repository selbst unterscheiden sich daher beide Konzepte, wenn überhaupt, nur minimal[4]. Die Unterschiede zwischen Tag und Baseline liegen in der Vorbereitungsphase. Einen Tag kann man »einfach so« festlegen, die Erstellung einer Baseline erfordert hingegen meist wochenlange Vorbereitung. Entscheidend ist, dass das System bei der Erstellung der Baseline einen gesicherten Status erreicht hat, z. B. durch den erfolgreichen Abschluss von Integrations- und Abnahmetests. Um diesen sicheren Zustand nicht zu gefährden, dürfen Änderungen an einer Baseline nur unter kontrollierten Bedingungen erfolgen. Spätestens an dieser Stelle setzt im Projekt das Änderungs- und Fehlermanagement auf (siehe Abschnitt 2.2.7).

Bildung von Releases

Sobald das erstellte Softwareprodukt in den produktiven Einsatz geht, bezeichnen wir es als *Release*. Releases kennzeichnen sich also dadurch, dass die Anwender damit arbeiten. Im Unterschied hierzu werden interne Auslieferungen des Produktes, z. B. für einen Integrationstest, nicht als Release bezeichnet.

Konzeptionell ist das Release den Baselines übergeordnet, d. h., für jedes Release wird eine Baseline erstellt, nicht jede Baseline ist aber gleichzeitig ein Release. Diese Unterscheidung ist für das bereits angesprochene Fehler- und Änderungsmanagement wichtig. Interne Auslieferungen unterliegen hier in der Regel anderen Einschränkungen als echte Releases, mit denen Kunden produktiv arbeiten. Releases müssen sorgfältig vorbereitet und geplant werden. Im Konfigurationsmanagement beschäftigt sich ein eigenes Verfahren mit dieser Thematik (siehe Abschnitt 2.3.1).

Branches

Die Vorbereitungen zur Erstellung einer Baseline oder eines Release benötigen je nach Projektgröße nur wenige Stunden, können aber durchaus auch Tage oder Wochen in Anspruch nehmen. In einem Projekt, an dem ich beteiligt war, haben beispielsweise alleine die Integrationstests mit anderen Systemen im Vorfeld eines Release vier Monate gedauert.

Für die Projektabwicklung stellen diese Vorbereitungsphasen ein kniffliges Problem dar. Die Funktionalität des Systems wurde bis zu diesem Zeitpunkt bereits (mehr oder weniger) vollständig entworfen und implementiert. Zwar müssen noch die gemeldeten Fehler behoben

4. Subversion kennt beispielsweise keine separaten Befehle zur Erstellung einer Baseline im Repository.

werden, doch mit dieser Aufgabe kann man nicht eine ganze Truppe an Entwicklern in Lohn und Brot halten.

Linearer Entwicklungspfad

Ein einfacher Lösungsansatz ist, dass man den Leerlauf einiger Teammitglieder während der Vorbereitung einer Baseline schlicht in Kauf nimmt. Insbesondere wenn man es nur mit einem überschaubaren Zeitraum zu tun hat, ist dies eine Option. Dieses Vorgehen wird auch *linearer Entwicklungspfad* genannt und in etwa wie folgt umgesetzt (siehe auch Abb. 2–10):

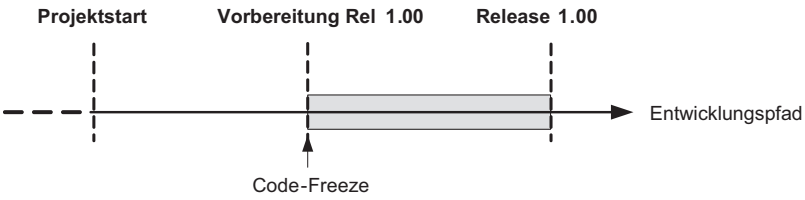


Abb. 2–10
Code-Freeze im linearen Entwicklungspfad

- Die Vorbereitungsphase für eine Baseline wird mit einem *Code-Freeze* begonnen. Zum Code-Freeze werden alle Änderungen an den Dateien des Projekts abgeschlossen und das Repository auf den neuesten Stand gebracht. Anschließend markiert man den Beginn der Vorbereitungsphase mit einem Tag.
- Sobald der Code-Freeze in Kraft ist, dürfen Änderungen an den Dateien nur noch über im KM-Handbuch beschriebene Prozesse vorgenommen werden. Beispielsweise könnten Richtlinien festgelegt werden, die ein Peer-Review[5] jeder durchgeführten Änderung vorschreiben.
- Zusätzlich erfolgt eine Unterscheidung in erlaubte und unerwünschte Änderungen. Die Implementierung neuer Funktionen wird man z. B. nach dem Code-Freeze vermeiden wollen, die Beseitigung von Fehlern ist hingegen explizit erwünscht. Es ist Aufgabe des Änderungsmanagements (siehe Abschnitt 2.2.7), diese Regelungen zu dokumentieren und im Projekt »offiziell« zu machen.
- Nach der erfolgreichen Erstellung der Baseline im Repository wird der Code-Freeze wieder aufgehoben. Mit Hilfe der Tags am Beginn und Ende der Vorbereitungsphase können bei Bedarf alle während des Code-Freeze durchgeführten Änderungen ermittelt werden.

Der Vorteil des linearen Entwicklungspfades ist die einfache Umsetzbarkeit. Abgesehen von der Erstellung einiger Richtlinien zum Code-Freeze sind im Projekt keine weiteren Vorkehrungen notwendig.

5. Unter einem Peer-Review versteht man die Kontrolle einer Änderung durch einen sachkundigen Gutachter.

Nachteile des linearen Entwicklungspfades

Erkauft wird die Einfachheit mit der ineffizienten Nutzung der Ressourcen im Projekt und dem damit einhergehenden hohen Zeitverbrauch. Hinzu kommt, dass der lineare Ansatz zwar die Vorbereitung einer Baseline abdeckt, nicht aber den Zeitraum danach. So gibt es beispielsweise keine Möglichkeit, kritische Bugfixes für das produktive Release separat auszuliefern, da im Entwicklungspfad ja schon die erweiterte Funktionalität des Folgerelease implementiert wird.

Verzweigte Entwicklungspfade

Die genannten Nachteile des linearen Entwicklungspfades können durch die Einführung von *Branches* (Verzweigungen) vermieden werden. Man spricht dann von *verzweigten* oder *parallelen Entwicklungspfaden*.

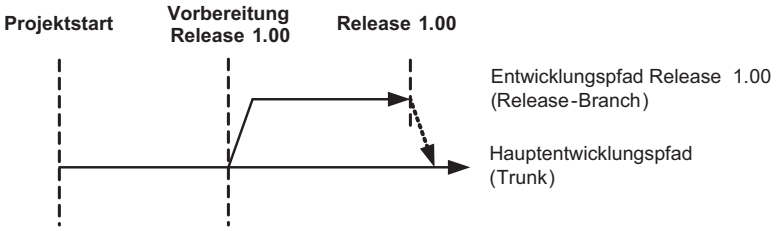


Abb. 2–11 *Parallele Entwicklungspfade zur Vorbereitung eines Release*

Abbildung 2–11 illustriert das zugrunde liegende Prinzip. Sobald die Vorbereitung eines Release beginnt, wird der Entwicklungspfad in zwei Branches aufgeteilt. Im sogenannten *Trunk* (Hauptentwicklungspfad) arbeitet ein Teil des Teams wie gewohnt weiter und beteiligt sich nicht an der Vorbereitungsphase für das neue Release. Im Release-Branch wird hingegen das anstehende Release 1.00 vorbereitet. Für diesen Branch gelten die gleichen Einschränkungen wie oben beim Code-Freeze erläutert.

Zusammenführen von Branches

Beide Branches enthalten an der Verzweigungsstelle (*Vorbereitung Rel 1.00* in Abb. 2–11) dieselben Dateiversionen. Erst im weiteren Verlauf der beiden Zweige entwickeln sich diese auseinander. Änderungen im Trunk sind nur dort sichtbar und haben keinerlei Auswirkungen auf den Release-Branch (und umgekehrt). Daher müssen alle im Release-Branch durchgeführten Änderungen und Bugfixes irgendwann auch in den Trunk übernommen werden. Die beiden Branches werden also wieder *zusammengeführt*. In Abbildung 2–11 wird dieser Vorgang durch den gestrichelten Pfeil am Ende des Release-Branch symbolisiert.

Im obigen Beispiel ist die Zusammenführung von Release-Branch und Trunk überschaubar, da Änderungen im Release-Branch nur sehr eingeschränkt durchgeführt werden. Wirklich schwierig in den Griff zu bekommen ist die echte Parallelentwicklung. Ein Beispiel hierfür sind die Pflege und Weiterentwicklung mehrerer Releases eines Produktes, weil wichtige Kunden nicht oder nur verzögert auf die jeweils neueste

Softwareversion umsteigen wollen. Oft kommt man in diesem Szenario nicht darum herum, neue Funktionalität auch in den Entwicklungszweigen älterer Releases nachzurüsten. Dies erfordert strikte Vorgaben, wer wann Änderungen zwischen den Entwicklungszweigen nachzieht.

2.2.6 Projektautomatisierung

Eine Regel aus einem der Bücher der *Pragmatic Programmer*-Reihe lautet, dass alle Tätigkeiten automatisiert werden sollten, die man bereits zwei Mal manuell durchgeführt hat [Clark04]. Begründet wird diese Forderung damit, dass es dann mit großer Wahrscheinlichkeit nicht bei einem dritten oder vierten Mal bleibt. Der Aufwand für die Umsetzung der Automatisierung zahlt sich dadurch schnell aus.

Wiederholbarkeit und Reproduzierbarkeit

Aus Sicht des Konfigurationsmanagements ist die Steigerung der Produktivität allerdings eher ein angenehmer Nebeneffekt der Projektautomatisierung. Viel wichtiger ist, dass die Schritte zur Erstellung des Produktes jederzeit wiederholt werden können (*Wiederholbarkeit*) und dass diese Schritte bei unveränderten Ausgangsbedingungen immer dasselbe Produkt erzeugen (*Reproduzierbarkeit*). Im Projekt werden diese Ziele der Projektautomatisierung mit Hilfe eines *Build-Prozesses* erreicht.

Aufbau eines Build-Prozesses

Die konkrete Umsetzung eines Build-Prozesses ist stark projektspezifisch. Wichtige Einflussfaktoren sind z. B. die verwendete Basistechnologie (Java, .NET etc.) sowie die Komplexität des erstellten Produk-

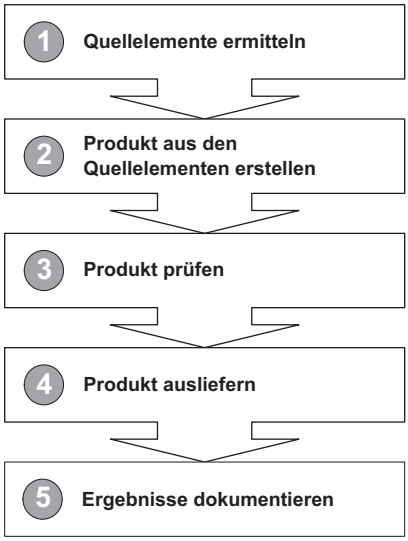


Abb. 2–12
Die fünf Schritte eines Build-Prozesses

tes. Trotzdem ist der prinzipielle Aufbau aller Build-Prozesse ähnlich. Abbildung 2–12 zeigt die fünf Schritte, die in fast allen Build-Prozessen durchlaufen werden.

Wie gesagt, die detaillierte Umsetzung der einzelnen Schritte ist abhängig vom Projekt. Die folgende Liste gibt einen Überblick, welche Aufgaben typischerweise in den einzelnen Schritten anfallen:

- *Quellelemente ermitteln*
  Auswahl der für die spätere Erstellung des Produktes benötigten Quelltext-Module, Bibliotheken und Ressourcen
- *Produkt aus den Quellelementen erstellen*
  Aufruf von Compiler, Linker und anderen Hilfsprogrammen zur Erstellung des Produktes aus den Quellelementen. In diesem Schritt werden eventuell auch neue Quellelemente über Generatoren erzeugt. Das Produkt liegt anschließend in seinen Einzelteilen vor, ist aber in der Regel noch nicht auslieferungsfähig.
- *Produkt prüfen*
  Die Einzelteile des Produktes werden mit Hilfe von automatisierten Modultests geprüft. Da in dieser Phase das Gesamtprodukt noch nicht ausgeliefert wurde, kann kein Integrationstest stattfinden. In der Praxis bedeutet dies, dass für die Modultests alle Verbindungen zu externen Ressourcen und anderen Komponenten des Systems über Mock-Objekte oder ähnliche Techniken simuliert werden müssen. Die automatisierte Durchführung von Modultests ist die Minimalanforderung an jeden Build-Prozess. Wer einen Schritt weiter gehen will, findet in Abschnitt 2.3.2 eine Reihe von Anregungen zur Qualitätssicherung mit Hilfe von automatisierten Audits und Metriken.
- *Produkt ausliefern*
  Hierzu wird zunächst aus den Einzelteilen ein Auslieferungspaket für das Produkt erstellt. In einem Java-EE-Projekt sind dies beispielsweise die *ear*- oder *war*-Dateien. Anschließend wird das Paket entweder direkt in der Zielumgebung installiert oder in einem Auslieferungsarchiv hinterlegt. Umfasst dieser Schritt auch die Installation, können im Anschluss automatische Integrationstests ausgeführt werden.
- *Ergebnisse dokumentieren*
  Alle Schritte des Build-Prozesses werden dokumentiert (z.B. in Log-Dateien). Zusätzlich kann im Repository ein Tag für den Build erzeugt werden. Damit die Log-Files, Testergebnisse und Metriken für alle Teammitglieder ohne Probleme einsehbar sind, empfiehlt sich zudem die Veröffentlichung per E-Mail oder auf der Projekt-Homepage.

Varianten des Build-Prozesses im Projekt

In den einzelnen Phasen eines Projektes bestehen unterschiedliche Anforderungen an einen Build-Prozess. Es ist ein Unterschied, ob man kurz ein neu implementiertes Feature testen will oder ob ein neues Release des Produktes an die Kunden ausgeliefert wird. Daher kommt der Build-Prozess in unterschiedlichen Varianten zum Einsatz:

- *Entwickler-Build*
  Entwickler führen den Prozess regelmäßig in ihrer lokalen Arbeitsumgebung aus. Ziel ist die Erstellung einer Testversion des Produktes. Mit dieser werden die lokal durchgeführten Änderungen an der Software überprüft. Um die Entwickler nicht unnötig aufzuhalten, werden in dieser Variante einige Schritte des Prozesses nur teilweise ausgeführt oder sogar ganz übersprungen. Da die Ausführungszeit und Einfachheit beim Entwickler-Build eine überragende Rolle spielen, muss er unter Umständen mit anderen technischen Mitteln implementiert werden als der Integrations- und Release-Build. Moderne Entwicklungsumgebungen, wie beispielsweise Eclipse, bieten sehr komfortable und performante Möglichkeiten zur Erstellung des Produktes und zur Auslieferung desselben in einen Applikationsserver. Allerdings erkauft man sich diese Einfachheit mit der doppelten Pflege des Build-Prozesses. Denn für die nächsten beiden Varianten muss man zwangsläufig auf einen über Skripte automatisierten Build-Prozess zurückgreifen.
- *Integrations-Build*
  Der Integrations-Build hat das Ziel, die Änderungen der einzelnen Entwickler in ihrer Gesamtheit zu überprüfen. Hierzu wird regelmäßig ein Stand des Produktes aus dem Repository abgerufen, über den Build-Prozess erstellt und in einer Testumgebung installiert. Existieren im Projekt parallele Entwicklungspfade, wird der Integrations-Build für jeden dieser Pfade separat durchgeführt. Die eigentliche Überprüfung findet entweder über automatische Testläufe oder mit Hilfe von Testanwendern statt. Diese Variante des Build-Prozesses wird immer automatisiert und in regelmäßigen Abständen ausgeführt[6]. Dies ist übrigens auch einer der Gründe dafür, warum man den Integrations-Build nicht mit der Entwick-

6. Über das »richtige« Zeitintervall zur automatischen Durchführung eines Integrations-Builds herrscht in der Fachwelt keine Einigkeit. Die Vertreter agiler Methoden propagieren beispielsweise die sogenannte kontinuierliche Integration [URL: ContinuousIntegration]. Hier wird der Integrations-Build in sehr kurzen Abständen, z. B. einmal alle 30 Minuten, durchgeführt. Andere Experten, wie z. B. Steve McConnell, halten die Ausführung einmal pro Tag für ausreichend [McConnell96].

lungsumgebung umsetzen kann (mehr zu diesem Thema im Kasten auf Seite 49). Stattdessen setzt man im Projekt spezialisierte Werkzeuge ein. In Kapitel 5 werden wir uns mit dem Automatisierungs-Tool *Hudson* näher beschäftigen.

- *Produktions-Build* oder *Release-Build*
  Mit dieser Variante wird ein neues Release des Produktes erzeugt. Einige Build-Tools, wie beispielsweise Maven, bieten von Haus aus Unterstützung für den Release-Build an. Dieser läuft dann in vorgegebenen Schritten ab, die z. B. auch das automatische Setzen eines Release-Tags im Repository beinhalten. Andere Tools, wie z. B. Ant, bieten diese Funktionalität nicht, dafür hat man dann auch mehr Freiheiten bei der Implementierung der Skripte. In Projekten, die Ant einsetzen, verwende ich z. B. meist einen Release-Build, der das Produkt überhaupt nicht neu erstellt. Stattdessen wird lediglich ein bereits bestehender und geprüfter Integrations-Build zum Release umdeklariert. Ein entsprechendes Beispiel finden Sie im Online-Kapitel zu Ant.

Umsetzung des Build-Prozesses

Die Basis jeder Automatisierung bilden Skripte, in denen die durchzuführenden Aufgaben in einer speziellen Skriptsprache formuliert werden. Ausgeführt werden Skripte mit Hilfe eines Interpreters, wie beispielsweise Ant oder Maven.

Deklarative und imperative Skriptsprachen

Klassischerweise werden für Skripte imperative Sprachen verwendet, so z. B. auch im Fall von Ant[7]. In einer imperativen Sprache legt man als Entwickler fest, wie eine bestimmte Aufgabe durchgeführt werden muss, um das gewünschte Ergebnis zu erreichen. Im Gegensatz dazu beschreibt man mit deklarativen Sprachen das Ergebnis selbst, nicht den Weg dorthin. Maven verfolgt beispielsweise einen waschechten deklarativen Ansatz.

Shell-Skripte

Zusätzlich zu den Skripten für spezialisierte Werkzeuge kommen zur Projektautomatisierung auch *Shell-Skripte* zum Einsatz. Man benötigt sie z. B. zum Starten und Stoppen von Web- und Applikationsservern oder zur Unterstützung des Build-Prozesses. Shell-Skripte haben allerdings den prinzipbedingten Nachteil der Plattformabhängigkeit. Sie sind immer auf eine bestimmte Shell-Umgebung zugeschnitten, z. B. die Windows-Kommandozeile oder eine der diversen

7. Um genau zu sein: Ant basiert auf einer Mischung aus deklarativen und imperativen Ansätzen. Meiner Ansicht nach überwiegt der imperative Teil – insbesondere im direkten Vergleich zu Maven. Ich möchte an dieser Stelle aber nicht verschweigen, dass die Entwickler von Ant dies anders sehen. Sie stufen Ant ebenfalls als deklaratives Werkzeug ein.

Unix-Shells. Werden in einem Projekt unterschiedliche Plattformen eingesetzt, müssen alle wichtigen Shell-Skripte mehrfach erstellt und gepflegt werden.

Automatisierung und Entwicklungsumgebungen

Integrierte Entwicklungsumgebungen bieten Funktionen wie *Build & Run* an, die sehr an die bisher beschriebene Projektautomatisierung erinnern. Entwickler können »auf Knopfdruck« das komplette System erstellen und sofort ausführen. Diese Funktionalität ist sehr komfortabel und durchaus geeignet, den lokalen Entwickler-Build umzusetzen – aber sie ist kein Ersatz für Projektautomatisierung mit Skripten!

Damit ein Build mit einer Entwicklungsumgebung gelingt, muss eine Vielzahl an Konfigurationseinstellungen manuell richtig gesetzt werden. Eine falsch gesetzte Option im x-ten Unterdialog, und der Build für das nächste Release unterscheidet sich auf subtile Weise von den bisherigen Auslieferungen. Hinzu kommt, dass Builds für Integrations- und Testzwecke oder gar für den produktiven Einsatz immer in einer kontrollierten Umgebung stattfinden sollten. Je nach Projektumfeld können dies z. B. zentral verwaltete Server sein. Oft ist es ein Ding der Unmöglichkeit, hier eine komplette Entwicklungsumgebung zum Laufen zu bringen. Selbst wenn dies gelingt, muss noch jemand »den Knopf drücken«, was dem Prinzip der Automatisierung zuwiderläuft.

2.2.7 Änderungs- und Fehlermanagement

Änderungs- und Fehlermanagement sind Prozesse zum kontrollierten Umgang mit Änderungen und Fehlerkorrekturen im Projekt. Ob eine Änderung aufgrund neuer Anforderungen oder eines Fehlers im System durchgeführt wird, hat auf die prinzipielle Vorgehensweise keinen Einfluss. Daher werde ich in diesem Abschnitt nicht zwischen »normalen« Änderungen und der Fehlerbehebung unterscheiden. Der Einfachheit halber verwende ich im Folgenden für beide Aufgabenbereiche den Begriff *Änderungsmanagement*.

Aufsetzpunkte im Projekt

Änderungsmanagement ist durchaus mit Aufwand verbunden. Klassischerweise wird ein formaler Änderungsmanagement-Prozess daher erst relativ spät im Projekt etabliert, beispielsweise nach der ersten Baseline. Geht man die Sache pragmatisch an, kann man jedoch schon in den frühen Projektphasen vom Änderungsmanagement und den dafür eingesetzten Werkzeugen profitieren.

Wenn wir uns die Zielsetzung des Änderungsmanagements nochmals kurz vor Augen führen, ergibt die frühe Nutzung eines »abgespeckten« Änderungsmanagement-Prozesses durchaus Sinn. Der kon-

trollierte Umgang mit Änderungen ist auch während der initialen Entwicklung eines Softwareproduktes wünschenswert. In dieser Phase liegt der Schwerpunkt noch auf der Umsetzung neuer Module und Funktionen, das Prinzip bleibt aber das gleiche: Wir wollen kontrollieren, wer was macht. In Kapitel 6 werden wir uns ausführlich mit einem solchen pragmatischen Änderungsmanagement auseinandersetzen. Für den Rest dieses Kapitels gilt unser Interesse jedoch dem »richtigen« Änderungsmanagement mit Änderungsanforderungen, Gremien und so weiter.

Aufgaben des Änderungsmanagements

Über das Änderungsmanagement wird sichergestellt, dass Veränderungen am Produkt erst nach einer Prüfung und Bewertung der Auswirkungen durchgeführt werden. Zudem werden alle Änderungen priorisiert sowie deren Umsetzung koordiniert und kontrolliert. Dadurch kann beispielsweise verhindert werden, dass zwei kritische Änderungen an einer Komponente kurz hintereinander von unterschiedlichen Bearbeitern vorgenommen werden.

Folgen eines fehlenden Änderungsmanagements

Das folgende Beispiel verdeutlicht die Folgen, wenn in einem Projekt auf das Änderungsmanagement verzichtet wird:

1. Ein Anwender bemerkt im Rahmen des Akzeptanztests das Fehlen einer für ihn persönlich unverzichtbaren Funktionalität im System. Beispielsweise ist er es gewohnt, eine bestimmte Funktion über einen Button aufzurufen, das System sieht hierfür jedoch nur eine Option in einem Kontextmenü vor.
2. Seiner Ansicht nach kann das Hinzufügen eines einzelnen Buttons kein schwerwiegendes Problem darstellen, daher ruft er direkt den zuständigen Entwickler an und fordert ihn auf, den Button nachzurüsten.
3. Der Entwickler will den Anwender zufriedenstellen und sagt die neue Funktionalität telefonisch zu.
4. Im Nachhinein merkt er, dass der neue Button technisch doch nicht so einfach zu realisieren ist. Er will seine Zusage jedoch nicht zurücknehmen und schafft es schließlich mit allerlei Tricks, den Button ins System einzubauen.

5. Der Anwender testet die neue Version der Software mit Button und stellt fest, dass dieser zwar funktioniert, dafür nun aber mehrere andere Funktionen des Dialogs fehlerhaft arbeiten. Er ruft wiederum den Entwickler an und bittet um die Behebung der Fehler.
6. Der Entwickler ist mittlerweile mit seinen eigentlichen Aufgaben im Hintertreffen und versucht nun zusätzlich, die neuen Fehler zu beheben. Diese erweisen sich als Nebenwirkungen des »Hacks«, der für den Einbau des neuen Buttons notwendig war. Er behebt die Fehler durch neue Fixes, die immer größere Teile des Systems mit einbeziehen.
7. Andere Anwender beschweren sich, dass eigentlich schon einwandfrei arbeitende Funktionen plötzlich wieder Fehler aufweisen. Es beginnt ein Teufelskreis, da der Entwickler zunehmend in Zeitdruck gerät und den Überblick über die von ihm durchgeführten Änderungen verliert.

Ich bin sicher, den meisten von Ihnen kommt das obige Szenario aus den eigenen Projekten bekannt vor. Hätten sich die Beteiligten des Beispiels im Vorhinein über die Auswirkungen der gewünschten Änderung ein paar Minuten Gedanken gemacht, wäre auf den Button vermutlich verzichtet worden. Selbst wenn nicht, hätte der Entwickler Gelegenheit gehabt, den notwendigen Zeitbedarf richtig einzuschätzen, und seine bisher geplanten Aufgaben wären auf andere Mitglieder des Teams verteilt worden.

Prinzipielle Vorgehensweise

Das Änderungsmanagement erzwingt eine solche »Denkpause«. Grundlage hierfür ist der in Abbildung 2–13 dargestellte Prozess. Er schreibt zunächst vor, dass *Änderungsanforderungen* (auf Englisch *Change Requests* oder kurz *CRs*) explizit formuliert und schriftlich festgehalten werden. Dasselbe gilt für die *Fehlermeldungen* (*Defect Reports*) der Anwender. Alleine die Forderung nach einer schriftlichen Erfassung hilft, Ordnung in den Änderungsprozess zu bringen. Die problematischen, im Nachhinein schwer nachvollziehbaren telefonischen oder sonstwie überbrachten Änderungsanforderungen fallen dadurch weg.

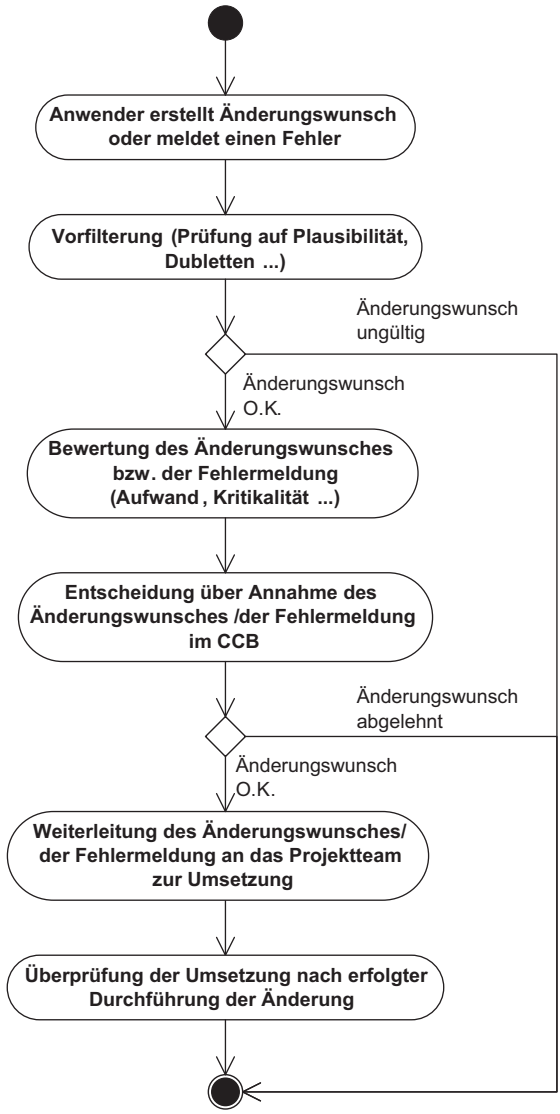


Abb. 2–13
Vorgehensweise beim Änderungsmanagement

Inhalt eines CR

Jeder CR und jede Fehlermeldung werden mit einer eindeutigen ID versehen (CR001, CR002, Defect0120 etc.). Über die ID können später z. B. Änderungen im Repository direkt auf den jeweiligen CR oder die Fehlermeldung zurückgeführt werden. Auch bei der Planung und Dokumentation von Releases sind die IDs ein wichtiges Hilfsmittel. Zusätzlich sollte ein CR mindestens die folgenden Daten enthalten:

- Name des Autors
- Datum der Erstellung
- aktueller Status (z. B. Vorgelegt, Angenommen, Umgesetzt, Freigegeben)
- Kurze inhaltliche Beschreibung der gewünschten Änderung. In komplizierten Fällen kann natürlich auch auf ein separates Dokument verwiesen werden.
- Beschreibung der Auswirkungen, falls der CR *nicht* umgesetzt wird
- Bewertung des CR hinsichtlich Kritikalität und Aufwand. Je nach Art und Umfang des CR sind eventuell verschiedene Bewertungen notwendig (z. B. eine Stellungnahme vom zuständigen Architekten und eine von einem Fachexperten).
- Priorisierung des CR durch das *Change Control Board* (*CCB*). Stattdessen kann ein CR natürlich auch komplett abgelehnt werden. In jedem Fall ist die Begründung der Entscheidung des CCB zu dokumentieren.

Inhalt einer Fehlermeldung

Fehlermeldungen entsprechen vom Aufbau her einem CR, nur dass die inhaltliche Beschreibung mehr technische Details enthält und aus mehreren Teilen besteht (z. B. Stack-Traces, Modulnamen, Fehlernummern). In CRs wird hingegen oft eine Freitext-Beschreibung gewählt.

Werkzeuge zum Änderungs- und Fehlermanagement

In der Praxis wird Änderungsmanagement erstaunlich oft quasi »per Hand«, sprich mit Excel-Tabellen, realisiert. Ein Grund hierfür mag sein, dass viele spezialisierte Werkzeuge zum Änderungs- und Fehlermanagement ausgesprochen unhandlich zu bedienen sind bzw. waren. Gerade in den letzten Jahren hat sich in diesem Bereich jedoch einiges zum Positiven gewandelt. Viele Open-Source-Werkzeuge zum Änderungsmanagement sind sehr leistungsfähig, einfach zu bedienen und auch für große Projekte eine gute Wahl. Auch die professionellen Anbieter haben dazugelernt. So stehen z. B. einige professionelle Tools Open-Source-Projekten kostenfrei zur Verfügung. Die Anbieter und deren Produkte profitieren dadurch vom offenen Feedback der Community.

Aktuelle Werkzeuge zum Änderungsmanagement umfassen in der Regel zumindest die folgenden Funktionen:

Typischer Funktionsumfang

- Unterstützung bei der Erfassung und Verwaltung von Aufgaben, CRs und Fehlerberichten mit Hilfe spezieller Dialoge
- Filterfunktionen, z. B. nach Datum, Kritikalität und Subsystemen
- Zuweisung einer Aufgabe, eines CR oder eines Fehlerberichts zu einem Bearbeiter. Dieser kann seinen Fortschritt bei der Umsetzung direkt im Werkzeug dokumentieren.

- Erstellung von Auswertungen, z. B. über neu hinzugekommene und behobene Fehler

In Kapitel 6 werden wir uns ausführlich mit dem Open-Source-Tool Redmine auseinandersetzen.

Etablierung eines Änderungsmanagers

Die Verwaltung der CRs und Fehlermeldungen übernimmt im Projekt der *Änderungsmanager*. Eine der wichtigsten Aufgaben des Änderungsmanagers ist die Vorfilterung der eingehenden CRs und Fehlermeldungen. Er muss sicherstellen, dass unvollständig beschriebene CRs und eindeutige Dubletten von vornherein zurückgewiesen werden. Dies erfordert eine gewisse Erfahrung, daher halte ich die dauerhafte Besetzung der Rolle des Änderungsmanagers im Projekt für wichtig. Wird diese Aufgabe immer von wechselnden Teammitgliedern wahrgenommen, kann eine effektive Vorfilterung nicht stattfinden.

Nach der prinzipiellen Annahme eines CR bewertet der Änderungsmanager diesen hinsichtlich Aufwand und Kritikalität. Je nach Art des CR oder des Fehlerberichts benötigt er hierfür Unterstützung von Anwendern, Analysten oder Entwicklern. Der CR wird im Rahmen der Bewertung eventuell noch vervollständigt. Beispielsweise können die Auswirkungen auf andere Komponenten oder auf Schnittstellensysteme analysiert und beschrieben werden.

Change Control Board

Erst danach wird der CR in das Change Control Board weitergeleitet. Dieses Gremium entscheidet abschließend über die Annahme oder Ablehnung eines CR. Die Zusammensetzung des CCB richtet sich nach Größe und Komplexität eines Projektes. In kleinen Projekten besteht das CCB oft aus einer einzelnen Person, in der Regel dem Projektleiter. Große Projekte benötigen CCBs mit Vertretern der einzelnen Interessengruppen, wie z. B. Anwendern, IT-Abteilungen und Dienstleistern. Nicht immer wird die Ablehnung einer Änderungsanforderung auf Begeisterung stoßen, daher muss das CCB im Projekt von allen beteiligten Parteien als Autorität anerkannt sein.

Berücksichtigung des Releaseplans

Sobald eine Änderungsanforderung vom CCB freigegeben worden ist, kann das Projektteam mit der Umsetzung beginnen. Die Einplanung erfolgt abhängig von der Bewertung der CRs und eventuell geäußerter Wünsche des CCB. An dieser Stelle ist es wichtig, dass auf die Releaseplanung Rücksicht genommen wird. In Abschnitt 2.3.1 werde ich auf dieses Thema genauer eingehen, nur so viel vorweg: Die geplanten Releases geben in einem Projekt gewissermaßen den Takt vor. Man kann nicht beliebig viele Änderungen oder Bugfixes in ein Release »hineinpacken«.

Der letzte Schritt im Änderungsmanagement ist die Überprüfung der Änderung nach der Umsetzung. Dies kann z. B. durch Audits erfolgen, mit denen wir uns in Abschnitt 2.3.2 noch näher beschäftigen werden.

2.3 Erweiterter Prozess

2.3.1 Releasemanagement

Wird eine Version des Softwareproduktes an die Endanwender ausgeliefert, bezeichnen wir diese als *Release*. Ein Release wird durch den normalen Build-Prozess erzeugt und unterscheidet sich technisch nicht von den projektinternen Auslieferungen. Bevor ein Release an die Anwender übergeben wird, durchläuft es allerdings zusätzlich eine umfangreiche Qualitätsprüfung in Form von Audits und Testläufen.

Im Rahmen des Releasemanagements machen wir uns Gedanken, welche Funktionalität die einzelnen Releases des Produktes beinhalten sollen und wann diese Releases an die Anwender ausgeliefert werden. Zudem wird festgelegt, welche Schritte zur Vorbereitung und Erstellung der Releases notwendig sind.

Beschreibung der Releases

Um ein Release vollständig zu beschreiben, müssen wir den funktionalen Umfang und den geplanten Termin der Auslieferung an die Endanwender kennen. Ein Release setzt sich aus den gesammelten und bewerteten Änderungsanforderungen der Anwender zusammen. Über das Releasemanagement stellen wir sicher, dass CRs und Bugfixes zu sinnvollen Releases zusammengefasst werden. Beispielsweise bevorzuge ich Releases, die funktionale Erweiterungen auf der Ebene von Subsystemen bündeln. Jedes Release ändert dadurch schwerpunktmäßig nur einen bestimmten Teil des Systems. Wird ein System komplett neu entwickelt, bilden statt CRs beispielsweise die umzusetzenden Use Cases die Grundlage für die Releaseplanung.

Zur eindeutigen Bezeichnung der Releases werden *Releasenummern* verwendet. Bewährt hat sich der in Abbildung 2–14 dargestellte Aufbau der Releasenummer.

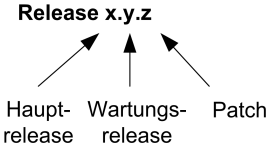


Abb. 2–14
Aufbau einer Releasenummer

Hauptrelease

Das *Hauptrelease* wird nur dann erhöht, wenn das Produkt signifikant erweitert wurde. Ein Kriterium für einen Wechsel des Hauptrelease können beispielsweise Änderungen an den verwendeten Dateiformaten oder eine grundsätzliche Überarbeitung der Benutzeroberfläche sein. Auch komplett neue Subsysteme oder Komponenten rechtfertigen die Erhöhung des Hauptrelease. Sowohl den Anwendern als auch uns selbst wird durch die Erhöhung des Hauptrelease verdeutlicht, dass der Releasewechsel mit erheblichem Aufwand verbunden sein wird. Beispielsweise müssen Anwender neu geschult oder die Daten des alten Release vor der Auslieferung migriert werden.

Wartungsrelease

Demgegenüber liefert ein Projekt *Wartungsreleases* in regelmäßigen Abständen aus. Wartungsreleases beinhalten hauptsächlich Fehlerbehebungen und kleinere funktionale Erweiterungen.

Patches

Der letzte Bestandteil der Releasenummer ist für Ausnahmefälle vorgesehen. *Patches* sollten im Normalfall nicht notwendig sein, nur welches Projekt ist schon normal? Trotz aller Sorgfalt lassen sich Fehler im Produkt nicht vermeiden. Wirklich kritische Bugs müssen den Anwendern und Kunden zuliebe schnell beseitigt werden. Ist der Zeitraum bis zum nächsten Wartungsrelease zu lange, ist die Auslieferung eines Patches nicht zu vermeiden. Patches werden ausschließlich zur Fehlerbehebung eingesetzt, funktionale Erweiterungen sind hier tabu. Da Patches nicht geplant werden können, bleibt in der Regel wenig Zeit zur Vorbereitung. Im Konfigurationsmanagement-Handbuch sollte daher das genaue Prozedere zur Erstellung und Auslieferung eines Patches dokumentiert werden. Dies ist auch deshalb empfehlenswert, da Patches erfahrungsgemäß immer zum ungünstigsten Zeitpunkt notwendig werden. Nichts verdirbt den Urlaub nachhaltiger als der mühsame Versuch, per Telefon den Prozess zur Auslieferung eines Patches zu beschreiben.

Planung der Releases

Kern des *Releaseplans* ist eine Tabelle mit der Beschreibung der Releases. Zusätzlich finde ich eine grafische Übersicht hilfreich, welche die zeitliche Abfolge der Releases und die Auswirkungen auf die Entwicklungspfade im Projekt verdeutlicht. Die Bezeichnung Releaseplan ist insofern etwas irreführend, als in der Tabelle sowohl künftige als auch die bereits ausgelieferten Releases beschrieben werden. Wen dies stört, der kann auch zwei Tabellen erstellen, einmal wirklich für den Plan und einmal für die Dokumentation der erfolgten Auslieferungen.

Zeitlicher Abstand zwischen Releases

Für zukünftige Releases ist insbesondere die Wahl des »richtigen« zeitlichen Abstands zwischen Haupt- und Wartungsreleases wichtig. Der Abstand zwischen den Wartungsreleases bestimmt, wie viele neue

Funktionen und Fehlerbehebungen umgesetzt werden können. Gerade bei neuen Produkten sollten Wartungsreleases in relativ kurzen Abständen ausgeliefert werden, beispielsweise alle drei Monate. Sind die Abstände allerdings noch kürzer, ist das Team mehr mit den Vorbereitungen für die Releasebildung als mit der Umsetzung von CRs und Bugfixes beschäftigt. Große Releaseabstände und die damit verbundenen umfangreichen Änderungen im System sind ebenfalls problematisch. Die dadurch erforderlichen umfangreichen Test- und Vorbereitungsphasen führen zu zeitlich ausgedehnten parallelen Entwicklungspfaden im Projekt.

In Tabelle 2–3 ist ein beispielhafter Releaseplan dargestellt. Unter der Annahme, dass der Plan im Dezember 2012 betrachtet wird, beinhaltet er sowohl die Dokumentation der bereits ausgelieferten als auch die Planung der kommenden Releases. Das erste Release wurde demnach Anfang April 2012 installiert. Die Vorbereitung des Release nahm insgesamt zehn Wochen in Anspruch. Für ein Hauptrelease, noch dazu das erste Release des Produktes überhaupt, kein übertrieben langer Zeitraum.

Anfang des nächsten Quartals wurde planmäßig das erste Wartungsrelease 1.1.0 ausgeliefert. Dieses umfasste eine funktionale Erweiterung des Systems um drei Change Requests und drei Bugfixes. Für die Beschreibung des Inhalts des Release sind die IDs der realisierten CRs und behobenen Defects völlig ausreichend. Der Vorlauf des Wartungsrelease ist mit zwei Wochen deutlich kürzer als der des Hauptrelease. Auch dieser Wert ist plausibel, schließlich müssen nur die wenigen Erweiterungen des Systems durch die Qualitätssicherung überprüft werden. Das folgende Wartungsrelease 1.2.0 erscheint wiederum drei Monate später. Offensichtlich ist im Release 1.2.0 allerdings ein schwerwiegender Fehler übersehen worden, denn Anfang Dezember musste ein ungeplanter Patch auf 1.2.1 installiert werden.

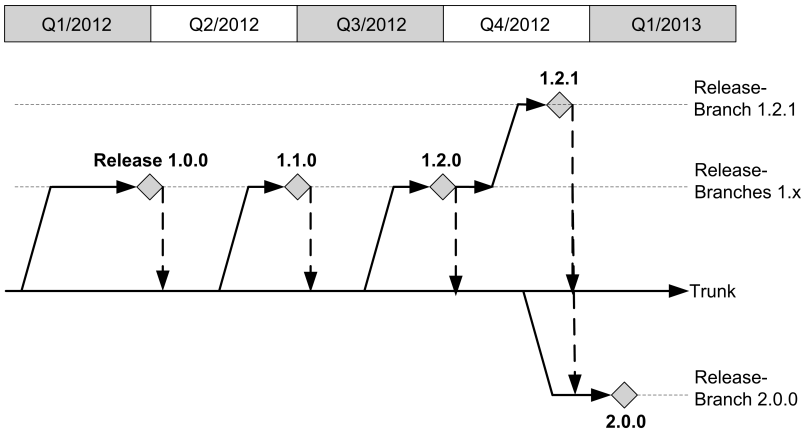
Rel.-Nr.	Inhalt	Vorlauf	Auslieferung
1.0.0	Initiale Auslieferung	10 Wochen	01.04.2012
1.1.0	CR002, CR010, CR011 Fixes für Defects: 102, 110, 150	2 Wochen	01.07.2012
1.2.0	CR004, CR020 Fixes für Defects: 170, 175, 180	2 Wochen	01.10.2012
1.2.1	Fix für Defect: 191	Keiner	04.12.2012
2.0.0	Neues Modul Auswertungen	8 Wochen	01.02.2013

Tab. 2–3
Beschreibung der Releases

Die Auswirkungen des Patches auf die Entwicklungspfade im Projekt sind aus der Tabelle nicht ohne Weiteres ersichtlich. Aus der grafischen

Darstellung des Releaseplans in Abbildung 2–15 wird jedoch sofort klar, warum nach 1.2.0 kein weiteres reguläres Wartungsrelease mehr geplant war und der Patch daher umso ärgerlicher ist. Bedingt durch die acht Wochen Vorbereitungsphase für Release 2.0.0 ist Anfang Dezember 2012 bereits ein neuer Entwicklungspfad im Projekt gestartet worden. Der Patch auf 1.2.1 wurde nur einige Tage später ausgeliefert. In der Folge sind nun nicht zwei, sondern drei aktive Pfade im Projekt.

Abb. 2–15
Grafische Darstellung des Releaseplans



Die gestrichelten Linien in Abbildung 2–15 markieren die Übernahme der Änderungen aus den Release-Branches in den Trunk. Nach dem Patch auf 1.2.1 muss zusätzlich zum Trunk auch der Release-Branch für 2.0.0 auf den neuesten Stand gebracht werden.

Das Beispiel ist so konstruiert, dass keine andere Wahl bleibt und der Mehraufwand für den Abgleich der drei Pfade zumindest temporär in Kauf genommen werden muss. Ich habe jedoch schon mehrfach Situationen im Projekt erlebt, in denen bewusst parallele Releases eingeplant wurden. Eigenartigerweise war der Hintergrund immer die Überlegung, dass durch die Parallelisierung bestimmter Arbeiten Zeit gespart wird und der eigentlich unrealistische Endtermin des Projektes dadurch doch noch einzuhalten ist.

Einfluss auf die Projektplanung

Es ist auch Aufgabe des Releasemanagements, derartigem Unfug vorzubeugen. Parallele Entwicklungspfade gilt es wann immer möglich zu vermeiden, da sie selbst mit den besten Werkzeugen sehr viel zusätzlichen Aufwand im Projekt erzeugen. Zudem ist jede Zusammenführung von Entwicklungspfaden eine potenzielle Fehlerquelle. Releasepläne und die oben gezeigte grafische Darstellung können helfen, dem Projektmanagement die Konsequenzen einer allzu kreativen Planung deutlich zu machen.

2.3.2 Audits

Mit Hilfe von *Audits* wird in einem KM-Prozess überprüft, ob die Konfigurationselemente alle an sie gestellten Anforderungen erfüllen. Dies umfasst beispielsweise die funktionale Prüfung des Systems mit Hilfe von Anwender- und Integrationstests. Andere Audits verifizieren, ob die vereinbarten Standards, wie beispielsweise die Quelltextformatierung oder die durchgängige Verwendung von Dokumentvorlagen, eingehalten werden.

Durchführung von Audits

Allen Audits ist gemeinsam, dass sie den Istzustand gegenüber einem vorher festgelegten Sollzustand überprüfen. Erfüllt das geprüfte Element das Soll, verläuft der Audit erfolgreich. Wenn nicht, muss das Element nachgebessert und der Audit wiederholt werden.

Typische Fehler bei der Durchführung von Audits

Erstaunlicherweise wird in der Praxis oft gegen dieses einfache Prinzip verstoßen. Ein Beispiel hierfür sind die von vielen Entwicklern verhassten Quelltext-Audits. In einem solchen Audit wird ein ausgewähltes Quelltextmodul auf die Einhaltung bestimmter Kriterien, wie beispielsweise die Vollständigkeit der Dokumentation, überprüft. Durchgeführt wird die Überprüfung von ein bis zwei fachkundigen Auditoren und einem der zuständigen Entwickler. Das Ergebnis des Audits ist eine klare Ja/Nein-Aussage. Entweder das Modul besitzt eine vollständige Dokumentation oder eben nicht. Tatsächlich laufen derartige Audits aber oft völlig anders ab. Statt die Dokumentation zu überprüfen, zerpflücken die Auditoren die Implementierung bis ins kleinste Detail. Da drei Spezialisten mindestens drei unterschiedliche Meinungen zur optimalen technischen Umsetzung einer gegebenen Anforderung haben, schlägt das Audit dadurch beinahe zwangsläufig fehl. Statt die Qualität des Produkts zu verbessern, führt ein derartiges Audit letztlich nur zu erheblichen Spannungen im Projekt.

Audit-Plan

Um dies zu vermeiden, muss im Projekt rechtzeitig ein Audit-Plan entworfen und im KM-Handbuch dokumentiert werden. Im Plan werden pro Audit der Teilnehmerkreis, der Zeitpunkt und das genaue Ziel festgehalten (siehe Tab. 2–4 für ein Beispiel).

Tab. 2–4
Beschreibung eines manuellen Audits

Audit:	Review Designdokument
Teilnehmer:	■ Der Autor des Dokuments ■ Ein Architekt aus dem Architekturboard ■ Ein Mitglied aus dem QS-Team
Zeitpunkt:	1 Woche nach Versand des Designdokuments an die Teilnehmer
Ziel:	Prüfung des Designdokuments auf folgende Kriterien: ■ Dokumentvorlage »Designdokumente« verwendet? ■ Dokumentversion und Änderungshistorie gefüllt? ■ Standardgliederung eingehalten? Ausnahmen sind begründet? ■ Design konform zur Softwarearchitektur? ■ Design geeignet zur Umsetzung der funktionalen und nichtfunktionalen Anforderungen an das Modul? (Wenn nein: genaue Begründung!)

2.3.3 Metriken

Im Gegensatz zu Audits, die im Prinzip ein boolesches Ergebnis liefern, erlauben Metriken differenziertere Aussagen über die Qualität des Systems und den Status des Projektes. Eine »Metrik ... bezeichnet im Allgemeinen ein System von Kennzahlen oder ein Verfahren zur Messung einer quantifizierbaren Größe« [Wikipedia: Metrik]. Wie wir etwas später sehen werden, können Metriken entweder manuell oder vollständig automatisch ermittelt werden. In einem Softwareprojekt existieren drei Kategorien von Metriken (vgl. [Kan02]):

Kategorien von Metriken

- *Produktmetriken*
  Sie beschreiben die Eigenschaften des erstellten Produktes, wie beispielsweise Größe, Komplexität, Antwortzeiten und allgemeine Qualitätsmerkmale (Fehlerrate etc.).
- *Prozessmetriken*
  Diese Metriken geben Auskunft über die Güte des Entwicklungsprozesses. Ein Beispiel hierfür ist der durchschnittliche benötigte Aufwand zur Behebung eines Fehlers im System.
- *Projektmetriken*
  Mit Projektmetriken wird das Softwareprojekt an sich beschrieben. Typische Beispiele sind das Auftragsvolumen, die Teamgröße und die Dauer des Projektes.

Wir setzen in unserem KM-Prozess hauptsächlich Metriken aus der Kategorie *Produktmetriken* ein. Diese erlauben es uns, den Fortschritt des Projektes und die Qualität des erstellten Produktes transparent zu machen. Die Metriken der anderen Kategorien sind übrigens nicht

weniger wichtig. Sie fallen jedoch tendenziell in andere Aufgabengebiete des Projektes. Die *Projektmetriken* werden beispielsweise in der Regel vom Projektmanagement erfasst und überwacht.

Sinnvolle und sinnlose Metriken

Metriken versuchen, einen bestimmten Aspekt eines Softwareproduktes in eine auf den ersten Blick leicht verständliche Zahl zu gießen. Dies macht die besondere Faszination von Metriken aus, schließlich ist Softwareentwicklung ein komplexes Geschäft und jede potenzielle Vereinfachung gern gesehen. Leider liegt genau in dieser Eigenschaft von Metriken auch deren größte Gefahr. Denn »jeder Versuch, die Komplexität zu reduzieren, erhöht die Komplexität an einem anderen Ort« [Lotter06]. Im Falle von Metriken bedeutet dies, dass die Ermittlung und vor allem die Interpretation der gemessenen Zahlen keinesfalls so einfach sind, wie es zunächst den Anschein hat.

Beispiel für eine sinnlose Metrik

Betrachten wir zum Beispiel die sehr beliebte Metrik LOC etwas genauer. LOC steht für *Lines of Code*, d. h., zur Ermittlung der Metrik werden einfach alle Zeilen des Quelltextes gezählt. Je größer die Zahl, desto größer das erstellte Produkt. Doch ist das wirklich so? Die Schwierigkeiten beginnen schon mit der genauen Definition einer Quelltextzeile. Ist eine leere Zeile im Quelltext auch eine Quelltextzeile? Was ist mit Kommentaren? Was passiert, wenn ein Entwickler 200 Zeichen lange Monsterzeilen schreibt, ein anderer aber sauber nach 80 Zeichen umbricht? Um diesen grundsätzlichen Problemen zu begegnen, wurde eine ganze Reihe von LOC-Varianten definiert, die beispielsweise explizit alle Kommentarzeilen von der Messung ausschließen oder nur die wirklich ausführbaren Statements zählen. Verwendet man die LOC-Metrik im Projekt, müssen alle Beteiligten genau verstehen, was wirklich gemessen wurde.

Sobald die Semantik der LOC-Metrik eindeutig festgelegt wurde, kann das Produkt »vermessen« werden. Im nächsten Schritt gilt es nun, die ermittelte Zahl zu interpretieren. Ist beispielsweise der LOC-Wert im Vergleich zur letzten Messung gestiegen, könnte man dies als Fortschritt im Projekt werten. Leider lauert auch hier der Teufel im Detail. Denn die LOC-Metrik verhält sich umgekehrt proportional zur Effizienz im Systementwurf. Anders ausgedrückt sorgt ein gutes Design dafür, dass möglichst wenig neuer Code zur Implementierung einer Funktion im System notwendig ist. Misst man in einem solchen Projekt den Fortschritt an der LOC-Metrik, bestraft man sozusagen die effiziente Arbeit des Design-Teams.

Alles in allem hat die LOC-Metrik nur einen wirklichen Vorteil: Sie lässt sich recht einfach automatisch ermitteln. Der Wert der Metrik

ist meines Erachtens allerdings sehr fraglich. Sie beantwortet keine der Fragen zum Fortschritt oder Qualitätsstand des Projektes wirklich zuverlässig.

GQM-Verfahren

Um an eine Liste mit sinnvollen Metriken zu gelangen, müssen wir offensichtlich anders vorgehen. Statt die technisch einfach zu realisierenden Metriken zu verwenden, sollten die Ziele der Messung im Vordergrund stehen. Diesen Ansatz bezeichnet man auch als das *GQM-Verfahren* (Goal Question Metric). Er besteht aus drei Schritten (vgl. [Ebert05]):

1. Festlegung eines konkreten Ziels. Beispielsweise soll im Projekt in einem bestimmten Zeitraum eine vorgegebene Anzahl von Anwendungsfällen umgesetzt werden.
2. Formulierung von Fragen zur Überprüfung der Zielerreichung. Im obigen Beispiel lautet die übergeordnete Frage: »Wie viele Anwendungsfälle sind bereits umgesetzt worden?« Um dies zu beantworten, muss man für jeden einzelnen Anwendungsfall wiederum die folgenden Fragen beantworten: »Existieren für den Anwendungsfall *xyz* das Use-Case-Dokument und das Designdokument im Status *Freigegeben*? Liegen alle im Designdokument spezifizierten Klassen fertig implementiert im Repository vor? Werden die Modultests für diese Klassen fehlerfrei durchlaufen?« Nur wenn alle diese Fragen mit *Ja* beantwortet werden, ist ein Anwendungsfall vollständig umgesetzt.
3. Definition einer Metrik, mit welcher der Grad der Zielerreichung ausgedrückt werden kann. Die Metrik im obigen Beispiel könnte *Umgesetzte Use Cases* lauten. Ihr Wert wird genau dann um eins erhöht, wenn für einen gegebenen Anwendungsfall alle oben genannten Fragen positiv beantwortet werden können.

Das Ergebnis des GQM-Verfahrens wird pro Metrik im KM-Handbuch dokumentiert. Hierbei sollte jeder einzelne Schritt, also das Ziel, die Fragen und die daraus abgeleitete Metrik, separat beschrieben werden.

Manuell ermittelte Metriken

Spezifiziert man die Metrik *Umgesetzte Use Cases* nach dem oben beschriebenen Verfahren, erhält man eine zuverlässige Aussage über die aktuelle Größe des Produktes und den Fortschritt im Projekt. Natürlich hat die Sache auch einen Haken. Diese Metrik kann, im Gegensatz zu LOC, nur manuell ermittelt werden. Beim Entwurf der Metriken für ein Projekt kann man realistischerweise nur eine Handvoll manueller Messungen vorsehen. Schließlich sollen die Messwerte

mit einem vertretbaren Aufwand ermittelt werden. Neben einer Metrik zur Bestimmung des Projektfortschritts, wie z. B. *Umgesetzte Use Cases*, halte ich aus Sicht des Konfigurationsmanagements die folgenden manuellen Metriken für sinnvoll:

Sinnvolle manuelle Metriken

- *Neu erfasste Fehler pro Zeitraum*
  Wenn ein Werkzeug zum Fehlermanagement eingesetzt wird, kann diese Metrik direkt aus der Fehlerdatenbank ermittelt werden. Sie gibt an, wie viele neue Fehler in einem definierten Intervall (z. B. eine Arbeitswoche) im System erfasst wurden. Hierbei werden Dubletten und ungültige Fehlermeldungen nicht gezählt.
- *Behobene Fehler pro Zeitraum*
  Gibt an, wie viele Fehler in einem definierten Intervall behoben wurden. Als behoben gilt ein Fehler erst dann, wenn der Bugfix mit Hilfe eines Testlaufes verifiziert wurde. Stellt man die behobenen Fehler den neu erfassten gegenüber, lässt dies Rückschlüsse auf die Entwicklung der Softwarequalität des Produktes zu. Werden über einen längeren Zeitraum mehr Fehler gefunden als behoben, ist entweder das Wartungsteam unterbesetzt, oder das Produkt hat wirklich ein ernsthaftes Qualitätsproblem.
- *Aufwand pro Fehlerbehebung*
  Mit Hilfe dieser Metrik kann man im Laufe der Vorbereitung eines Release abschätzen, wie viel Aufwand insgesamt noch zur Behebung der bekannten Fehler investiert werden muss.
- *Fehlerdichte*
  Diese Metrik ist der Quotient aus der Anzahl der bekannten Fehler und der Produktgröße. Mit Hilfe der Fehlerdichte kann eine Prognose der wahrscheinlichen Fehleranzahl in neuen Releases erstellt werden. Liegt der Wert für die Fehlerdichte aktuell beispielsweise bei fünf Fehlern pro Use Case und werden in einem kommenden Release zehn neue Use Cases ausgeliefert, muss man mit ca. 50 Fehlern in dem neuen Produkt rechnen. Sind in der Vorbereitungsphase des Release bisher nur 30 Fehler entdeckt worden, sollte man den Zeitraum bis zum nächsten Wartungsrelease nicht allzu groß ansetzen.

Automatisierte Metriken

Zusätzlich zu den manuell erfassten Metriken kann eine Reihe von Messgrößen direkt aus den Konfigurationselementen ermittelt werden. Hierbei ist insbesondere der Quelltext des Systems eine gute Grundlage für automatisierte Messungen. Man nennt diesen Vorgang daher

auch *statische Quelltextanalyse*. In der Tabelle 2–5 finden Sie eine Übersicht häufig verwendeter Metriken für objektorientierte Software.

Automatisierte Audits

Neben der Ermittlung der in der Tabelle genannten Metriken können durch die statische Quelltextanalyse auch *automatisierte Audits* durchgeführt werden. Spezialisierte Werkzeuge prüfen beispielsweise die Einhaltung von Namenskonventionen und die Vollständigkeit der Quelltextkommentare. In Kapitel 5 werden wir uns konkret mit diesem Einsatzgebiet der Quelltextanalyse befassen.

Tab. 2–5
Gängige automatisch ermittelbare Metriken

Metrik	Beschreibung	Kommentar
Coupling between Objects (CBO)	Ermittelt die Anzahl der Klassen, zu denen die vermessene Klasse in Beziehung steht	Je höher dieser Wert ist, desto »enger« ist die Kopplung einer Klasse mit anderen Modulen des Systems. Dies widerspricht den Prinzipien des modularen Designs und hat negative Auswirkungen auf die Wartbarkeit, Änderbarkeit und Testbarkeit (vgl. [Chidamber94]).
Cyclomatic Complexity (CC)	Zählt die linear unabhängigen Ausführungspfade in einer Methode	Siehe die Erläuterungen im folgenden Abschnitt
Depth of Inheritance Tree (DIT)	Liefert die Anzahl der Oberklassen einer Klasse	Klassen mit vielen Oberklassen erben deren Funktionalität und sind daher schwer zu verstehen und zu warten. Auch der Entwurf derartiger Klassen ist nicht einfach und daher fehleranfällig (vgl. [Chidamber94]).
Lines of Code (LOC)	Zählt die Quelltextzeilen	Metrik mit zweifelhaftem Wert, siehe Erläuterungen einige Seiten zuvor
Number of Methods (NOM)	Zählt die Methoden einer Klasse	Aus Gründen der Übersichtlichkeit sollten Klassen mit sehr vielen Methoden (z. B. > 15) einem Refactoring unterzogen werden.
Weighted Methods per Class (WMC)	Ermittelt die Komplexität einer Klasse aus der Summe der CC-Werte aller Methoden	Erlaubt eine ähnliche Aussage wie CC auf Klassenebene

Auswahl geeigneter automatisierter Metriken

Das GQM-Verfahren gilt im Prinzip auch für automatisch ermittelte Metriken. In der Praxis wird man jedoch Kompromisse eingehen müssen. Da in der Regel vorhandene Werkzeuge zur Durchführung der Messungen eingesetzt werden, ist man auf die dort vordefinierten Metriken angewiesen. Was nun keinesfalls passieren sollte, ist der Einsatz aller im Werkzeug verfügbaren Metriken, frei nach dem Motto »Viel hilft viel«. Da gerade bei automatisierten Metriken die richtige Interpretation schwierig ist, sollte man sich auf die für das Projekt wirklich wichtigen Messwerte beschränken. Hier schließt sich der Kreis zum GQM-Verfahren. Ausgehend von den Zielen wählt man aus den im Werkzeug angebotenen Metriken diejenigen aus, welche die Fragen zur Zielerreichung am besten beantworten.

Überwachung nichtfunktionaler Anforderungen

Gut geeignet zur Überwachung durch automatisierte Metriken sind nichtfunktionale Anforderungen, wie z. B. die einfache Wartbarkeit und Erweiterbarkeit. Systeme sind dann einfach zu warten und zu erweitern, wenn die einzelnen Module nicht zu komplex sind (vgl. [Banker93] und [Gill91]). Gleichzeitig reduziert eine geringe Komplexität die Fehleranfälligkeit eines Moduls; es lohnt sich also, diesen Aspekt während der Entwicklung des Produktes im Auge zu behalten. Die Fragestellung lautet dementsprechend: Gibt es Module im System, deren Komplexität zu hoch ist?

Verwendung der Metrik »Cyclomatic Complexity«

Diese Frage kann für viele Projekte zumindest indirekt mit der Metrik *Cyclomatic Complexity* (CC) beantwortet werden. Die etwas vereinfachte Aussage der Metrik lautet, dass eine Methode umso verständlicher und einfacher zu testen ist, je geringer der gemessene Wert ist[8]. Dementsprechend ist eine Klasse dann problematisch, wenn sie eine Methode mit einem zu hohen CC-Wert beinhaltet.

Nur, welcher CC-Wert ist »zu hoch«? Dieser Schwellwert kann nur auf empirischer Basis ermittelt werden. Idealerweise bestimmt man hierzu die CC-Werte von problematischen Methoden aus früheren Projekten und leitet aus den Ergebnissen den Schwellwert für neue Projekte ab. Ich habe dies beispielhaft für eines meiner größeren Java-Projekte durchgeführt. Bei der Analyse der Ergebnisse habe ich festgestellt, dass die mir bekannten problematischen Module mit schöner Regelmäßigkeit einzelne Methoden mit einem CC-Wert von über 10 enthalten haben. Diesen Schwellwert würde ich daher ohne Weiteres für ähnlich gelagerte Projekte wieder verwenden.

Grenzen der CC-Metrik

Allerdings, und das muss man sich wirklich immer wieder klarmachen, sind Metriken wie der CC-Wert nur *ein Indikator* und kein endgültiger Nachweis für den Grad an Übersichtlichkeit, Fehleranfälligkeit und Wartbarkeit eines bestimmten Moduls. Viele andere Aspekte eines Softwaresystems, wie beispielsweise die verwendete Architektur und die Komplexität der fachlichen Domäne, werden durch Metriken überhaupt nicht erfasst. Ein schönes Beispiel hierfür ist der Quelltext von Maven. Dieser liefert durchgehend sehr niedrige CC-Werte. Trotzdem ist Maven ein komplexes Produkt, nicht zuletzt deshalb, weil intern ein sogenannter *Dependency Injection Container* verwendet wird. Dieser lagert sehr viele Abhängigkeiten zwischen den Modulen von Maven in externe Konfigurationsdateien aus. Dies bringt eine Reihe von Vorteilen, erschwert aber gerade unerfahrenen Entwicklern das Verständnis des Quelltextes ungemein. Die niedrigen CC-Werte führen in diesem Fall also in die Irre.

8. Da mit CC die Anzahl der Ausführungspfade gemessen wird, entspricht dies gleichzeitig der Anzahl der benötigten Testfälle, um eine Methode vollständig abzusichern.

Psychologische Konsequenzen im Projekt

Metriken haben viele Vorteile und können helfen, die Qualität eines Softwareproduktes dauerhaft zu sichern. Allerdings gilt leider auch das Gegenteil, wenn man gerade die automatisierten Metriken unüberlegt verwendet. Bezeichnenderweise stellen die Autoren von [Ebert05] fest, dass die Einführung von Metriken eindeutig eher vom Management als von den Entwicklern als positiv bewertet wird. Dies deckt sich mit meinen Erfahrungen. Entwickler empfinden Metriken als eine stark vereinfachende Bewertung ihrer Arbeit. Ein hochkomplexes Softwareprodukt wird auf wenige Zahlen reduziert. Liegt eine dieser Zahlen oberhalb eines mehr oder weniger willkürlich festgelegten Schwellwertes, müssen sie sich im schlimmsten Fall vor der Projektleitung dafür rechtfertigen. Oft sind hohe Messwerte gut zu begründen, doch welcher Manager hat die Muße, sich von einem Entwickler über die technischen Feinheiten einer bestimmten Methode aufklären zu lassen? In der Folge werden zu hohe Messwerte oft schlicht als Fehler gewertet. Dies ist natürlich ungerechtfertigt und demotiviert jedes Entwicklungsteam in kürzester Zeit.

Best Practices beim Einsatz von Metriken

Im Folgenden habe ich kurz zusammengefasst, was aus meiner Sicht bei der Verwendung von Metriken im Projekt unbedingt beachtet werden sollte:

- Gezielte Auswahl von Metriken nach dem GQM-Verfahren *am Anfang* des Projektes. Keinesfalls dürfen mitten im Projekt willkürlich ausgewählte Metriken eingeführt werden. Die Schwellwerte in den gekauften Werkzeugen entsprechen beinahe mit Sicherheit nicht den Anforderungen Ihres Projektes. Eine Messung würde daher zu ellenlangen »Fehlerlisten« und in der Folge zu erheblicher Aufregung im Projekt führen.
- Ausführliche Dokumentation der verwendeten Metriken und Schwellwerte im KM-Handbuch. Jeder Projektbeteiligte muss eine Chance haben, den Sinn und Zweck einer bestimmten Messung zu verstehen.
- Verwendung *realistischer* Schwellwerte. Niemandem ist geholfen, wenn beispielsweise ein CC-Schwellwert von 5 gefordert wird und in der Folge unzählige Mini-Methoden entstehen.
- Ausnahmen müssen explizit möglich sein. Metriken liefern nur Anhaltspunkte, keine absoluten Aussagen. Wenn ein Modul für eine bestimmte Metrik den Schwellwert nicht einhalten kann, ist dies nicht unbedingt ein Hinweis auf ein Qualitätsproblem. Sehen Sie daher im KM-Handbuch die Möglichkeit vor, bestimmte Module – mit Begründung – von den Messungen auszuschließen.

- Und am wichtigsten: Metriken liefern nur Hinweise auf *mögliche* Probleme. Wenn ein Schwellwert verletzt wird, setzen Sie ein Quelltext-Audit an und verschaffen Sie sich ein genaues Bild von der Lage. Fordern Sie keinesfalls einfach die »Behebung des Fehlers«!

2.3.4 Berichte

Die Ergebnisse der Audits, die ermittelten Metriken und generell der Status des Projektes werden regelmäßig in Form von Berichten veröffentlicht. Idealerweise sollte das gesamte Team Zugriff auf diese Berichte haben. Dies erreicht man beispielsweise durch die Einrichtung einer *Projekt-Homepage*. In der folgenden Tabelle habe ich den Aufbau einer typischen Projekt-Homepage dargestellt.

Menüpunkt	Beschreibung
Allgemeine Informationen	■ »Management Summary« des Projektinhaltes ■ Zeitplan und wichtige Meilensteine
Aktuelles	■ Neueste Informationen zum Projektverlauf ■ Kalender mit den demnächst anstehenden Terminen und Audits
Projekt-organisation	■ Überblick in Form einer Grafik ■ Tabelle mit den Kontaktdaten aller Teammitglieder
Wichtige Dokumente	■ Überblick aller wichtigen Dokumente im Projekt (z. B. Projekthandbuch, KM-Handbuch, Konzeptdokumente, Use-Case-Übersichten, Beschreibung der Softwarearchitektur) ■ Für alle Dokumente sollte ein direkter Zugriff von der Homepage aus möglich sein.
Projektstatus	■ Darstellung des Projektfortschritts mit Hilfe einer geeigneten Metrik ■ Detaillierter Status pro Team. Für das Entwicklerteam werden an dieser Stelle beispielsweise die Berichte mit den automatisch ermittelten Metriken angezeigt.

Tab. 2–6
Aufbau einer Projekt-Homepage

Abgesehen von der regelmäßigen Bereitstellung der aktuellen Berichte fällt die Erstellung und Pflege einer Projekt-Homepage nicht unbedingt in den Aufgabenbereich eines KM-Prozesses. In großen Projekten liegt die Verantwortung für die Homepage oft bei Stabsstellen, die generell für die Innen- und Außenkommunikation im Projekt verantwortlich sind. Kleinere und mittelgroße Projekte können sich einen derartigen Luxus nicht leisten, man muss die Projekt-Homepage also in Eigenregie erstellen und pflegen. In Kapitel 5 werden wir uns mit der Frage auseinandersetzen, inwieweit Maven uns in einem solchen Szenario unterstützen kann.