Fachhochschule Südwestfalen
University of Applied Sciences
Campus Hagen
Fachbereich Technische Betriebswirtschaft

Software Engineering

Materialsammlung zur Vorlesung
des ersten Teils des Moduls *Application Engineering*
für Wirtschaftsinformatiker des vierten Semesters

Andreas de Vries

Version: 3. Januar 2011

Inhaltsverzeichnis

1 Was ist Software-Engineering? 6
- 1.1 Die Softwarekrise . . . 6
- 1.2 Ursachen . . . 8
- 1.3 Definition . . . 9

2 Der Zyklus der Softwareentwicklung 11
- 2.1 Softwareentwicklung als Projekt . . . 13
  - 2.1.1 Das Wasserfallmodell . . . 13
  - 2.1.2 Das V-Modell: Projektmanagement und Qualitätssicherung . . . 15
  - 2.1.3 Scrum . . . 15
- 2.2 Softwareentwicklung als Prozess . . . 17
  - 2.2.1 Das Spiralmodell . . . 17
  - 2.2.2 Versionen-Entwicklung *(Versioning)* . . . 18
  - 2.2.3 Der Rational Unified Process (RUP) . . . 18
  - 2.2.4 Das Eclipse Process Framework (EPF) . . . 20
  - 2.2.5 Open-Source: das Basarmodell . . . 20
- 2.3 Psychologie des Software-Engineering . . . 22
  - 2.3.1 Agile Softwareentwicklung . . . 22
  - 2.3.2 eXtreme Programmierung (XP) . . . 23
- 2.4 Flexibles Rahmenmodell: V-Modell XT . . . 25

3 Analyse und Entwurf mit UML 27

4 Programmierleitsätze und *Best Practices* 28
- 4.1 Programmierleitsätze nach der Unix-Philosophie . . . 28
  - 4.1.1 Klein ist schön . . . 28
  - 4.1.2 Jedes Programm soll genau eine Sache gut machen . . . 29
  - 4.1.3 Erstelle so schnell wie möglich einen Prototyp . . . 29
  - 4.1.4 Portabilität geht über Effizienz . . . 29
  - 4.1.5 Speichere numerische Daten in ASCII-Dateien . . . 30
  - 4.1.6 Nutze die Hebelwirkung von Software zu deinem Vorteil . . . 30
  - 4.1.7 Vermeide Benutzeroberflächen, die den Benutzer gefangen halten . . . 30
- 4.2 Programmierleitsätze nach OpenSource . . . 31
- 4.3 Modularität und Schnittstellen . . . 31
  - 4.3.1 Plug-ins . . . 33
- 4.4 Code-Konventionen . . . 33

5 Kollaborative Softwareentwicklung 34
- 5.1 Grundsätzliche Problematiken . . . 34
  - 5.1.1 Merge . . . 34
  - 5.1.2 Branching, Trunk und Fork . . . 35

5.2 CVS . . . 35
5.2.1 Versionierung . . . 37
5.3 Subversion . . . 38
5.3.1 Revisionen . . . 38
5.3.2 *Tag*- und *Branch*-Konzept von SVN . . . 39
5.3.3 Der *Merge*-Prozess in SVN . . . 39

6 Technische Qualität: Softwaremetriken 40
6.1 Lines of Code (LOC) . . . 40
6.2 Zyklomatische Komplexität von McCabe . . . 42
6.3 Objektmetriken . . . 43
6.3.1 WMC . . . 43
6.3.2 DIT . . . 43
6.3.3 NOC . . . 43
6.3.4 CBO oder Ce . . . 44
6.3.5 RFC . . . 44
6.3.6 LCOM . . . 44
6.4 Analysemetriken . . . 44
6.5 Halstead-Maße . . . 44
6.6 Funktionspunktanalyse . . . 45

7 Softwarequalität 48
7.1 Qualität und Anforderungen . . . 48
7.2 Kosten und Nutzen von Softwarequalität . . . 49
7.3 Systematisches Testen . . . 50
7.3.1 Fehler . . . 50
7.3.2 Erstellen von Testfällen: *Black-Box* und *Glass-Box* . . . 51

Literaturverzeichnis 53

Einleitung

Die Erstellung von Software ist kein einfaches Geschäft. Im Gegensatz zu den meisten materiellen menschlichen Produkten wie Werkzeugen, Maschinen oder Fahrzeugen zeichnet sich Software durch hohe Komplexität und Flexibilität aus. Freilich ist es genau diese Flexibilität, die heute zunehmend die Grenzen zwischen den klassischen Ingenieurswissenschaften und der Informatik verschwimmen lässt. Wo ist denn genau der Unterschied zwischen einer Maschine, einem Roboter oder einem Computer?

So komplex das Produkt Software, so schwierig ist auch dessen Herstellung. Seit den 1960er Jahren gibt es Versuche, den Herstellungsprozess von Software mit ingenieursmäßigen Ansätzen anzugehen und zu planen. In diesem Geiste bildete sich der Name „Software Engineering“, und mit ihm eine neue wissenschaftliche Disziplin. Als wichtiges Element des Software-Engineering erwies sich von Anfang an das Projektmanagement, mit der Zeit entstanden jedoch zusätzlich spezifisch programmiertechnische Ansätze, die in klassischen Entwicklungsprozessen so gar nicht möglich waren, wie kollaborative Softwareentwicklung in großen Teams, insbesondere seit den 1990er Jahren ganz neue und hocheffiziente soziale Erstellungsprozesse im Zusammenhang mit Open-Source-Projekten mit tausenden von Entwicklern, vernetzt über das Internet.

Das vorliegende Skript über Software-Engineering behandelt den ersten Teils des übergreifenden Moduls *Application Engineering*. Das ist hier verstanden als diejenige Disziplin, die sich mit Techniken und Methoden zur arbeitsteiligen, ingenieursmäßigen Erstellung gebrauchstauglicher Anwendungs- und Informationssysteme befasst. Das *Application Engineering* ergänzt damit das Software-Engineering, also die systematische und effiziente *Erstellung* von Software, um das *Usability-Engineering*, d.h. die Untersuchung der *Gebrauchstauglichkeit* von Software. Das *Application Engineering* hat also zum Ziel, Methoden, Modelle, Prinzipien und Werkzeuge zu untersuchen, aus denen sich ein nicht nur effizientes und effizient hergestelltes, sondern auch ein ergonomisches und qualitativ hochwertiges Produkt namens Software ergibt. Diese beiden Aspekte, also effiziente Herstellung und ergonomische Gebrauchstauglichkeit von Software, lassen sich in der Praxis freilich nur schwer trennen, sie stellen lediglich zwei verschiedene, zeitlich und inhaltlich oft eng verwobene Perspektiven der Software-Entwicklung dar.

Grundbegriff des *Application Engineering*, und damit insbesondere des Software-Engineering für Wirtschaftsinformatiker, ist die *Applikation*, üblicherweise auch *Anwendungssystem* oder einfach *Anwendung* genannt. Das ist ein Softwareprodukt, das in bestimmten Anwendungsbereichen eingesetzt wird und Arbeits- oder Geschäftsprozesse unterstützt oder gar ausführt, beispielsweise in der Lagerhaltung, der Finanzverwaltung oder der Auftragsabwicklung. Ein Anwendungssystem kann eine *Individualsoftware* sein, die für einen spezielen Auftraggeber in einem bestimmten Anwendungsbereich erstellt wird („Maßanzug“), beispielsweise eine Webseite für eine Firma, oder eine *Standardsoftware*, die allgemein einen bestimmten Anwendungsbereich abdeckt und durch Konfiguartion *(„Customizing“)* an die unterschiedlichen Bedürfnisse der Anwender angepasst werden kann, z.B. Finanzbuchhaltungs- oder Warenwirtschaftssysteme, oder Office-Programme für Textverarbeitung und Tabellenkalkulation.

Oft steuern Anwendungssysteme technische Prozesse (Fertigungsstraßen, Flugüberwachung), mit sog. *eingebetteten Systemen (embedded systems)* wie Motormanagement im Automobil oder Steuerung des DVD-Players wird die Grenze zur an sich nicht mehr unter dem Begriff der Anwendungssoftware fallenden *Systemsoftware* wie Betriebssysteme oder Datenbanken überschritten, die sich keinem bestimmten Anwendungsbereich zuordnen lässt.

Natürlich befasst sich das Software-Engineering im Allgemeinen mit dem Herstellungsprozess jedweder Software, ohne Unterschied zwischen Anwendungs- oder Systemsoftware. Allerdings wird der Schwerpunkt der beruflichen Tätigkeit vorwiegend auf Anwendungssoftware liegen, wahrscheinlich allen Folgen der Globalisierung zum Trotz: Systemsoftware, auch Standardsoftware kann auch in Ländern mit billigerer Lohnstruktur und genügend hohem Bildungsniveau ausgeführt werden, die Konfiguration oder Erstellung von Anwendungssoftware wird auch in Zukunft die Kenntnis der lokalen wirtschaftlichen, juristischen und soziokulturellen Hintergründe erfordern. Und der Bedarf an Anwendungsentwicklung ist heute schon hoch, er wird zukünftig sicherlich steigen.

Hagen, den 3. Januar 2011 Andreas de Vries

Kapitel 1

Was ist Software-Engineering?

> *Wer sich an die Vergangenheit nicht erinnert, ist dazu verdammt, sie zu wiederholen.*
>
> George Santayana
>
> *Wer sich an die Vergangenheit nicht erinnert, ist dazu verdammt, deren Erfolge nicht zu wiederholen.*
>
> Barry Boehm, *ObjektSpektrum* **6**, 2008

Bevor wir uns der Definition des „Software-Engineering“, dessen Entwicklung und Methoden widmen, betrachten wir zunächst das historische Phänomen der „Softwarekrise“ Ende der 1960er Jahre, das die Ursache zu dessen Entstehung und Notwendigkeit besteht.

1.1 Die Softwarekrise

Auf einer NATO-Tagung 1968 in Garmisch-Partenkirchen wurde das Problem diskutiert, dass erstmalig die Kosten für die Software die Kosten für die Hardware überstiegen und die ersten großen Software-Projekte gescheitert waren. Dort wurde der Begriff des Software Engineering geprägt. Die Idee war, als Vorbild die ingenieursmäßige Herangehensweise an die Produktion zu übernehmen und somit Software als komplexes industrielles Produkt zu sehen.

Doch wo ist eigentlich das Problem? Zunächst einmal bleibt festzustellen, dass bis in die jüngste Vergangenheit Softwareprojekte grandios scheitern. Beispiele für die spektakulärsten (und teuersten) davon seien hier erwähnt [5, S. 410ff], [12, §1.2].

- *(Mars Climate Orbiter)* Im Jahr 1998 startete die NASA den Mars Climate Orbiter, eine Sonde, deren Mission darin bestand, einen Orbit um den Mars zu erlangen und von dort aus die klimatischen Bedingungen auf dem Mars zu erkunden. Der Orbiter erreichte am 23. September 1999 den Mars, ging dann aber verloren. Der später eingesetzte Untersuchungsausschuss fand heraus dass die Bahn des Orbiters um 170 km zu niedrig verlief, weil ein einzelnes Modul englische Längenmaße benutzte, während andere den Vorgaben der NASA entsprechend die international gebräuchlichen SI-Einheiten verwendeten. Unter den im Bericht des Untersuchungsausschuss aufgeführten Ursachen befand sich ein Vermerk über *unzureichende Kommunikation* zwischen den Entwicklern und den Programmierteams.

- *(Mars Polar Lander)* Auch der Mars Polar Lander, der sechs Wochen später auf dem Mars landen sollte, ging verloren. Der eingesetzte Untersuchungsausschuss identifizierte einen Softwarefehler als wahrscheinlichste Ursache des Verlusts. Hintergrund war, dass die Triebwerke des Landers sofort nach der Landung abgeschaltet werden

sollten, da er sonst umkippen würde. Sensoren an den Landefüßen des Polar Landers erzeugten noch vor dem Oberflächenkontakt ein Signal. Allerdings war bekannt, dass die Sensoren gelegentlich unechte Signale erzeugen konnten, und so war die Software so programmiert, dass sie solche Signale erkennen konnte, indem sie zwei aufeinander folgende Signale verglich und nur dann reagierte, wenn beide denselben Wert hatten. Nun konnte es jedoch zur Erzeugung längerer Kontaktsignale kommen, wenn die Landefüße von ihrer Stauposition in die Landeposition gebracht wurden. Dies ist an sich aber kein Problem, denn die Füße werden in einer Höhe von etwa 1500 Metern ausgefahren, während die Software die Maschinen nicht ausschaltet, solange die durch den Radar gemessene Höhe des Landers über der Oberfläche mehr als 40 Meter beträgt. Leider wurde nur das beim Ausfahren der Landefüße registrierte unechte Signal nicht gelöscht, so dass die Maschinen abgeschaltet wurden, als der Lander eine Höhe von 40 Metern erreichte, und er somit abstürzte und am Boden zerschellte. Der Bericht führte an:

> *„Dieses Verhalten war bekannt, und die Flugsoftware sollte diese Ereignisse ignorieren; jedoch beschrieben die Anforderungen diese Erignisse nicht spezifisch, so dass die Software-Entwickler nicht dafür verantwortlich gemacht werden können."*

Der Schaden belief sich auf etwa 230 Millionen US-$.

- *(Explosion der Ariane 5)* Etwa 40 Sekunden nach dem Start zu ihrem Jungfernflug am 4. Juni 1996 nahm die Rakete Ariane 5 der ESA einen so abrupten Kurswechsel vor, dass sie zerbrach und explodierte. Der Untersuchungsausschuss stellte fest, dass die in Teilen von der Ariane 4 übernommene Software nicht mehr den nötigen Anforderungen entsprach. Die Ariane 5 beschleunigt schneller als die Ariane 4. Dies führte zu einem Überlauf einer Variablen des Lenksystems, nämlich bei der Umwandlung einer 64-Bit-Gleitkommazahl für die horizontale Geschwindigkeit in eine vorzeichenbehaftete 16-Bit-Ganzzahl. Das Ergebnis war ein Programmabsturz des Lenksystems, was dazu führte, dass die Navigationsanlage nur noch Statusdaten an den Navigationscomputer sandte, die dieser als echte Fluglage interpretierte, die beträchtlich vom geplanten Kurs abwich, und so die Schubdüsen der Booster bis zum Anschlag schwenken ließ. Dadurch begann die Rakete auseinanderzubrechen.

  Glücklicherweise kamen keine Menschen ums Leben, doch der materielle Schaden belief sich auf etwa 500 Millionen US-Dollar, womit der Fehlstart einen der teuersten Computerfehler der Geschichte darstellt.

- *(Y2K)* Erstmals traten 1998 Fehlerfälle auf, bei denen Computer gültige Kreditkarten als abgelaufen ablehnten. Hintergrund war, dass die Jahreszahlen in den Programmen nur zweistellig abgespeichert wurden, so dass das Ablaufdatum im Jahr 2000 für das Jahr 1900 gehalten wurde. Tatsächlich wurde Ende der 1990er Jahre vermutet, dass der größte Teil der damals verwendeten Computerprogramme in allen Bereichen und Branchen an diesem „Jahr-2000"-Problem (Y2K = *year 2 kilo*) litten. Es gab im Vorfeld Befürchtungen, dass nach dem Jahreswechsel 1999/2000 Sicherheitssyteme von Atomkraftwerken oder Navigationssysteme von Flugzeugen versagen würden, Zinsen und Gebühren falsch berechnet werden, ja dass sogar das öffentliche Leben in großen Teilen zusammenbrechen würde. Am Ende passierte — nichts. Obschon es damals auch ironische Stimmen gab, die das entstandene Problembewusstsein als übertriebene Panikmache ansahen, gehören die weltweiten Anstrengungen zum Y2K-Problem zu den erfolgreichsten Softwareprojekten überhaupt. Für Software gilt nun einmal ähnlich wie für viele andere Dienstleistungen das *Gesetz der asymmetrischen Aufmerkdamkeit*: Wenn

sie ihren Zweck erfüllt, bemerkt es niemand, nur Fehler und Bugs fallen auf.[1] Man sollte sich nichts vormachen, das nächste Jahrtausendproblem mit wohl *allen* derzeitig laufenden Programmen ist zu lösen vor dem 31.12.9999!

- *(Fiskus)* 1991 wurde von den Fianzministern von Bund und Ländern das Softwareprojekt *Fiskus* begonnen, das spätestens ab 2006 den 650 Finanzämtern der Bundesrepublik Deutschland bringen sollte. Es wurde 2004 als gescheitert erklärt, nach internen Schätzungen beliefen sich die Kosten auf Beträge zwischen 250 und 900 Millionen Euro.

Untersuchungen der Standish-Gruppe aus den Jahren 1994 und 2004 versuchten, die Entwicklung der Ergebnisse von Softwareprojekten zu beziffern. Dabei wurden 1994 ins-

Jahr	erfolgreich	kritisch	aufgegeben
1994	16%	50%	31%
2004	29%	53%	18%

Tabelle 1.1: Bewertung von Softwareprojekten in den USA nach Untersuchungen der Standish Group. Hier bezeichnet „erfolgreich“ die Projekte, die im Zeit- und Budgetrahmen fertiggestellt wurden, „kritisch“ diejenigen, die den Zeit- oder Budgetrahmen erheblich überschritten (1994 um über 100%) oder mit erheblich reduziertem Funktionsumfang ausgeliefert wurden; Zahlen aus [12, S. 3–5].

gesamt 8 380 Projekte berücksichtigt, 2004 insgesamt 9 236 Projekte. Die Ergebnisse sind in Tab. 1.1 dargestellt. Ferner zeigte sich 2004, dass nur 4% der Projekte sich mit dem Kauf kompletter Anwendungssysteme ohne weitere Anpassungen befassten, alle anderen stellten Anpassungen (13%), Komponentenerstellung (22%) oder gar Neuentwicklungen (55%, davon objektorientiert 19%) dar.

1.2 Ursachen

> *“[The major cause of the software crisis is] that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.“*
>
> Edsger Dijkstra (1972)[2]

Allen Problemfällen der Softwarekrise gemeinsam ist die *Komplexität* der betreffenden Programme. Natürlich gibt es einen großen Unterschied zwischen Programmieraufgaben, die Sie im Laufe der ersten Semester zu lösen hatten, und Programmierprojekten, bei denen es um Millionen Zeilen Quelltext geht. So gehören moderne Betriebssysteme zu den komplexesten Produkten, die Menschen jemals erschaffen haben. Ein so großer quantitativer Unterschied bewirkt entsprechend einen *qualitativen* Unterschied in der Komplexität und erfordert eine vollkommen neue Handhabung. Eine einzelne Person kann vielleicht noch alle oder wenigstens wesentliche Details eines kleinen Problems überschauen. Wird aber das Problem umfangreicher, so wird es notwendig, im Team zu arbeiten und den Zweck der verschiedenen Komponenten des Projekts und deren Beziehungen untereinander schriftlich festzuhalten und zu protokollieren. In der Regel müssen bereits erstellte Softwaremodule

[1] Entsprechend wird ein Zugschaffner wohl nur zum Zeitplan seines Zuges angesprochen, wenn dieser *nicht* eingehalten wird. Ein Dankeschön nach pünktlicher Ankunft würde höchstwahrscheinlich als ironisch aufgefasst.

[2] *http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF*

eingebunden werden, so dass eine Dokumentation über ihre Funktionen und ihre Schnittstellen unumgänglich ist. Ein weiteres Mittel zur Bewältigung der Komplexität sind die Anwendung von Standardisierungen, Normen oder Konventionen, z.B. ISO-Normen für Prozesse oder zur Qualitätskontrolle, festgelegte Struktur von Programmen, oder Namenskonventionen.

Oft trägt eine mangelnde Qualitätssicherung zum Scheitern von Softwareprojekten bei. Ebenso kann eine schlechte Projektorganisation verantwortlich für das Scheitern von Projekten sein wie ungenügende Einbeziehung des Anwenders und damit einhergehender Know-How-Verlust, Zeitdruck oder fehlerhafte Zielvorstellungen. Oft ist der Projekttreiber auch nicht ein Anwender oder die Geschäftsführung, sondern die IT-Abteilung, was potentiell zu nicht an den Anwenderbedürfnissen orientierten Ergebnissen und letztlich zu Akzeptanzproblemen führt.

1.3 Definition

Was ist nun genau unter dem Begriff „Software-Engineering“ zu verstehen? Dazu müssen wir zunächst wissen: Was ist Software? Software ist ein vielfältiges Produkt. Es setzt sich zusammen aus verschiedenen Teilprodukten: Analysemodell, Architekturmodell, Programme, Code, ferner gehören dazu Dokumentationen, Benutzerhandbuch, Hilfesystem, Testdaten, Review-Ergebnisse usw. *Software Engineering*, oder *Softwaretechnik*, beschäftigt sich mit der systematischen Erstellung von Software. Es ist die *„zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden und Werkzeugen für die arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Softwaresystemen.“* [1, S. 36] Etwas detallierter formuliert untersucht und entwickelt das Software-Engineering Prinzipien, Methoden, Techniken und Werkzeuge, um Software

- mit einem festgelegten Funktionsumfang
- in ausreichender Qualität
- innerhalb eines gegebenen Budgetrahmens
- zum geplanten Termin

zu erstellen. Software Engineering behandelt also nicht nur die eigentliche Entwicklung von Software, sondern auch Fragen der Qualitätssicherung von Software und des Projektmanagements.

Nach dem offiziellen *Guide to the Software Engineering Body of Knowledge* [swebok] (Handbuch des gesammelten Wissens zum Software-Engineering, kurz: SWEBOK) der IEEE Computer Society umfasst das Software-Engineering die folgenden zehn Teildisziplinen:

- Anforderungsermittlung *(software requirements)*
- Entwurf *(software design)*
- Implementierung *(software construction)*
- Testen *(software testing)*
- Wartung *(software maintenance)*
- Konfigurationsmanagement *(software configuration management)*
- Projektmanagement und Metriken *(software engineering management)*

- Vorgehensmodelle *(software engineering process)*
- Werkzeuge und Methoden *(software engineering tools and methods)*
- Softwarequalität *(software quality)*

Demnach ist Software Engineering eine sehr fassettenreiche Disziplin, die Querbezüge zu den Ingenieurs- und Wirtschaftswissenschaften, aber auch zur Kommunikations- und Arbeitspsychologie hat. In diesem Skript werden nur einige dieser Themen angerissen werden können.

Kapitel 2

Der Zyklus der Softwareentwicklung

Um Managementmethoden für das Software-Engineering entwickeln zu können, muss man zuerst den Prozess der Softwareentwicklung verstehen. Ein wesentlicher Punkt dazu ist das Vorgehensmodell. Ein *Vorgehensmodell*, auch: *Lebenszyklusmodell*, gliedert im Allgemeinen einen Gestaltungs- oder Produktionsprozess in strukturierte *Aktivitäten* oder *Phasen*, denen wiederum entsprechende Methoden und Techniken der Organisation zugeordnet sind. Aufgabe eines Vorgehensmodells ist es, die allgemein in einem Gestaltungsprozess auftretenden Aufgabenstellungen und Aktivitäten in ihrer logischen Ordnung darzustellen: *Wer macht was zu welcher Zeit und mit welchem Ergebnis?* Im Rahmen eines Projektmanagements wird eine Aktivität oft durch einen *Meilenstein* beendet. Die Begriffe „Phase“ und „Aktivität“ sind oft nicht trennscharf definiert, im Projektmanagement wird ein Projekt üblicherweise in Phasen eingeteilt [6, S. 10],

$$\boxed{\text{Vorphase}} \rightarrow \boxed{\text{Analyse}} \rightarrow \boxed{\text{Konzeption}} \rightarrow \boxed{\text{Realisierung}} \rightarrow \boxed{\text{Test}} \rightarrow \boxed{\text{Abnahme}},$$

während ein Software-Projekt meist eher als Entwicklungsprozess verstanden wird, der aus einzelnen Aktivitäten besteht. In der Softwareentwicklung unterscheidet man gewöhnlich die folgenden Aktivitäten oder Phasen [1, §I.2.1], [6, §1.2], [12, §3.3]:

Aktivität		**Ergebnis/Meilenstein**
Anforderungsdefinition	*(Requirements)*	Lastenheft
Analyse	*(Specifications)*	Pflichtenheft, Sollkonzept
Entwurf	*(Design)*	UML-Diagramme, Algorithmen
Implementierung	*(Implementation)*	Programme
Funktionsprüfung	*(Testing)*	Abnahme
Inbetriebnahme	*(Deployment)*	Installation der Software, Schulungen
Wartung & Support	*(Maintenance)*	

Software wird für Anwender erstellt. Das ist eine gegebene Zielgruppe, beispielsweise ein Kunde, ein System, vielleicht auch eine Maschine. Wie im Software Engineering üblich wird der Anwender hier im Folgenden kurz *Kunde* genannt.

- **Anforderungsdefinition**, oft auch **Bedarfserfassung**. In diesem Prozess wird ermittelt, was der Kunde tatsächlich braucht. Da bei dem Kunden im Allgemeinen spezielle Informatikkenntnisse nicht vorausgesetzt werden können, und andererseits bei den Entwickler keine Kenntnisse aus dem Anwendungsbereich, ist die Hauptaufgabe der Anforderungsdefinition die Klärung der Anforderungen und, im tieferen Sinne, Aufbau und Aufrechterhaltung von Kommunikation und Wissensermittlung.

  Manche Software wird *angebotsorientiert* erstellt, also ohne einen individuellen Kunden im Sinn. Beispiele dafür sind kommerzielle Softwareprodukte wie ERP-Systeme,

Betriebssysteme, oder Spielesoftware. In diesen Fällen werden die Kundenbedürfnisse antizipiert.

- **Analyse**, oft auch **Anforderungsspezifikation**. In dieser Phase werden die Ergebnisse der Anforderungsdefinition möglichst exakt festgeschrieben. Die Spezifikation (oder das Pflichtenheft) soll nur beinhalten, *warum* die Software erstellt wird und *was* sie leisten soll (und das möglichst präzise), nicht aber, *wie* sie es schaffen soll.

  Grundproblem der Analyse ist, dass sie zwei gegensätzliche Ziele erfüllen soll. Auf der einen Seite muss sie informell, intuitiv und für Nicht-Informatiker verständlich sein, damit der Kunde validieren kann: *„Wir erstellen das richtige Produkt."* Auf der anderen Seite muss sie präzise genug sein, damit Entwickler und Tester stets verifizieren können: *„Wir erstellen das Produkt richtig."*

- **Entwurf**. In diesem Prozess wird das Problem in Module zerlegt und geeignete Algorithmen und Datenstrukturen ermittelt und modelliert. Hierzu ist ein tiefes algorithmisches Verständnis notwendig.

- **Implementierung**. In diesem Prozess wird der Entwurf in spezifische Programme umgesetzt. Jedes Modul wird separat für sich implementiert, am Ende werden die Module zu einem vollständigen System zusammen gefügt. Dieser Teilschritt heißt *Integration*.

- **Funktionsprüfung, Verifikation**. Bei diesem Prozess wird verifiziert, ob jedes einzelne Modul (**Modultest**) und das gesamte System (**Integrationstest**) die Spezifikationen erfüllt.

  Die Verifikation erfordert eine formale Spezifikation, gegenüber der das Programm geprüft werden soll. Allerdings ist die Übersetzung der *tatsächlichen* Anforderungen und Bedürfnisse in eine formale Spezifikation ein hartes Problem und mindestens so fehleranfällig wie die Programmierung.

- **Inbetriebnahme, Installation**. Die Inbetriebnahme umfasst alle Tätigkeiten, die die Installation der Software betreffen, ggf. Beta-Tests oder Lösen von Migrationsproblemen bei der Ablösung von Altsystemen *(legacy systems)*, aber auch Schulungen oder Unterstützung der Anwender.

- **Wartung**. Anders als mechanische Systeme nutzen Softwaresysteme nicht mit der Zeit ab. Ein Programm muss nicht nach 300 000 km oder 10 Jahren einer Überholung unterzogen werden, es wird auf ewig so funktionieren wie am ersten Tag. Das ist das Problem! Damit Software nützlich bleibt, muss sie modifiziert werden. Modifikationen können nötig werden, weil die Hardware ausgetauscht wird, weil gesetzliche Bestimmungen sich geändert haben, die Anforderungen gestiegen oder Sicherheitslücken bzw. Fehler („Bugs") aufgetreten sind. Solche Modifikationen werden üblicherweise durch *Patches* („Flicken") durchgeführt, also punktuelle Änderungen der Software.

  Häufig kann der bemerkenswerte rekursive Effekt beobachtet werden, dass die neu eingesetzte Software den Bereich verändert, in dem sie eingesetzt wird. Beispielsweise könnte die zur Beschleunigung seiner Arbeitsabläufe eingeführte Software bei einem Steuerberater dazu führen, dass er neue Kunden gewinnt. Damit ist er nun aber von der Software abhängig, er kann nicht mehr ohne sie auskommen.

Statistiken zeigen, dass der größte Teil der Ausgaben bei Software im Wartungsbereich liegen, oft bis zu 80% der Gesamtkosten [5, S. 419].

Der internationale Standard zur Methodenbeschreibung von Auswahl, Implementierung und Überwachung des Lebenszyklus von Software ist ISO 12207.

Prototyping

Oft wird vor der Entwurfsphase ein Prototyp der des Softwaresystems oder wichtiger Systemteile erstellt, die Anforderungsspezifikation zu validieren. Der Prototyp wird so lange überarbeitet, bis Kunde bzw. Anwender zustimmen.

Ein Prototyp veranschaulicht nur grundsätzliche Aspekte der Software. Typisch ist die Erstellung von Prototypen der Benutzeroberfläche *(horizontale Prototypen)*, um bestimmte Aspekte der Benutzeroberfläche zu beleuchten; im Extremfall können solche Prototypen aus Papierskizzen bestehen, zur Verdeutlichung ganzer Abläufe können Sequenzen von solchen Oberflächenskizzen *(Storyboards)* ähnlich einem Comic Strip erstellt werden. Bei einem *Funktionsprototyp (vertikaler Prototyp)* dagegen wird ein Ausschnitt der geforderten Funktionalität realisiert, der alle Schichten der Software überdeckt, beispielsweise von der Benutzeroberfläche bis zur Datenbankanbindung zur Überprüfung bestimmter Anforderungen an die Performanz einer Stammdatenverwaltung.

Häufig werden Prototypen im *Rapid Prototyping* realisiert, also „quick and dirty“, um sich schnell Klarheit über einen bestimmten Aspekt zu verschaffen und Aufwand zu sparen,

Man kann den Prototyp beim Übergang zum Entwurf wegwerfen *(Wegwerfprototyp)* oder aber weiterentwickeln und in das Endprodukt einfließen lassen. Untersuchungen deuten darauf hin, dass die Qualität des Endprodukts bei Wegwerfprototypen höher ist [12, S. 22].

2.1 Softwareentwicklung als Projekt

2.1.1 Das Wasserfallmodell

Das einfachste und grundlegenste Vorgehensmodell ist das *Wasserfallmodell*. Ihm gemäß werden die oben genannten Phasen der Softwareentwicklung streng nacheinander abgearbeitet (Abb. 2.1). Dem Wasserfallmodell unterliegt die Grundannahme, dass die Erstellung

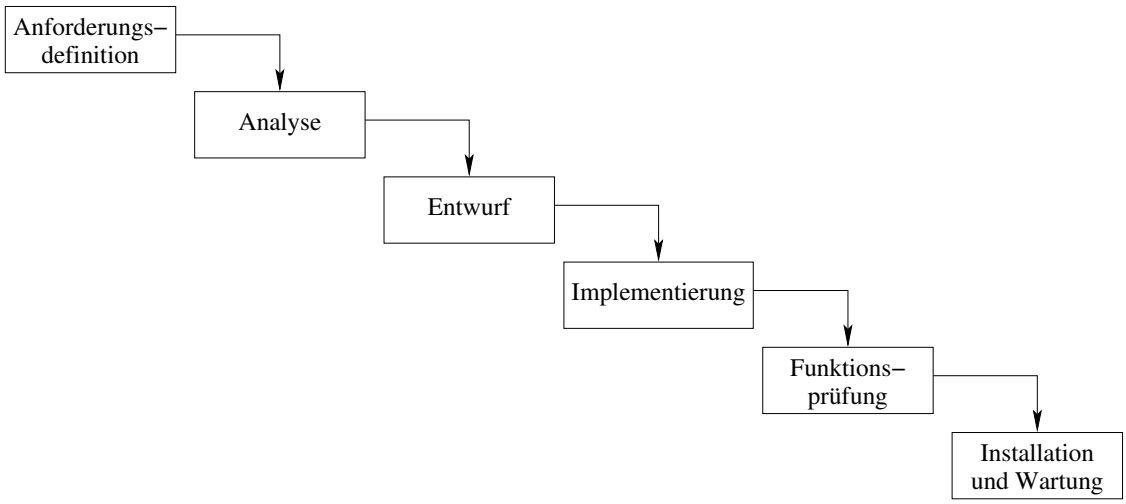

Abbildung 2.1: Das Wasserfallmodell

der Software ein *Projekt* ist. Jede der Phasen kann zeitlich mit Terminen für ihren Meilenstein, also ihre Beendigung, versehen werden. Für das Projektmanagement wird es damit möglich, durch die zeitliche Planung und Ressourcenzuteilungen zu den einzelnen Phasen Kosten und Termine abzuschätzen.

Insgesamt ist das Wasserfallmodell (in seiner strikten Auslegung) ein *dokumentorientiertes* Vorgehensmodell, denn zu jeder Entwicklungsphase gibt es eine klare Definition darüber,

welche Produkte oder *Ergebnisdokumente* nach ihrem Abschluss verfügbar sein müssen. Solche Ergebnisdokumente können Programme oder Dokumentationen sein und dienen als Kontrollpunkte des Gesamtprozesses. Die Projektleitung, und ggf. der Kunde, können damit den Fortschritt des Prozesses anhand der Ergebnisdokumente verfolgen.

Das Modell hat neben diesen hinsichtlich des Projektmanagements positiven Eigenschaften aber auch einen gravierenden Nachteil. Damit eine Phase begonnen werden kann, muss die vorherige Phase vollständig erledigt sein. So muss der Anforderungskatalog komplett erstellt sein, bevor man mit dem Entwurf beginnen kann. Umgekehrt ist damit der Anforderungskatalog endgültig eingefroren, wenn die Entwurfsphase gestartet ist, und kann nicht mehr verändert werden. Oft wird das Problem jedoch in den ersten Phasen nur vage verstanden, oder wesentliche Detailprobleme oder auch Denkfehler werden erst in den späteren Phasen entdeckt. Während der Entwurfsphase oder der Implementierung wächst häufig erst das Verständnis für das Problem. Somit gibt es häufig, und gerade bei neuartigen und innovativen Systemen, falsche Entscheidungen in den ersten Phasen, die später korrigiert werden müssen. Löst man jedoch das Prinzip der Unumkehrbarkeit der Phasenentscheidungen auf, so wird das Projekt schlechter planbar, und ein anfänglich angesetztes Budget kann schnell aus dem Ruder laufen.

Das Wasserfallmodell eignet sich also besonders, wenn die Entwickler (!) eine hinreichend genaue Vorstellung des Problembereichs haben, beispielsweise wenn ein erfahrenes Softwareteam ein neues Produkt ausarbeitet, das so ähnlich ist wie eines, das sie bereits zuvor entwickelt haben. Bei besonders innovativen oder einfach noch nicht hinreichend gut verstandenen Systemen dagegen ist das Wasserfallmodell eher ungeeignet.

Das *modifizierte Wasserfallmodell* war eine frühe, recht naheliegende Anpassung des Vorgehensmodells an diese Schwierigkeiten (Abbildung 2.2). Es ermöglicht die Rückkopplung

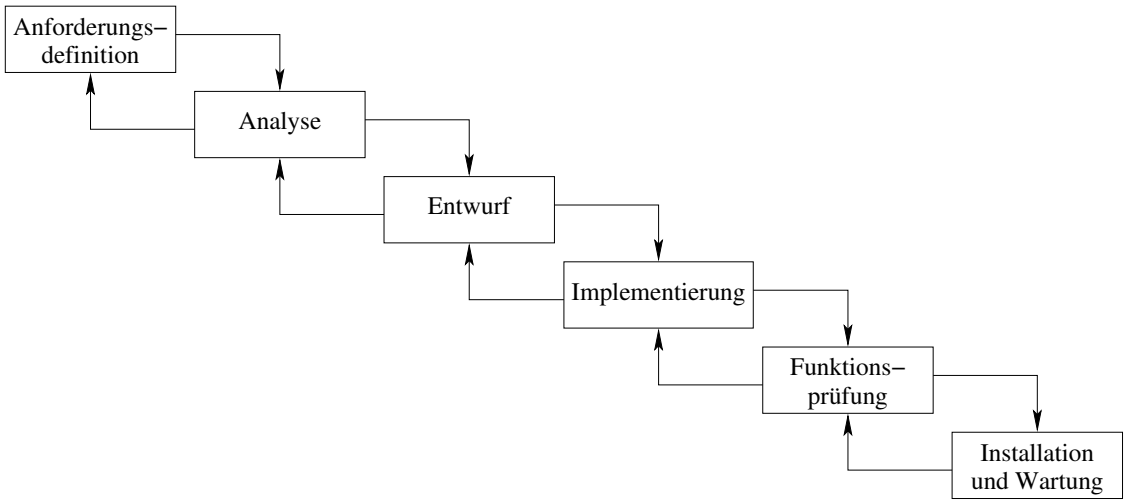

Abbildung 2.2: Das modifizierte Wasserfallmodell

zumindest zur jeweils nächstfrüheren Phase, falls Fehler aufgetreten sind. Wird dieses Vorgehensmodell konsequent eingesetzt, so kann schlimmstenfalls der Albtraum eines klassischen Projektmanagers eintreten und bereits erreichte Meilensteine revidiert werden und erneut zu erlangen sind.

Jede Softwareentwicklung bewegt sich im Spannungsfeld dieses Zielkonflikts zwischen Plan- und Budgetierbarkeit einerseits und Anforderungserfüllung und Qualität andererseits. Die folgenden Abschnitte beschreiben Ansätze zu realisierbaren Lösungen dieses Zielkonflikts.

2.1.2 Das V-Modell: Projektmanagement und Qualitätssicherung

Das *V-Modell* entstand in den 1980er Jahren aus dem Wasserfallmodell und legt verstärkten Wert auf Qualitätssicherung. Zu jeder Phase des Projekts werden entsprechende Testphasen definiert, sowohl die Anforderungs- bzw. Entwurfsspezifikationen *verifizieren* als auch die vorgesehene Nutzbarkeit der Produkte *validieren*. Die Verifikation hinterfragt also *„Wurde*

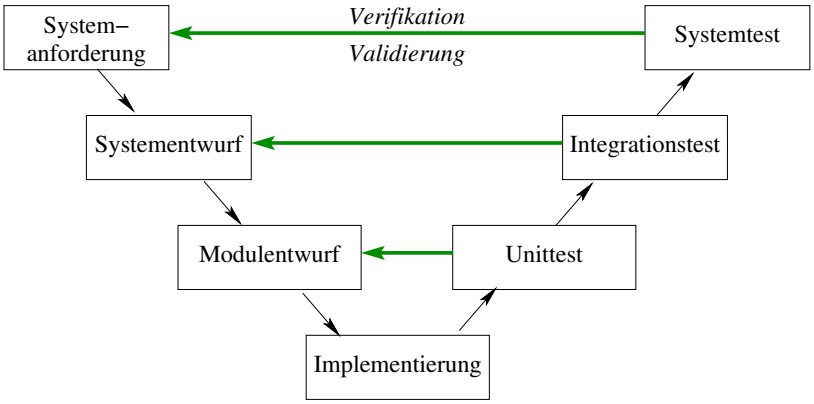

Abbildung 2.3: Das V-Modell

das Produkt richtig entwickelt?“, die Validierung dagegen *„Wurde das richtige Produkt entwickelt?“*

Im Unterschied zum Wasserfallmodell werden Testfälle nicht erst gegen Ende des Projekts spezifiziert, sondern bereits dann, wenn die Anforderungs- bzw. Entwurfsspezifikationen fertig gestellt sind. In der Abbildung 2.3 bedeutet das, dass die Testfälle nicht erst im rechten Ast erstellt werden, sondern bereits im linken.

2.1.3 Scrum

Scrum („Gedränge“) ist ein Vorgehensmodell des Projektmanagements, das ursprünglich aus Modellen der Produktfertigung der 1980er Jahre stammt und Methoden der *Lean Production* wie TPS ähnelt. Die Grundlagen von Scrum liegen im Wissensmanagement. Scrum geht von der Grundannahme aus, dass Produktions- und Entwicklungsprozesse zu komplex sind, um sie durch Vorgehensmodelle mit abgeschlossenen Phasen wie das Wasserfallmodell oder das V-Modell zu beherrschen. Stattdessen geht Scrum davon aus, dass es produktiver ist, lediglich einen groben Rahmen vorzugeben und das Team sich selbst organisieren zu lassen.

Scrum kennt die drei Rollen und besteht aus drei Spielphasen [6, §2.7]. Die Rollen lauten:

1. Der *Product Owner.* Er ist er Kunde und legt das gemeinsame Ziel fest, das das Team zusammen mit ihm erreichen muss, und in den jeweiligen Iterationsschritten die Prioritäten der entsprechenden Teilprodukt-Anforderungen (Product Backlogs, s.u.). Zur Definition der Ziele dienen ihm *(User) Stories*, d.h. in Alltagssprache mit maximal zwei Sätzen bewusst kurz gefasste Software-Anforderungen, oder andere Techniken wie UML Anwendungsfälle.

2. Das *Team*. Es besteht idealerweise aus 7 ± 2 Personen, schätzt die Aufwände der einzelnen Anforderungen (Backlog-Elemente) ab und implementiert die für den nächsten Iterationsschritt (Sprint, s.u.) machbaren Elemente.

3. Der *Scrum Master*. Er überwacht die Aufteilung der Rollen und Rechte zu überwachen, hält die Transparenz während der gesamten Entwicklung aufrecht und unterstützt dabei, Verbesserungspotentiale zu erkennen und zu nutzen. Er steht dem Team zur Seite,

ist aber weder Product Owner noch Teil des Teams. Er versucht, für die ordnungsgemäße Durchführung und Implementierung im Rahmen des Projektes zu sorgen. Er hat die Pflicht, darauf zu achten, dass der Product Owner nicht in den adaptiven Selbstorganisationsprozess des Teams eingreift.

Die Spielphasen sind:

1. Die *Projektgrobplanung*. Es werden die Vorgaben für das Projekt definiert, beispielsweise die Projektzusammensetzung, die verwendeten Entwicklungswerkzeuge und Standards (z.B. Programmierrichtlinien). Ferner erstellt der Product Owner in dieser Phase das *Product Backlog*, das die Anforderungen an das zu entwickelnde Produkt enthält.

2. Die *Game-Phase*. Sie besteht aus mehreren *Sprints*, d.h. Iterationsschritten der Dauer von etwa 30 Tagen, die als Ergebnis jeweils eine lauffähige Software haben und in denen das Team grundsätzlich frei entscheidet, wie es arbeitet, d.h. in denen der Product Owner keinen Einfluss nehmen darf. Vor jedem Sprint allerdings werden gemeinsam mit dem Product Owner in dem *Product Backlog* die Produktanforderungen und Priorisierungen besprochen und modifiziert, und zu Beginn eines jeden Sprints mit ihm in einem *Sprint Planning Meeting* das Sprintziel festgelegt. Der Scrum-Master be-

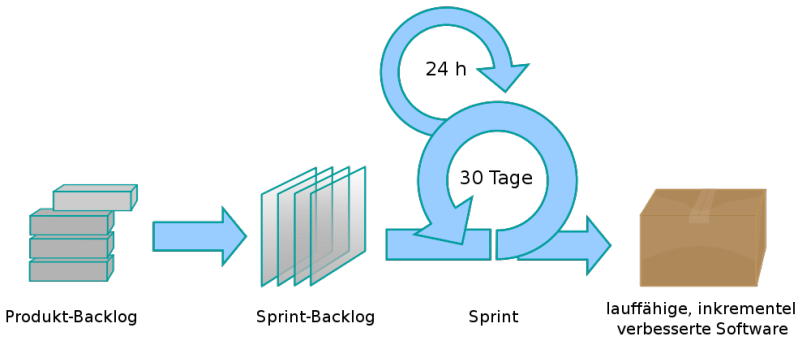

Abbildung 2.4: Sprint-Prozess. Quelle: *http://de.wikipedia.org/wiki/Scrum* [2010-09-23]

stimmt danach mit dem Team, wie das Sprintziel zu erreichen ist. Daraus ergibt sich das *Sprint-Backlog*, in dem festgehalten wird, was genau im kommenden Sprint umzusetzen ist. In einem täglichen etwa 15-minütigem, meist nach dem Mittagessen (und oft im Stehen) stattfindenden *Daily Scrum* oder *Stand-Up Meeting* wird bestimmt, welche Themen aus dem Sprint-Backlog von wem umgesetzt werden und in dem jeder die folgenden Fragen zu beantworten hat: Was habe ich seit dem letzten Meeting erreicht? Was plane ich bis zum nächsten Meeting umzusetzen? Was behindert mich bei meiner Arbeit? Im Daily Scrum haben nur die Teammitglieder und der Scrum-Master Rederecht. Am Ende eines Sprints wird dem Product Owner im *Sprint Review Meeting* das Sprintergebnis vorgestellt und von ihm entschieden, ob es einen weiteren Sprint geben muss. In diesem Falle wird das Product Backlog überarbeitet, also Anforderungen hinzugefügt, geändert oder umpriorisiert, und der nächste Sprint gestartet.

3. Die *Post-Game-Phase*. Ziel dieser Phase ist die Überführung des fertigen Systems in die Produktion beziehungsweise Anwendung. Typischerweise gehören hierzu die Erstellung der Dokumentation, die Durchführung von Systemtests und Akzeptanztests und die Auslieferung des Systems.

Scrum ist ein gut strukturiertes und dennoch felxibles Vorgehensmodell. Das gesamte Team trägt die gemeinsame Verantwortung für das Endprodukt. Durch die täglichen Treffen, die Daily Scrums, bewirken eine gute Abstimmung und eine hohe Projekttransparenz innerhalb

des Teams und fördern damit eine Fokussierung auf das Sprintziel. Durch die verhältnismäßig kurze Sprintdauer von etwa 30 Tagen mit jeweils mess- und überprüfbarem Ergebnis ergibt sich eine hohe Planungssicherheit.

2.2 Softwareentwicklung als Prozess

Zu den komplexesten, und dennoch im Verhältnis zu ihrer Komplexität am besten funktionierenden Softwaresystemen gehören die modernen Betriebssysteme oder große Open-Source-Projekte wie Apache, Firefox oder Thunderbird. Die Entwicklung dieser Systeme wird jedoch an sich nicht als zu einem Zeitpunkt zu beendendes Projekt verstanden, sondern radikal anders als Prozess, der in verschiedenen Versionsstufen evolutionär und offen verläuft.

2.2.1 Das Spiralmodell

Das Spiralmodell wurde von Boehm eingeführt [3]. Grundsätzlich ermöglicht es auch ein Vorgehen mit offenem Ende. Es wird in der Literatur nicht immer einheitlich interpretiert, hier verwenden wir die Darstellung von Balzert [1, S. 130]. Ein Phasenzyklus wird hierbei

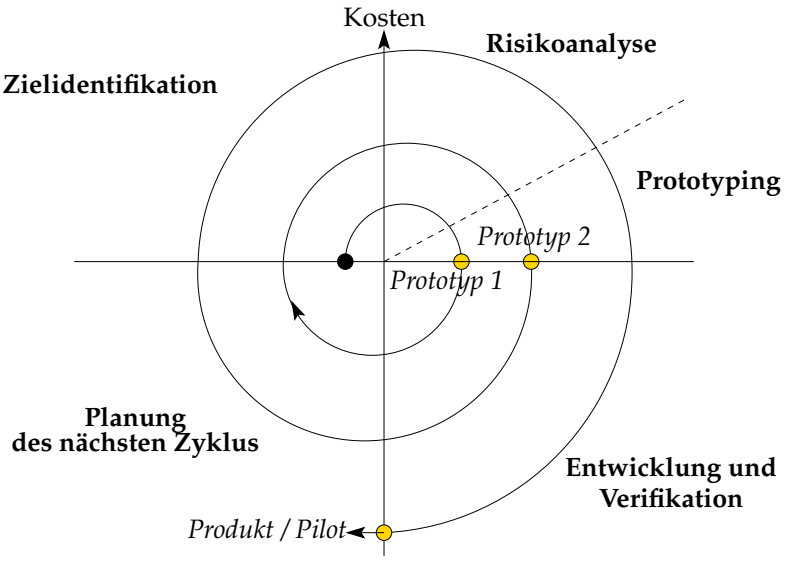

Abbildung 2.5: Das Spiralmodell

mehrfach durchgeführt, wobei aber pro Phase stets dieselben 4 (bzw. $4\frac{1}{2}$) Schritte durchgeführt werden (Abb. 2.5).

1. Schritt (Zielidentifikation)

- Identifikation der Ziele des nächsten Zyklus (Leistung, Funktionalität, Anpassbarkeit, ...)
- Erkennen alternativer Möglichkeiten zur Phasenrealisierung (Entwurf A, Entwurf B, Make, Buy, ...)
- zu beachtende Randbedingungen (Kosten, Zeit, Schnittstellen, ...)

2. Schritt (Risikoanalyse)

- Evaluierung der Alternativen unter Berücksichtigung der Ziele und Randbedingungen
- Entwicklung kosteneffizienter Strategien zur Risikoüberwindung, z.B. durch Prototypen, Simulationen, Benutzerbefragungen usw.

3. Schritt (Entwicklung und Verifikation)

- Entwurf, Implementierung und Tests

4. Schritt (Planung des nächsten Zyklus)

- Planung des nächsten Zyklus unter Einbeziehung der benötigten Ressourcen, ggf. Aufteilung des Produkts in Komponenten, die unabhängig voneinander weiterentwickelt werden.
- Review (Überprüfung) der Schritte 1 bis 3 des aktuellen Zyklus.
- Commitment (Einverständnis) über den nächsten Zyklus herstellen.

Das mehrfache Durchlaufen dieser Schritte wird als Spirale dargestellt. Die Fläche der Spirale vergrößert sich mit jedem Schritt und stellt die akkumulierten Kosten dar, die durch die bisherigen Aktivitäten angefallen sind.

Von seiner Absicht her ist das Spiralmodell somit ein risikogetriebenes Modell, oberstes Ziel ist die Risikominimierung. Zudem ist es ein offenes Vorgehensmodell, denn es gibt nicht notwendig ein von Beginn an festgelegtes Ende. Die Ziele für jeden Zyklus werden aus den Ergebnissen des vorherigen Zyklus abgeleitet, es gibt auch keine Trennung zwischen Wartung und Entwicklung.

2.2.2 Versionen-Entwicklung *(Versioning)*

Eng verwandt mit dem Spiralmodell ist der Ansatz des *Versioning*, der auch als *evolutionäres Modell* oder *Prototyping* bezeichnet wird [2, S. 120f]. Allerdings ist es nicht risikogetrieben wie das ursprüngliche Spiralmodell, sondern code-getrieben, d.h., Priorität haben jeweils lauffähige Teilprodukte.

Eine Variante des Versioning ist das *inkrementelle Modell*. Der Unterschied zum Versioning ist nach Balzert, dass hier schon zu Beginn die Anforderungen an das Produkt vollständig erfasst werden. Allerdings werden dann ähnlich wie beim Versioning Teilsysteme bzw. Ausbaustufen entworfen und implementiert.

2.2.3 Der Rational Unified Process (RUP)

Der *Rational Unified Process (RUP)* ist ein objektorientiertes inkrementelles Vorgehensmodell zur Softwareentwicklung und ein kommerzielles Produkt der Firma Rational Software, die seit 2002 Teil des IBM Konzerns ist. Der RUP benutzt die UML (*Unified Modeling Language*, vgl. §3) als Notationssprache. Der RUP sieht verschiedene Disziplinen *(disciplines)* vor, welche die zu Beginn dieses Kapitels aufgeführten Phasen der Softwareentwicklung etwas variieren und fachlich-inhaltlich strukturieren [12, §3.3]:

- Bei der *Geschäftsprozessmodellierung (business process modeling)* werden die für das System relevanten Geschäftsprozesse in so genannten „Anwendungsfällen“ gemäß UML dokumentiert (vgl. §3), um ein gemeinsames Verständnis des Anwendungsumfelds zwischen Entwicklern und Anwendern zu schaffen.

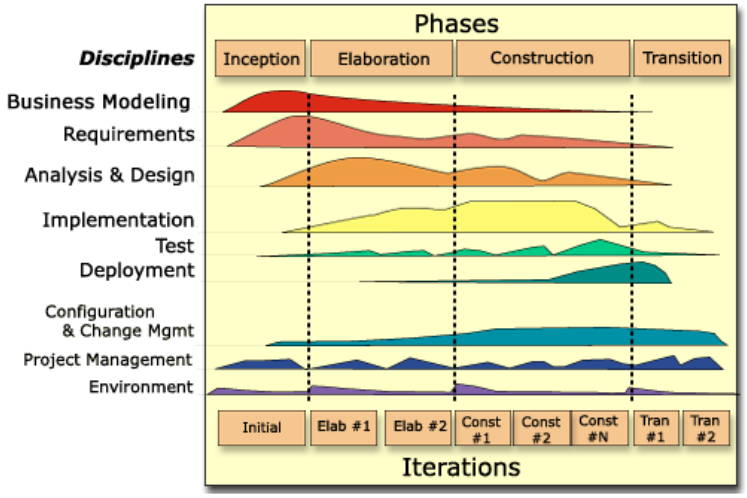

Abbildung 2.6: Der RUP (© Rational Software Corporation, jetzt IBM)

- In der *Anforderungsdefinition (requirements)* wird ermittelt, was das Anwendungssystem leisten soll.
- Gegenstand von *Softwarespezifikation* und *Entwurf (analysis and design)* sind die präzise Spezifikation und der Entwurf des Systems. Zentrales Ergebnis sind die Klassendiagramme nach UML, die das System modellieren.
- In der *Implementierung (implementation)* werden die einzelnen Klassen aus der Entwurfsdisziplin programmiert und zusammengeführt. Besonderes Augenmerk wird hierbei auf Aspekte der Wiederverwendung durch Schnittstellen, Module und Komponenten gelegt.
- Durch den *Test (test)* werden sowohl die einzelnen Module und Komponenten und ihr Zusammenspiel als auch die Erfüllung der für das System definierten Anforderungen überprüft.
- Die *Inbetriebnahme (deployment)* umfasst Installation des Systems, Schulung der Anwender, Beta-Tests usw.

Die Disziplinen des RUP durchläufen jeweils gleichzeitig die vier Phasen *Konzeption (inception)*, *Ausarbeitung (elaboration)*, *Konstruktion (construction)*, und *Übergang in den Betrieb (transition)*, wie in Abb. 2.6 illustriert. Je größer dabei die Fläche unter der Kurve innerhalb einer Phase ist, desto umfangreicher sind die in der entsprechenden Disziplin durchzuführenden Aktivitäten. Insbesondere am Ende der Ausarbeitungsphase liegt ein ausführbarer Prototyp der Entwurfspezifikation vor, nach der Konstruktionsphase die „Beta-Version“ des Systems.

Die Phasen des RUP sind jeweils wieder durch *Iterationen* unterteilt, und wie in einer Phase werden innerhalb einer Iteration praktisch alle Disziplinen durchlaufen mit dem Ziel, jeweils eine ausführbare Zwischenversion des Systems zu erhalten, bis schließlich das beim Anwender einzusetzende System vorliegt.

Trotz der prinzipiellen Unterschiede zum Wasserfallmodell zielt auch der RUP darauf ab, die Anforderungen nach einer angemessenen Zeit (am Ende der Ausarbeitungsphase) vorliegen zu haben. Auch der Entwurf sollte zu Beginn der Konstruktionsphase weitestgehend abgeschlossen sein. Auch wenn die starren Abgrenzungen der Phasen des Wasserfallmodells beim RUP nicht vorliegen, werden sich Anforderung und Entwurf beim RUP idealtypisch nicht in nennenswertem Umfang in die späteren Phasen hineinziehen.

2.2.4 Das Eclipse Process Framework (EPF)

The *Eclipse Process Framework (EPF)*[1] is an open source project that is managed by the Eclipse Foundation. It lies under the top-level Eclipse Technology Project. It has two goals:

- To provide an extensible framework and exemplary tools for software process engineering - method and process authoring, library management, configuring and publishing a process.
- To provide exemplary and extensible process content for a range of software development and management processes supporting iterative, agile, and incremental development, and applicable to a broad set of development platforms and applications. For instance, EPF provides the OpenUP/Basic, an agile software development process optimized for small projects.

Teil des EPF ist der *Open Unified Process (OpenUP)*, ein Open Source Software-Entwicklungsprozess, der von der Eclipse Foundation entwickelt wird. OpenUP orientiert sich an *Best Practices* der Softwareentwicklung und hält die grundlegenden Eigenschaften des Rational Unified Process (RUP) ein, also

- iterative Softwareentwicklung *(iterative and incremental development)*,
- Anwendungsfälle *(use cases)*,
- szenarienbasierte Entwicklung,
- Risikomanagement und
- architekturzentriertes Vorgehen

2.2.5 Open-Source: das Basarmodell

Das *Basarmodell (Bazaar model)* der Softwareentwicklung verzichtet bewusst auf klar unterscheidbaren Rollen von Anwendern und Entwicklern. Seinen Namen [7] bekam es in Abgrenzung zu den klassischen Vorgehensmodellen nach der „Kathedral"-Bauweise mit zentralisierter Planung und möglichst wenigen hochspezialisierten Programmierern. Das Modell zeichnet sich durch die folgenden Eigenschaften aus [7, 8]:

- *Anwender werden als Ko-Entwickler behandelt.* D.h. sie haben Zugriff auf den Quelltext und werden ermutigt, Erweiterungen der Software, Fehlermeldung *(bug report)*, Fehlerbehebungen *(bug fix)*, Dokumentation usw. vorzulegen. Je mehr Ko-Entwickler involviert sind, desto schneller werden die Entwicklungszyklen der Software. Das „Gesetz von Linus" besagt: *Genügend viele Tester finden jeden Softwarefehler, und zu jedem einmal erkannten Problem gibt es einen Entwickler, der es löst.*[2] Durch die breite Basis und die Kommunikation über das Internet können die beiden zentralen Aktionen eines Zyklus im Basarmodell, *Bug-Reporting* und *Bug-Fixing*, sehr schnell und zahlreich ablaufen.

> *Here, I think, is the core difference underlying the cathedral-builder and bazaar styles. In the cathedral-builder view of programming, bugs and development problems are tricky, insidious, deep phenomena. It takes months of scrutiny by a dedicated few to develop confidence that you've winkled them all out. Thus the long*

[1] *http://www.eclipse.org/epf/* [Letzter Abruf 17.3.2008]

[2] „Somebody finds the problem, and somebody *else* understands it. (...) Finding it is the bigger challenge." (Linus Torvalds [7])

release intervals, and the inevitable disappointment when long-awaited releases are not perfect.

In the bazaar view, on the other hand, you assume that bugs are generally shallow phenomena – or, at least, that they turn shallow pretty quick when exposed to a thousand eager co-developers pounding on every single new release. Accordingly you release often in order to get more corrections, and as a beneficial side effect you have less to lose if an occasional botch gets out the door.

And that's it. That's enough. If "Linus' Law" is false, then any system as complex as the Linux kernel, being hacked over by as many hands as the Linux kernel, should at some point have collapsed under the weight of unforseen bad interactions and undiscovered "deep" bugs. If it's true, on the other hand, it is sufficient to explain Linux's relative lack of bugginess.

(...) Linux was the first project to make a conscious and successful effort to use the entire world *as its talent pool. I don't think it's a coincidence that the gestation period of Linux coincided with the birth of the World Wide Web, and that Linux left its infancy during the same period in 1993-1994 that saw the takeoff of the ISP industry and the explosion of mainstream interest in the Internet. Linus was the first person who learned how to play by the new rules that pervasive Internet made possible.*

Eric S. Raymond [7]

Nach Raymond basiert die „kollaborative Innovation" von Open-Source auf demselben soziologischen Effekt, der auch der Delphi-Methode[3] unterliegt, nämlich dass der Durchschnitt aller Meinungen einer breiten Masse an Experten signifikant besser ist als die Meinung eines einzelnen zufällig aus dieser Masse gewählten Experten.

- *Frühe und schnelle Releases.* Die erste Version der Software sollte so früh wie möglich veröffentlicht werden, so dass sehr früh Ko-Entwickler gefunden werden.

- *Regelmäßige Integration.* Neuer Code sollte so oft wie möglich integriert werden, um einen Überhang zu vieler zu behebender Bugs am Ende des Entwicklungszyklus zu vermeiden. Einige Open-Source-Projekte haben *„nightly builds"*, deren Integration täglich automatisiert durchgeführt wird.

- *Mehrere Versionen.* Es sollte stets zwei Versionen der Software geben, eine Betaversion oder Entwicklerversion mit mehr Features und eine stabilere Version mit weniger Features. Die Entwicklerversion ist für Anwender bestimmt, die die neuesten Features sofort verwenden wollen und bereit sind die Risiken zu tragen, die der Gebrauch von noch nicht gründlich getestetem Code in sich birgt. Die Anwender können dann als Ko-Entwickler agieren, Bugs melden und Bugs fixen. Die stabile Version dagegen enthält weniger Bugs und weniger Funktionalitäten.

- *Hohe Modularisierung.* Die allgemeine Struktur der Software sollte modular sein, um parallele Entwicklung zu ermöglichen.

- *Dynamische Struktur zur Entscheidungsfindung.* Es muss eine Struktur zur Entscheidungsfindung vorhanden sein, formell oder informell, um strategische Entscheidungen aufgrund sich verändernder Anwenderanforderungen oder anderer Faktoren zu treffen.

[3] Die *Delphi-Methode*, auch *Delphi-Studie* oder *Delphi-Befragung*, ist ein systematisches, mehrstufiges Befragungsverfahren mit Rückkopplung, um zukünftige Ereignisse oder Trends schätzen zu können.

Historisch war Linux das erste Open-Source-Projekt in dieser Form. Es entstand 1991 durch Linus Torvalds aus einer Terminalemulation für die damaligen Unix-Systeme der Universitäten auf Basis des Minix-Systems von Andrew Tanenbaum an der Universität Amsterdam und des freien GNU-C-Compilers. Die erste Version 0.01 von Linux wurde am 17.9.1991 auf einem öffentlichen FTP-Server bereitgestellt. Torvalds gab Linux zunächst unter einer eigenen, proprietären Lizenz heraus, erst die Mitte Dezember 1992 veröffentlichte Version 0.99 war unter der GNU GPL. Durch diesen Schritt erst wurde Linux ein freies Betriebssystem, und erst jetzt zog es weltweit viele Programmierer an, um sich an der Entwicklung von Linux und GNU zu beteiligen. Durch das damals entstehende Internet konnte so der Entwicklungszyklus rapide beschleunigt werden (es gab oft mehrere Releases von Linux *an einem Tag*!). Je schneller die Release-Wechsel erfolgten, desto effizienter wurde die Fehlerbehebung, da die Wahrscheinlichkeit, dass mehrere Entwickler an demselben Fehler arbeiteten und sich damit womöglich behinderten, geringer wurde. Ebenso bezeichnete Torvalds bestimmte Versionen des Linuxkerns als „stabil“ und stellte sie auch zukünftig bereit, so dass die Anwender zwischen der aktuell in Entwickelung befindlichen Betaversion mit neuen, aber womöglich fehlerhaften Funktionen oder einer alten stabilen Version wählen konnten.

> *Linux was the first project to make a conscious and successful effort to use the entire* world *as its talent pool. I don't think it's a coincidence that the gestation period of Linux coincided with the birth of the World Wide Web, and that Linux left its infancy during the same period in 1993–1994 that saw the takeoff of the ISP industry and the explosion of mainstream interest in the Internet. Linus was the first person who learned how to play by the new rules that pervasive Internet made possible.*
>
> Eric S. Raymond [7]

Nach Richard Stallman, dem Gründer der GNU Public License und der Bewegung für Freie Software, ging mit der Entstehung von Open-Source ein Paradigmenwechsel einher, indem die *Freiheit* von Software als Grundsatz ausgetauscht wurde durch die *Effizienz der Entwicklung* von Software [11].

2.3 Psychologie des Software-Engineering

Neben standardisierten Methoden oder Daumenregeln des Software-Engineering, die die Effizienz der Softwareentwicklung steigern, gibt es einen weiteren Faktor: die Software-Analytiker, Entwickler und Manager als Menschen, mit all ihren Stärken und Schwächen. Nach einer bekannten Anekdote [5, S. 428ff] befand sich an einem Ende eines großen Rechnerraum für Studenten in einer Universität ein Getränkeautomat. Nachdem es von einigen Studenten Beschwerden gab, dass von dieser Ecke des Raumes zu großer Lärm kam, wurde der Automat entfernt. Daraufhin stieg jedoch der Beratungsaufwand der beiden angestellten Tutoren so dramatisch, dass sie die Anfragen der Studenten nicht mehr bearbeiten konnten. Was war passiert? Der Lärm der Studenten an dem Getränkeautomaten rührte daher, dass sie über ihre Computerprobleme miteinander sprachen und sie durch diese Kommunikation an Ort und Stelle lösen konnten. Nur außergewöhnlich schwere Probleme wurden so an die Tutoren herangetragen. Durch die Wegnahme des zwar unkonventionellen, aber auf diese Weise hocheffizienten Beratungsservice des Getränkeautomaten sank die geistige Produktivität, obwohl das Gegenteil beabsichtigt war.

2.3.1 Agile Softwareentwicklung

Das Ziel *agiler Softwareentwicklung* ist es, den Softwareentwicklungsprozess flexibler und schlanker zu machen als nach den klassischen Vorgehensmodellen. Es soll mehr auf die zu

erreichenden Ziele geachtet und auf technische und soziale Probleme bei der Softwareentwicklung eingegangen werden. Die agile Softwareentwicklung ist eine Gegenbewegung zu den oft als schwergewichtig und bürokratisch angesehenen Softwareentwicklungsprozessen wie dem Rational Unified Process.

Ein ganz neuartiges Prinzip der agilen Softwareentwicklung ist die *Paarprogrammierung (pair programming)*. Hierbei wird der Quelltext von jeweils zwei Programmierern an einem Rechner erstellt. Ein Programmierer, der „Driver“, schreibt dabei den Code, während der andere, der „Navigator“, über die Problemstellungen nachdenkt, den geschriebenen Code kontrolliert und Probleme, die ihm dabei auffallen, sofort anspricht, z.B. Schreibfehler oder logische Fehler. Diese können dann sofort im Gespräch zu zweit gelöst werden. Die beiden Programmierer sollten sich bezüglich dieser beiden Rollen periodisch abwechseln. Auch die Zusammensetzung der Paare sollte sich regelmäßig ändern. Paarprogrammierung führt zu qualitativ signifikant höherwertigen Programmen als die Einzelprogrammierung [5, S. 428].

Ferner verfolgen traditionelle Entwicklungsmethoden eine „Fließbandstrategie“ und betrachten die Entwickler (Analysten, Programmierer, Tester) als austauschbar innerhalb ihrer jeweiligen Gruppe. Die agile Softgwareentwicklung dagegen konzentriert sich auf das individuelle Können und setzt keine Grenzen, jeder Entwickler soll Wissen und Erfahrungen durch die Arbeit mit anderen gewinnen. So durchläuft dasselbe Team alle Phasen eines Zyklus von der Analyse bis zur Funktionsprüfung.

2.3.2 eXtreme Programmierung (XP)

Eine der verbreitetsten agilen Entwicklungsmodelle ist die *eXtreme Programmierung (XP)*. Es eignet sich für kleinere Projekte mit bis zu 15 Entwicklern in einem Umfeld mit sich rasch ändernden Anforderungen. Oberstes Prinzip der eXtremen Programmierung ist Einfachheit, und sie heißt „extrem“, da sie alle aus Erfahrung gute Methoden, in der Terminologie von XP *Praktiken* genannt, bis zum äußersten treibt. Beispielsweise befürwortet sie häufige und automatische Tests. Auch die evolutionäre Entwicklung wird extrem ausgeführt. Erstens findet die Systemintegration häufig statt (mindestens einmal am Tag), so dass stets ein voll arbeitsfähiges System verfügbar ist, auch wenn es in den Zwischenversionen nicht alle Anforderungen erfüllt; dadurch kann man jedoch das System zu jeder Zeit aktuell testen. Zweitens basiert die Entwicklung auf kurzen Zyklen (etwa drei Wochen) nach Art des Spiralmodells (Abb. 2.7). Drittens gibt es alle paar Monate Kundenversionen *(Releases)*, was bedeutet, dass der Kunde die ganze Zeit in den Prozess eingebunden sein muss (und nicht nur in der Anfangsphase wie im Wasserfallmodell); ein vor Ort verfügbarer Kundenvertreter gehört somit zu den grundlegeneden Forderungen der extremen Programmierung. Viertens wird das agile Prinzip der Paarprogrammierung verlangt.

Neben der Einfachheit sind effektive Kommunikation und sofortiges Feedback Grundprinzipien des XP. Um effektive Kommunikation zu ermöglichen, sitzen alle Entwickler in einem Raum, und repräsentative Anwender sowie Fachexperten sind eng in die Entwicklung eingebunden; statt mit formalen Dokumenten erfolgt die Kommunikation möglichst in direkten Gesprächen und durch den Quelltext (Inline-Dokumentation). Das sofortige Feedback wird erreicht durch konsequente Paarprogrammierung auf der Codierungsebene und durch die immer wieder durchzuführenden Tests auf der funktionalen Ebene. Das Vorgehensmodell des XP ist evolutionär und umfasst in jedem Iterationsschritt das *Refactoring*, worunter man alle Maßnahmen versteht, durch die die Qualität der Software und insbesondere des Entwurfs verbessert wird, ohne dass die Funktionalitäten geändert werden. Refactorings sollen die vorliegende Software soweit verbessern, dass sie lesbar, verständlich, wartbar und erweiterbar wird. Es ist dazu extra ein spezieller Arbeitsschritt vorgesehen, der also nicht direkt zur Erweiterung der Funktionalität beiträgt und somit, scheinbar, unproduktiv ist.

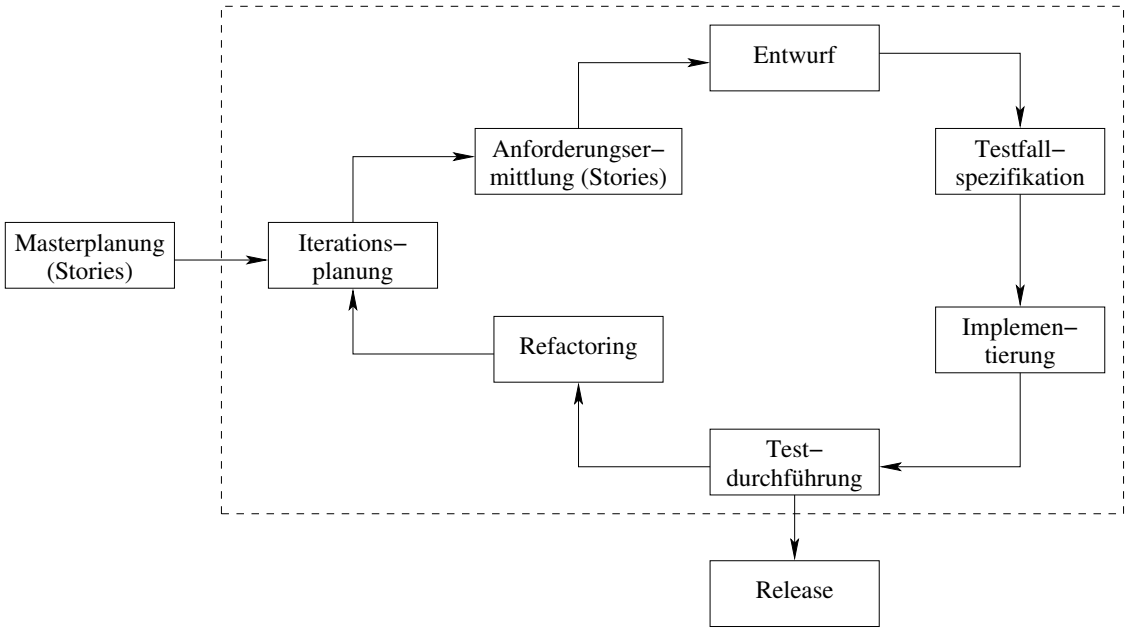

Abbildung 2.7: Das Vorgehensmodell von XP

Ein grundlegendes Element bei XP bilden die *Stories*, Szenarien, die der Beschreibung der Benutzeranforderungen dienen und vergleichbar mit den Anwendungsfällen der UML sind. Ziel der Stories ist allerdings weniger die Erstellung einer vollständigen Spezfikation, sondern vielmehr die Beschreibung eines Kundenwunsches, der mit einer Priorität versehen als Einheit für die Masterplanung verwendet wird. Im Laufe des Entwicklungsprozesses können neue Stories hinzukommen oder bestehende verfeinert werden.

Wichtig ist auch die Testfallspezifikation *vor* der Implementierung, so kann man schon während der Codierung automatisiert erste Funktionstests durchführen.

Die eXtreme Programmierung setzt einige soziale Voraussetzungen an die Entwickler und Kunden.

- Alle Entwickler sollten über eine gute und breite Qualifikation und hohe soziale Kompetenz verfügen, da sie laufend in anderen Paaren und für andere Aktivitäten eingesetzt werden. Paare aus einem erfahrenen Mitarbeiter und einem Anfänger sollten nur zu Schulungs- und Einarbeitungszwecken gebildet werden.

- Alle am Projekt Beteiligten (Management, Teammitglieder, Auftraggeber und Anwender) müssen die XP-Praktiken akzeptieren. Insbesondere muss also der Auftraggeber bereit und in der Lage sein, einen qualifizierten Mitarbeiter als Anwendervertreter für das Projektteam abzustellen. Aufgrund der hohen Dynamik und den sich oft erst im Projekt ergebenden Systemanforderungen ist es problematisch, für das Projekt vertragliche Vereinbarungen zu treffen, die allen Beteiligten Planungssicherheit geben.

- XP erfordert hohe intensive Kommunikation und funktioniert am besten, wenn die Entwickler an einem Ort konzentriert sind und die gleichen Arbeitszeiten haben. Zudem stellt die arbeitsteilige Paarprogrammierung hohe Anforderungen an die Konzentration und Leistungsfähigkeit der Entwickler, so dass unbedingt auf *geregelte Arbeitszeiten* zu achten ist. Überstunden zeugen von schlechtem Projektmanagement.

2.4 Flexibles Rahmenmodell: V-Modell XT

Das *V-Modell XT* ist ein sehr umfangreiches Vorgehensmodell, das (in seiner aktuellen Version 1.3 von 2009) das ursprüngliche V-Modell um iterativ-inkremnetelle Vorgehensweisen und agile Elemente (s.u.) wesentlich erweitert. Es ist in 1990er Jahren ursprünglich für die Bundeswehr entwicjkelt worden, wurde später für sämtliche Bundesbehörden verbindlich und findet mittlerweile auch in der Industrie weite Verbreitung.

Wesentliche Elemente des V-Modells XT sind die *Vorgehensbausteine*, die zusammengehörige Rollen, Produkte und Aktivitäten kapseln und aus denen das konkrete Vorgehensmodell eines Projekts zusammengestellt wird („Tailoring“).[4] Ein Vorgehensbaustein deckt eine konkrete Aufgabenstellung ab, die im Rahmen eines V-Modell-Projektes auftreten kann. Festgelegt werden dabei die innerhalb dieser Aufgabenstellung zu erarbeitenden Produkte, die Aktivitäten, durch welche die einzelnen Produkte erstellt werden, sowie die an den einzelnen Produkten mitwirkenden Rollen. Die einzelnen Vorgehensbausteine sind dabei jeweils in sich abgeschlossen. Ein *Produkt* bezeichnet ein allgemeines Arbeitsergebnis oder Erzeugnis und kann beispielsweise das zu erstellende System, ein Dokument, Prüfprotokoll oder ein Software-Modul sein. Eine Aktivitäten bearbeitet oder verändert Produkte. Eine Rolle ist die Beschreibung einer Menge von Aufgaben und Verantwortlichkeiten im Rahmen eines Projekts oder einer Organisation. Insgesamt müssen die in Abbildung 2.8 aufgeführten Beziehungen eingehalten werden: Im Mittelpunkt der Vorgehensbausteine stehen die

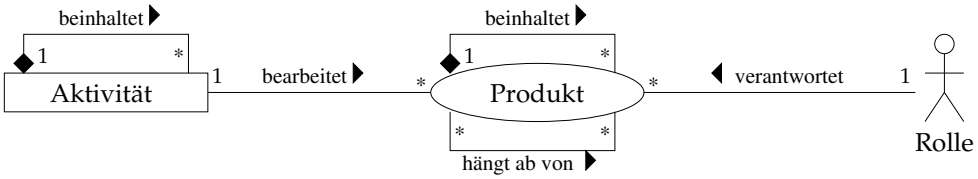

Abbildung 2.8: Ein Vorgehensbaustein im V-Modell XT bildet sich aus Aktivitäten, Produkten und Rollen und ihren Beziehungen untereinander. Grafik nach [6, §2.4]

Arbeitsergebnisse, die Produkte. Ein Produkt kann aus mehreren untergeordneten Produkten bestehen, außerdem können Abhängigkeiten zwischen Produkten existieren. Eine Rolle ist verantwortlich für Produkte, und Produkte werden durch Aktivitäten bearbeitet. Aktivitäten können wieder in weitere Aktivitäten unterteilt werden. Vorgehensbausteine können eigenständig verwendet werden, wobei gegebenenfalls Abhängigkeiten zu anderen Vorgehensbausteinen zu berücksichtigen sind.

Vorgehensbausteine legen keine konkrete Reihenfolge im Projekt fest. Dies geschieht durch die *Projektdurchführungsstrategien*, Abbildung 2.9. Sie legen die Reihenfolge von zu

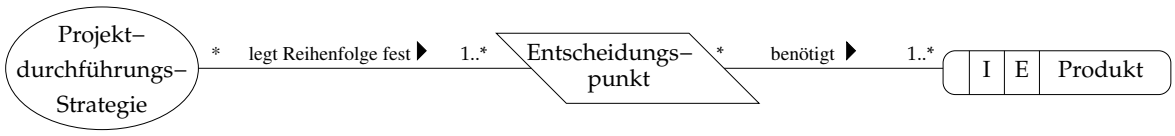

Abbildung 2.9: Durchführungsstrategien im V-Modell XT. Grafik nach [6, §2.4]

erreichenden Projektstufen fest. Das Ende jeder Projektstufe ist durch einen *Entscheidungspunkt*, oder *Meilenstein*, versehen, für den wiederum Kriterien erfüllt sein müssen, zum Beispiel zu erstellende oder zu überprüfende Produkte. Im V-Modell XT sind verschiedene Projektdurchführungsstrategien verfügbar, so unter Anderem die inkrementelle Systementwicklung, die komponentenbasierte Systementwicklung und die agile Systementwicklung.

[4] *to tailor*: schneidern. In der Tat steht XT für *„extreme Tailoring“* und soll die Anpassbarkeit des Modells an die jeweiligen Bedürfnisse ausdrücken.

Bei der Einführung und Umsetzung des V-Modells XT geht man üblicherweise zweistufig vor. Zunächst wird das V-Modell XT auf die unternehmensspezifischen Bedürfnisse angepasst. Durc dieses Tailoring auf Unternehmensebene entsteht so ein unternehmensspezifisches Vorgehensmodell.

Kapitel 3

Analyse und Entwurf mit UML

Inhalte von [10] als e-Book

http://www.springerlink.com/content/jm3124/

unter *springerlink.com*. Die Kapitel des e-Books sind nur über den FH-Zugang (d.h. Poolräume oder VPN) kostenfrei als Lehrmaterial zu dieser Veranstaltung herunterladbar.

Kapitel 4

Programmierleitsätze und *Best Practices*

4.1 Programmierleitsätze nach der Unix-Philosophie

[1]Die kontinuierlich steigende Popularität der Unix-Betriebssysteme seit ihren Anfängen und ihre resultierende heutige Mächtigkeit, denen sie ihren ersten Rang als Plattform für Serveranwendungen im Internet verdanken, beruht auf der konsequenten Anwendung einiger weniger fundamentaler Prinzipien.

Eine der grundlegenden Annahmen der Unix-Entwicklung war stets, dass der Benutzer mit dem Computer umgehen kann, dass er weiß was er tut. Als Folge daraus sind, anders als in anderen Umgebungen, Unix-Programme nicht darauf ausgerichtet, den Benutzer daran zu hindern etwas zu tun, sondern ihm zu ermöglichen ihre volle Leistungsfähigkeit auszuschöpfen. Die Grundsätze der Unix-Philosophie sind geprägt von konsequenter Einfachheit, Motto

KISS = *„Keep it simple and stupid“*.

Im folgenden werden sie in einer Reihe von einprägsamen Leitsätzen zusammengefasst und diese jeweils erläutert. Die sequentielle Darstellung soll nicht täuschen, diese Prinzipien greifen alle ineinander und machen in ihrer Kombination die Mächtigkeit des Systems aus.

4.1.1 Klein ist schön

„Perfektion scheint erreicht, nicht wenn sich nichts mehr hinzufügen lässt, sondern wenn man nichts mehr wegnehmen kann.“[2] Zögere nicht, überholte Funktionalitäten wegzuwerfen, wenn es ohne sie auch ohne Verlust an Effektivität geht. Wenn dein Quelltext sowohl besser als auch einfacher wird, dann *weißt* du, dass du es richtig machst.

Kleine Programme sind leicht zu verstehen. Durch ihre begrenzte Quelltextlänge und Funktionalität bleiben kleine Programme überschaubar, was wiederum zu weniger Fehlern führt. Natürlich kann ein Programm gleich welcher Größe aufgrund seiner besonderen Funktionalität schwer zu verstehen sein, aber dies ist die Ausnahme. Es gibt keine allgemeingültige Regel, wann aus einem kleinen Programm ein großes wird. Warnzeichen sind z.B. wenn der Entwickler bei einer Fehlermeldung selbst nicht mehr weiß, wo sie herkommt, oder wenn man den Quelltext ausdrucken muss, um den Überblick zu wahren. Kleine Programme sind leicht zu warten. Da ein kleines Programm im allgemeinen leicht zu verstehen ist, ist es auch leicht zu warten (korrigieren, erweitern, portieren).

[1]Dieser Abschnitt ist eine Zusammenfassung des Buchs Mike Gancarz: *The UNIX Philosophy*. Digital Press, 1995, nach Christian Weisgerber: *Die Unix-Philosophie*, 25. Juni 1998 (*http://sites.inka.de/mips/unix/unixphil.html*)

[2]Antoine de Saint-Exupéry, in: *Terre des Hommes, III: L'Avion*, p. 60 (1939). Original. franz.: *„Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.“*

Kleine Programme verschlingen weniger Systemressourcen. Sei es Plattenplatz oder Speicher, kleine Programme sind schonender im Umgang mit begrenzten Betriebsmitteln. Man mag einwenden, dass die Summe mehrerer kleiner Programme mehr Ressourcen benötigt als ein großes mit der gleichen Funktionalität, aber:

Kleine Programme lassen sich leichter mit anderen Werkzeugen kombinieren. Wie weiter unten noch angesprochen wird, liegt die wahre Mächtigkeit des Unix-Konzepts in der Kombinierbarkeit vieler kleiner Werkzeuge. Kein Autor kann alle Anwendungen, die seine Programme erfahren werden, voraussehen. Große, monolithische Anwendungen sind wenig flexibel. Eine Sammlung kleiner, handlicher Programme ist neuen Anforderungen viel eher gewachsen.

4.1.2 Jedes Programm soll genau eine Sache gut machen

Kleine Programme haben meistens nur eine Funktion. Programme mit nur einer Funktion neigen wiederum dazu klein zu sein, was sich mit dem ersten Leitsatz ergänzt. Ein viel kritisiertes Problem ist der sogenannte „creeping featurism“, der schleichende Einzug von immer mehr Funktionalität, die eigentlich überflüssig ist oder ausgelagert gehört, und die kleine Programme zu großen aufbläht.

Einige Fragen, die man sich stellen kann, um das Wuchern des schöpferischen Drangs in sinnvolle Bahnen zu lenken: Muss ein Programm interaktiv sein oder reicht es nicht, wenn der Benutzer die Eingaben in eine Datei oder über Kommandozeilenparameter macht? Falls Ein-/Ausgabedaten besonders formatiert werden sollen, gibt es dann nicht schon andere Werkzeuge, die das übernehmen können? Gibt es überhaupt nicht bereits ein ähnliches Programm, das man einfach verwenden kann?

4.1.3 Erstelle so schnell wie möglich einen Prototyp

Dieser Leitsatz steht im Widerspruch zur klassischen Lehre, dass der Entwurf eines Programms schon abgeschlossen sein soll, bevor man dann im letzten Schritt die Implementierung vollzieht. „90% des Programmentwurfs sollten vor dem ersten Compilerlauf vollzogen sein“, so oder ähnlich heißt es. Das klingt vernünftig in der Theorie, beruht aber auf der unrealistischen Annahme, dass der Entwurf von vornherein schon komplett überschaubar ist.

Die Erstellung eines Prototypen ist ein lehrreicher Vorgang. Man kann frühzeitig erkennen, ob das Projekt in dieser Form überhaupt realisierbar ist, und wie die eigenen Vorstellungen sich in der Praxis bewähren. Es ist besser eine fehlerhafte Annahme frühzeitig zu erkennen als kurz vor Fertigstellungsfrist.

Die frühe Anfertigung eines Prototypen verringert Risiken. Die Erfahrung zeigt, dass Programmspezifikationen in der Zeit, die ein Projekt zur Fertigstellung braucht, nicht stabil sind, und dass Anwender und Auftraggeber oft Spezifikationen abgeben, die vom Programmierer anders verstanden werden oder schlicht falsch sind. Ein früher Prototyp hilft, solche Fehler zu erkennen, bevor Zeit und Arbeit vergeudet sind.

4.1.4 Portabilität geht über Effizienz

Der mächtigste Computer ist nicht der mit der schnellsten CPU und dem meisten Speicher, sondern der, der am meisten benutzt wird. Ein Laptop, den man den ganzen Tag bei sich hat, kann eine viel größere Produktivität haben, als der vielfach leistungsfähiger Tischrechner, der unbenutzt im Büro steht.

Aber auch in der umgekehrten Richtung ist Portabilität Trumpf. Es hat keinen Sinn, zuviel Zeit damit zu vergeuden, ein Programm durch systemspezifische Optimierungen

schneller zu machen. Nächstes Jahr, bei den heutigen Produktzyklen vielleicht schon nächstes Quartal, wird die Hardware schneller sein. Die effizienteste Programmierung ist selten die portabelste. Niemand wird sich mehr dafür interessieren, dass eine Applikation früher auf einer bestimmten Plattform viel schneller als die anderen war, wenn man sie auf der neuen Plattform, wo alle schneller sind, erst gar nicht einsetzen kann. Portable Software verringert auch den Bedarf für Benutzerschulung.

Gute Programme sterben nie, sie werden auf neue Plattformen portiert.

4.1.5 Speichere numerische Daten in ASCII-Dateien

Genauso wichtig wie die Portabilität von Programmen ist die Portabilität der Daten. Ein wesentlicher Beitrag dazu ist, numerische Daten nicht in einem Binärformat zu speichern, sondern in ASCII-Text.

ASCII-Text, mit allen seinen Schwächen, ist das verbreitetste Austauschformat. Er unterliegt keinen Problemen mit der Byte-Reihenfolge oder verschiedenen Fließkommaformaten, die den Austausch von Binärdaten immer wieder komplizieren. Er ist leicht zu lesen und mit normalen Editoren und unter Einsatz der Unix-Text-Werkzeuge wie cut, diff, grep, sed, sort, wc usw. leicht zu bearbeiten. Bessere Portabilität überwiegt gegenüber dem Geschwindigkeitsverlust durch die Wandlung bei der Ein-/Ausgabe, der in der Praxis im Vergleich zur eigentlichen Verarbeitungszeit ohnehin meist vernachlässigbar ist.

Heute, zu Beginn der 2010er Jahre, hat sich Unicode als neuer Standard für Textdaten etabliert, und so werden ganz im Sinne der Unix-Philosophie strukturierte Daten in XML und Unicode gespeichert.

4.1.6 Nutze die Hebelwirkung von Software zu deinem Vorteil

Durch einen Hebel kann man mit geringer Kraft große Lasten bewegen. In diesem Sinne gilt es, die Hebelwirkung *(„leverage")* von Software zu erkennen und zu verwenden.

Gute Programmierer schreiben guten Code, wirklich große Programmierer borgen guten Code. Es bringt keinen Fortschritt, das Rad immer wieder neu zu erfinden. Es gilt den eigenen Stolz zu bezwingen und das sogenannte NIH-Syndrom zu vermeiden (engl. „Not Invented Here"), und auch Dinge, die nicht selbst oder nicht im eigenen Haus erfunden worden sind, einzusetzen. Umgekehrt ist es wichtig, anderen Leuten zu erlauben, seinen eigenen Code zu verwenden, damit diese mit ihrer Arbeit darauf aufbauen können.

Automatisiere alles. Ein mächtiges Mittel, die Hebelwirkung der Software zum eigenen Vorteil einzusetzen, ist es, die Arbeit auf die Maschine zu übertragen. Es ist Zeitverschwendung etwas manuell zu machen, wenn der Computer es auch tun kann. Immer wiederkehrende, monotone Arbeiten sind typische Kandidaten für eine Automatisierung. Genau hier schlägt die Batchfähigkeit von Programmen zu und das Konzept der vielen kleinen unabhängigen Werkzeuge und der Filter, das bei anderen Leitsätzen angesprochen wird.

4.1.7 Vermeide Benutzeroberflächen, die den Benutzer gefangen halten

Eine „CUI" *(Captive User Interface)* ist eine Applikation, die eine Interaktion mit dem Benutzer außerhalb des höchsten Befehlsinterpreters erzwingt. Ist die Anwendung einmal gestartet, gehen alle Eingaben des Benutzers an sie und nicht an den Befehlsinterpreter, bis das Programm wieder beendet wird.

CUIs neigen zu „groß ist schön" im Widerspruch zu Leitsatz 1. Da der Benutzer in der Oberfläche gefangen ist, hat er keinen direkten Zugriff auf andere Systemfunktionen, weswegen zunehmend Druck entsteht, diese innerhalb der Applikation zu duplizieren.

Programme mit CUI sind schwer mit anderen Programmen zu kombinieren, weil sie auf der Annahme basieren, der Benutzer sei ein Mensch. Die Stärke von Unix liegt aber darin, wie Programme miteinander arbeiten. Programme mit CUI lassen sich gar nicht oder nur sehr schwer in Skripte einbinden, Abläufe nicht automatisieren, sie ziehen keinen Vorteil aus der Software-Hebelwirkung. Da sie nicht mit anderen Werkzeugen kombiniert werden können, müssen deren Funktionen mit eingebaut werden. In einem Teufelskreis gefangen wächst das Programm zu einem immer größeren und schwerfälligeren Monolith.

CUIs skalieren nicht. Ein Programm, das z.B. der Reihe nach alle notwendigen Angaben abfragt, die notwendig sind, um einen neuen Benutzerzugang auf einem System anzulegen, oder ein entsprechendes Menü anbietet, mag auf den ersten Blick benutzerfreundlich aussehen, aber wenn damit tausend Benutzer einzutragen sind, dann wird das nichts.

4.2 Programmierleitsätze nach OpenSource

In seinem einflussreichen und sehr lesenswerten Artikel [7] untersucht Eric S. Raymond anhand des Beispiels von Linux und seiner eigenen OpenSource-Entwicklung des Mail-Clients `fetchmail` die Programmiergrundsätze, die eine Entwicklung nach einem „Basar"-Modell von einer traditionell organisierten „Kathedral"-Bauweise von Software mit zentralisierter Planung und wenigen hochspezialisierten Programmierern unterscheidet und die wie im Falle von Linux für den überraschenden Erfolg verantwortlich sind. Er nennt u.a. die folgenden:

1. *Every good work of software starts by scratching a developer's personal itch.* Notwendigkeit ist die Mutter der Erfindung. Schreibe nur Programme, die deine Probleme lösen, aber keine, die dich nicht interessieren oder die du nicht brauchst.

2. *Plan to throw one away; you will, anyhow.* Um ein Programm richtig zu schreiben, sei bereit, es mindestens einmal wegzuwerfen. Oft beginnt man erst während oder nach der ersten Implementierung, das Problem und seine Lösung zu verstehen.

3. *When you lose interest in a program, your last duty to it is to hand it off to a competent successor.* Entsprechend dem ersten Leitsatz sollte derjenige ein Projekt betreiben, der das größte Interesse daran hat.

4. *Smart data structures and dumb code works a lot better than the other way around.* Oft wird der Algorithmus („Code") eines Programms als wesentlich angesehen und die Datenstrukturen als bloßes Hilfsmittel dazu. Um ein Programm schnell zu verstehen, sind saubere und dem Problem gut angepasste Datenstruktutren wesentlich, die Algorithmen ergeben sich dann meist fast von selbst. In objektorientierten Programmiersprachen werden die Datenstrukturen als Klassen programmiert und unterstützen damit diese Sichtweise.

5. *Often, the most striking and innovative solutions come from realizing that your concept of the problem was wrong.*

4.3 Modularität und Schnittstellen

Computerprogramme sind eher für Menschen geschrieben als für Computer. Aha? Computerprogramme sind doch ganz offensichtlich erstellt, um auf Computern ausgeführt zu werden!? Wenn das so wäre, würde man heute nur noch Assemblerprogramme schreiben (Abb. 4.1). Oder direkt Maschinencode. Stattdessen schreibt man Programme, insbesondere solche mit

```
*0200                   / Code starts at Address 0200
Main,   cla cll         / Clear AC and Link
        dca Q           / Zero Quotient
        dca R           / Zero Remainder
        tad B           / Load B
        sna             / skip if B not zero
        jmp Error       / halt if zero divisor
        cia             / Negate it
        dca MB          / Store at -B
        tad A           / Load A
Loop,   tad MB          / Add -B
        spa             / Skip on Positive Accumulator
        jmp Done        /   otherwise Done!
        isz Q           / Increment Quotient
        jmp Loop        /    and go again
Done,   tad B           / Restore Remainder
        dca R           /    and Save
Error,  hlt             / Halt (Halt on Error)
        jmp Main        / Go Again
```

Abbildung 4.1: Der Euklidische Algorithmus in PDP-8 Assembler, implementiert von Brian Shelburn (*http://www4.wittenberg.edu/academics/ mathcomp/bjsdir/PDP8Labs.zip*)

etwas komplizierteren Algorithmen, fast ausschließlich in Hochsprachen, die zumeist an das Englische angelehnt sind (Abb. 4.2). Dem Computer ist es egal, wie ein Programm ge-

```
public static long euclid(long a, long b) {
   long tmp;
   while (n > 0) {
      tmp = m;
      m = n;
      n = tmp % m;
   }
   return m;
}
```

Abbildung 4.2: Der Euklidische Algorithmus in Java

schrieben ist. Doch Programmierer müssen sie lesen und verstehen, um sie zu schreiben oder zu modifizieren. Fast jedes eingesetzte Programm wird während seiner Lebenszeit modifiziert, Modifikation von Software ist nicht die Ausnahme, sondern die Regel. Einer der wichtigsten Programmiergrundsätze lautet damit:

Programme müssen möglichst leicht lesbar, verstehbar und modifizierbar sein.

Ein Mittel dazu ist das Prinzip der *Modularität*. Computerprogramme sollen in kleine und möglichst unabhängige Teile zerlegt werden. Jedes Modul hat seine spezifischen Aufgaben. Dadurch werden die drei obigen Ziele simultan erreicht. Denken Sie an ein gutes Benutzhandbuch eines technischen Produkts. Es ist in separate Themen unterteilt, die sich in ihren Funktionalitäten unterscheiden. Bei dem Benutzhandbuch für ein Auto werden die Instrumente und Kontrollanzeigen erklärt, in anderen Abschnitten wird angegeben, wie man Reifen wechselt oder wie man die Scheinwerfer, die Blinker und das Standlicht bedient. Bei Modifikationen des Autos kann der Hersteller das Handbuch weiterverwenden, es müssen nur die entsprechenden Abschnitte geändert werden. Und dennoch gibt es, gerade in einem guten Handbuch, wichtige Informationen, die dort *nicht* zu finden sind, beipielsweise die Vorschrift, dass Sie den linken Blinker setzen müssen, bevor Sie links abbiegen, oder in welchen Ländern man auch tagsüber mit eingeschaltetem Scheinwerfer fahren muss.

Natürlich können Module nicht hermetisch abgeriegelt und logisch unabhängig vom Hauptprogramm oder auch von anderen Modulen funktionieren.

Modularität bedeutet auch, dass minimale Informationen modulübergreifend geteilt werden.

Die Strukuren der geteilten Informationen (die genormten „Stecker“ und „Buchsen“) werden durch *Schnittstellen (interfaces)* bereitgestellt. Sie enthält alle Details darüber, *welche* Informationen („Dienste“) verlangt werden, aber nicht, *wie* die Informationen geschaffen werden. In objektorientierten Programmiersprachen wie Java ist das Schnittstellenkonzept umgesetzt.

4.3.1 Plug-ins

Plug-ins, oft auch *Erweiterungen*, sind Beispiele spezieller Module, die für bestimmte Programme geschrieben (z.B. ein Browser) werden und und deren Funktionalität (z.B. Plug-In zum Lesen von PDF-Dokumenten) erweitern. Dazu verwenden sie die von dem Hauptprogramm angebotenenen Schnittstellen.

4.4 Code-Konventionen

http://java.sun.com/docs/codeconv/

Kapitel 5

Kollaborative Softwareentwicklung

Die Softwareentwicklung in Teams wird auch *kollaborative Softwareentwicklung* genannt. Diese umfasst insbesondere die Zusammenarbeit mehrerer Entwickler über ein Netzwerk oder über das Internet.

5.1 Grundsätzliche Problematiken

Eine grundsätzliche Problematik kollaborativer Entwicklung ist, dass mehrere Entwickler *gleichzeitig* eine Datei ändern. Beim Abspeichern in einem solchen Falle kommt es zu Konflikten, die in der Regel durch eine mit den beteiligten Entwicklern abgesprochene gemeinsame Versionierung der betroffenen Dateien gelöst wird.

Ein weiteres prinzipielles Problem in der Softwareentwicklung (nicht nur der kollaborativen) ist, dass man relativ leicht auf eine frühere stabile Version der Software zurückkommen muss, falls durch unbeabsichtigte Fehler oder nicht absehbare Seiteneffekte der aktuellen Entwicklerversion die Software nicht mehr funktioniert.

Ein automatisches Versionierungssystem, welches kollaborative Entwicklung ermöglicht, muss also den einen aktuellen Versionsstand als auch die vergangenen Versionen („Historie“) aller Dateien speichern und bei Abspeicherungen etwaige Konflikte und deren beteiligte Entwickler anzeigen. Um Entwickler an verschiedenen Rechnern und Orten zu beteiligen, muss das System über ein Netzwerk verfügbar sein. Der natürliche Aufbau eines solchen Systems ist also entweder eine Client-Server-Architektur, wo der Server alle aktuellen Dateien und deren Historie speichert und die Clients sich alle Dateien kopieren und lokal entwickeln. Oder aber eine Host-Struktur, wo der Host alle Dateien speichert und die Entwickler an bloßen Terminals arbeiten. Alle gängigen Verwaltungssysteme funktionieren nach dem Client-Server-Prinzip. In beiden Varianten müssen die neuen Versionen der Dateien nach den gewünschten Änderungen auf den Server zurück gespeichert werden. Dabei kann es zu Konflikten kommen, wenn simultan von anderen Entwicklern schon Änderungen der Dateien vorgenommen wurden, die von dem System angezeigt werden müssen.

5.1.1 Merge

Der Vorgang des Abgleichens mehrerer Änderungen, die an verschiedenen Versionen derselben Dateien getätigt wurden, heißt *Merge*[1]. Das Ergebnis eines Merge-Vorgangs ist eine einzige Datei, die alle Aspekte der verschiedenen Datei-Versionen vereinigt. In vielen Fällen gelingt der Merge-Vorgang automatisch, ohne menschliche Interaktion. Werden jedoch verschiedene Änderungen zusammengeführt, die den gleichen Abschnitt einer Datei betreffen, so kommt es zu einem Merge-Konflikt. Dieser kann nur manuell aufgelöst werden.

[1] *to merge* – vereinigen, zusammenführen

5.1.2 Branching, Trunk und Fork

Die Verdopplung eines Objekts, beispielsweise einer Quelltextdatei oder ein Verzeichnisbaum, heißt in der Versionsverwaltung (revision control) und dem Softwarekonfigurationsmanagement *Branching*. Die separate Weiterentwicklung beider Zweige *(branch)* geschieht nun parallel. Nach einem Branching ist es möglich, die beiden Zweige wieder zusammenzuführen *(merge)*. Beachten Sie, dass ein *Merge* von Entwicklungszweigen Änderungen auf der Projektmanagementebene bewirkt, während der Merge zweier Quelltextdateien sich zunächst nur auf der Programmierebene abspielt; technisch sind beides freilich gleiche Prozesse.

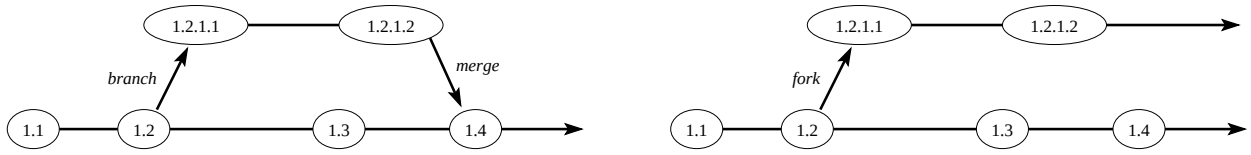

Abbildung 5.1: Versionen durch Branch, Merge und Fork eines Projektes

Ein spezieller Zweig, der *Trunk* oder *HEAD*, ist der Hauptzweig des Systems. Er enthält üblicherweise die aktuelle stabile Verson des Systems. Oft wird diese verzweigt und etwaige Bug-Fixes werden von Zweigen in den Trunk zusammengeführt. Jeder Zweig kann mit dem *Trunk* zusammengeführt werden, auch wenn er nicht der Elternzweig ist.

Wird bei einem Branching von vornherein ein Merge nicht beabsichtigt, so spricht man von einem *Fork* oder einer *(Projekt-) Abspaltung*. Ein Fork ist also ein Entwicklungszweig, bei dem Teile des Projektes oder das gesamte Projekt kopiert werden und unabhängig weiterentwickelt werden. Ein solcher Fall liegt vor, wenn beispielsweise die bisherige Software einem neuen Zweck angepasst wird oder es neue Lizenzmodelle für den Zweig geben soll. Bekannte Projekte, die aus Forks entstanden sind, sind Firefox und Thunderbird aus dem Mozilla-Projekt, OpenOffice aus StarOffice, aMuke aus xMule, oder das NTSF-Dateisystem aus HPFS, dem Dateisystem von OS/2.

Voraussetzung für ein funktionierendes Verzweigen und Zusammenführen ist eine ausreichende Modularität der Software und eine genügend klare funktionale Abstimmung der Entwicklungsstränge. Wenn in Abb. 5.1 beispielsweise Version 1.2.1.1 wesentlich auf Funktionen der Version 1.2 aufbaut, die aber in Version 1.3 verändert wurden, wird ein Zusammenführen womöglich *inhaltlich* nicht mehr möglich sein, da die neue Funktion zu Fehlern in 1.2.1.2 führt. Im schlimmsten Fall kann der Konflikt nur gelöst werden, indem man die Teilprojekte abspaltet.

5.2 CVS

Concurrent Versions System (CVS), auch *Concurrent Versioning System*, ist ein Software-System zur Versionsverwaltung von Dateien, insbesondere von Software-Quelltext. Es hält alle Änderungen einer gegebenen Menge von Dateien vor und ermöglicht so kollaborative Entwicklung. Ein durch CVS verwaltetes Projekt heißt *Modul (module)*.

CVS hat eine Client-Server-Architektur. Alle Dateien eines Softwareprojektes werden an einer zentralen Stelle, dem *Repository*, gespeichert. Dabei können jederzeit einzelne Dateien verändert werden, es bleiben jedoch alle früheren Versionen erhalten, stets einsehbar und wiederherstellbar. Auch können die Unterschiede zwischen verschiedenen Versionen angezeigt werden und so ein Überblick über die einzelnen Versionen der Dateien und der dazugehörigen Kommentare gewonnen werden. Die Entwickler dagegen arbeiten lokal auf

ihren Rechnern im sogenannten *Workspace* oder der *Workbench*. D.h. die Entwicklung, also editieren, kompilieren, testen und debuggen, wird ausschließlich auf den lokalen Rechnern der Teammitglieder durchgeführt. Um als Entwickler Schreibrechte zu haben, muss man sich an dem CVS-Server als User authentifiziert einloggen.

CVS verwendet ein *Branch-Modell*, um isolierte parallele Entwicklungen zu ermöglichen, die aber gegenseitig voneinander abhängen. Hierbei ist ein *Branch* eine Art verteilter Ar-

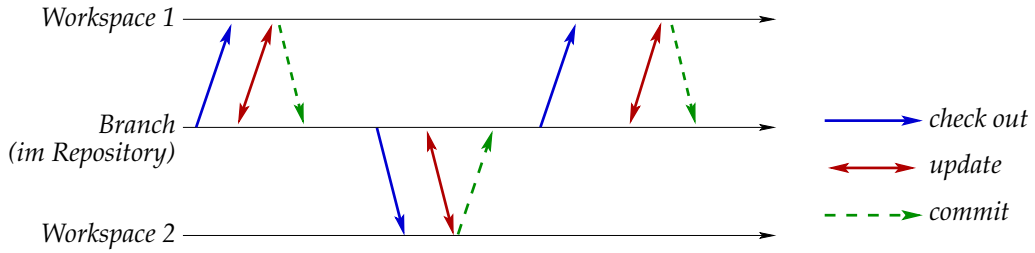

Abbildung 5.2: Ablauf der kollaborativen Entwicklung: das *Branch-Modell* von CVS

beitsspeicher, der von den Teammitgliedern durch ihre Änderungen am Projekt aktualisiert wird. Ein *Branch* stellt somit stets einen aktuellen Zustand eines Teilprojekts dar, der *HEAD* als spezieller *Branch* entsprechend repräsentiert das gesamte Projekt. Ein Projekt wird zu Beginn *importiert*, d.h. auf dem Repository gespeichert. Der Arbeitsablauf mit CVS geschieht danach wie folgt (Abb. 5.2):

1. *Checkout.* Ein Entwickler des Moduls holt sich zunächst den aktuellen Stand aller Dateien eines Projekts aus dem Repository. Dies bezeichnet man bei CVS als „auschecken". Dabei werden von CVS Metadaten angelegt, die es ermöglichen, zu erkennen, welche Versionen der Dateien zuletzt ausgecheckt wurden.

2. *Update.* Nachdem Änderungen an einer oder mehreren Dateien im lokalen Workspace vorgenommen worden sind und der endgültige oder (bei größeren Änderungen) zumindest konsistente Zwischenstand des modifizierten Systems erreicht ist, beginnt der Entwickler mit dem *Update*-Befehl die Vorbereitungen, die neuen Versionen der geänderten Dateien zurück ins Repository zu übergeben. Der *Update*-Befehl vergleicht zunächst einfach alle lokalen Dateien mit denen des Repositorys. Es können sich dabei Konflikte ergeben, wenn andere Entwickler seit dem letzten Check-out, Update oder Commit (s.u.) Dateien geändert haben. Der Update-Befehl zeigt dabei alle Unteschiede zur aktuellen Repository-Version an, und der Entwickler kann sie in seine lokale Version übernehmen. eine Datei verändern, Normalerweise können Konflikte durch einfaches Zusammenführen *(Merge)* gelöst werden, wenn unterschiedliche Teile einer Datei verändert wurden, oder müssen manuell, ggf. in Absprache mit den anderen beteiligten Entwicklern, ausgeräumt werden. Eine ähnliche Funktion übernimmt in Eclipse der Befehl *Synchronize*.

3. *Commit*.[2] Der Entwickler lädt die aktualisierten Dateine hoch ins Repository. Dabei wird in CVS automatisch die Versionsnummer erhöht. Im Falle eines Konfliktes muss in der Regel manuell nachgearbeitet werden (Abb. 5.3).

Da jeder Enrwickler jede Datei verändern kann, auf die er Zugriff hat, geht CVS von einem *optimistischen Team-Modell* aus. Es heißt deshalb „optimistisch", da es davon ausgeht, dass Konflikte selten auftreten. Voraussetzung dazu ist eine hinreichende Modularisierung sowohl der Software als auch der Aufgaben- oder zeitlichen Verteilung des Teams.

[2] *commit*: festlegen, übergeben

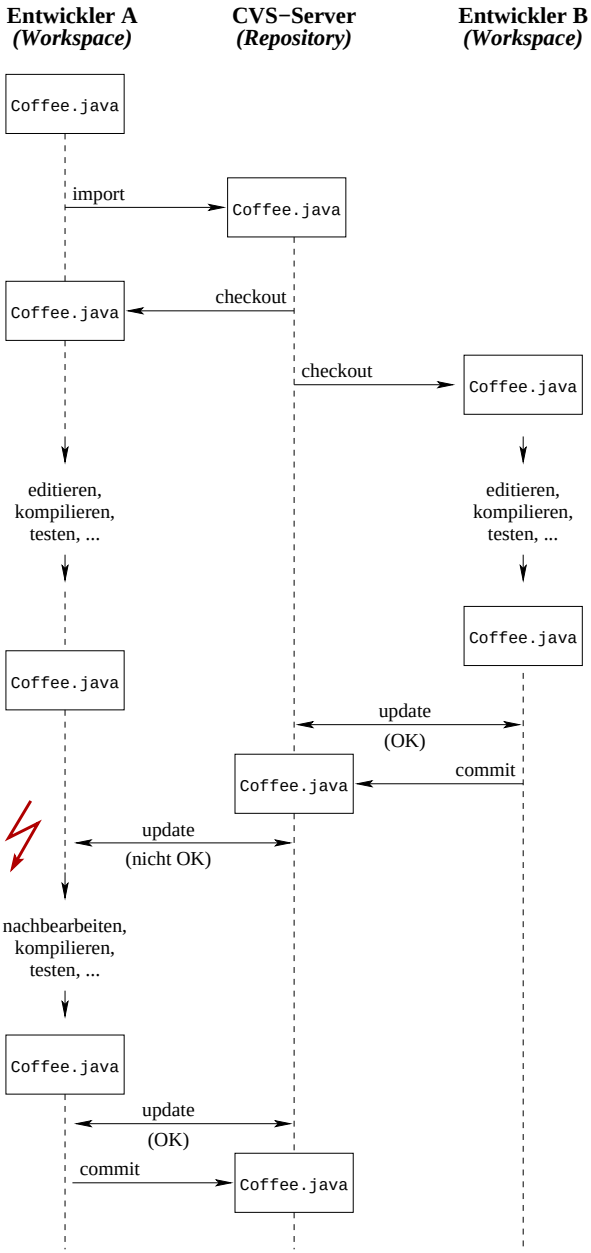

Abbildung 5.3: Der Ablauf der kollaborativen Entwicklung mit CVS bei einem Konflikt

5.2.1 Versionierung

In CVS kann ein Projekt in zwei Arten vorliegen, nämlich veränderbar in einem *Branch* (wie im vorigen Abschnitt beschrieben) oder unveränderlich als eine *Version*. Eine *CVS-Resorce*, also eine Datei, ein Verzeichnis oder ein Projekt, werden versioniert, um einen Schnappschuss ihres aktuellen Zustands zu erstellen. In CVS wird eine Resource versioniert, indem man sie mit einem *Version Tag* versieht. Eine so versionierte Resource wird unveränderlich im Repository gespeichert und ist damit auch später immer wieder reproduzierbar.

Es gibt in CVS zwei Möglichkeiten, eine Version einer Resource zu erstellen, nämlich einerseits im lokalen Workspace und danach mit *commit* im Repsoitory, oder direkt in einem *Branch* im Repository. Wird eine Resource im Workspace versioniert, so werden auch nur die lokal enthaltenen Resourcen markiert, während bei Versionierung in einem *Branch* alle zu diesem Zeitpunkt darin enthaltenen Dateien versioniert werden. Die üblicherweise durchgeführte Versionierung ist diejenige vom Workspace aus.

5.3 Subversion

Subversion (SVN)[3] ist eine Open-Source-Software zur Versionsverwaltung von Dateien und Verzeichnissen. Sie steht unter einer Apache/BSD-artigen Lizenz und ist damit freie[4] Software. Es funktioniert im Wesentlichen wie CVS, insbesondere der Arbeitsablauf ist gleich (Abb. 5.2, 5.3). Subversion wurde entwickelt, einige Schwächen von CVS zu beheben. Da CVS in Entwicklerkreisen sehr verbreitet war und ist, ist es mit Bedacht in der Bedienung sehr ähnlich gehalten.

Ein entscheidender Unterschied zwischen CVS und SVN liegt in der Rangfolge von örtlicher (L) und zeitlicher (T) Kennzeichnung der Inhalte eines Projektarchivs (P): während in CVS primär örtlich, sekundär zeitlich adressiert wird (P:L.T), werden in SVN die Koordinaten umgekehrt (P:T.L). SVN bildet die erfahrungsgemäße Realität von Veränderungen damit anschaulicher nach, denn jede Veränderung benötigt Zeit, und ihre Endpunkte sind das Vorher (Revision vor Änderung) und das Nachher (Revision nach der Änderung). SVN versioniert oder revisioniert also grundsätzlich das gesamte Projektarchiv und damit jeweils die gesamte Verzeichnisstruktur, während CVS auf der unabhängigen Versionierung jedes einzelnen Inhalts beruht.

5.3.1 Revisionen

Das Versionsschema von Subversion bezieht sich nicht mehr auf einzelne Dateien, sondern auf das ganze Projektarchiv, mit jeder Änderung bekommt es eine neue *Revisionsnummer* zugeordnet. Eine *Revision* in Subversion ist ein „Schnappschuss“ des Repositorys zu einem bestimmten Zeitpunkt. Jede Revision kann mit einem Kommentar versehen werden („Bug *xy* behoben“).

Mit Hilfe der Revisionen kann man einfacher – und im Gegensatz zu CVS konsistent – eine exakte Version beschreiben (z. B. „Revision 2841“ statt „Version vom 23. März 2008 20:56:31 UTC“). Die Revisionsnummer einer Datei entspricht dabei der Revisionsnummer des Projektarchivs, als sie das letzte Mal geändert wurde, die Revisionsnummer eines Verzeichnisses entspricht der höchsten Revisionsnummer der enthaltenen Dateien und Verzeichnisse. Die Abfolge der Revisionsnummern einer einzelnen Datei kann also durchaus lückenhaft sein, wenn die Datei nicht bei jedem Commit am Repository geändert wurde. Beispielsweise könnte eine Datei bei der Revision 25 zum Projektarchiv hinzugefügt wor-

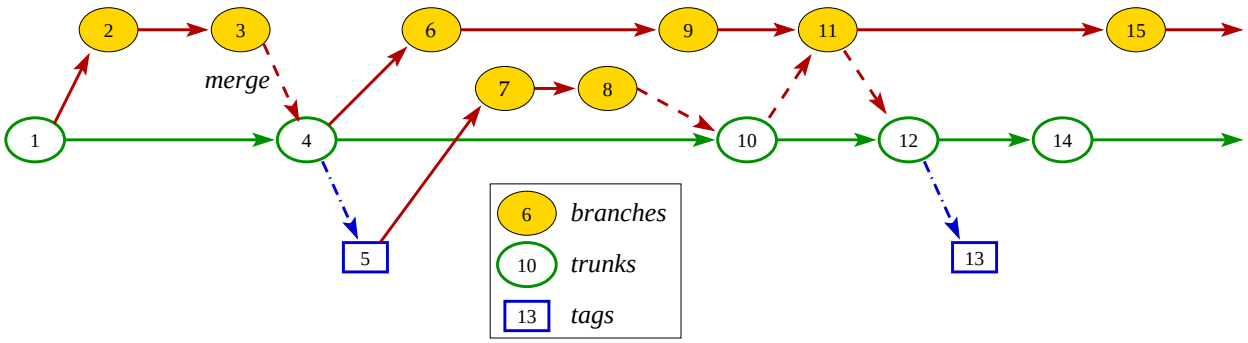

Abbildung 5.4: Revisionen und das *Tag*-und-*Branch*-Konzept von Subversion

den sein, und einmal in der Revision 48 und 52 verändert worden sein. Bei einem Checkout

[3] *http://subversion.tigris.org*

[4] „frei“ ist hier im selben Sinne wie in „freie Rede“ zu verstehen, aber nicht wie in „Freibier“ [11]. Freie Software ist ohne Beschränkungen zu benutzen, und ist nicht nur kostenlos.

einer Datei wird die größte Revisionsnummer ausgecheckt, die kleiner oder gleich der angeforderten ist. Wird in dem Beispiel die Revision 52 angefordert, so wird die Revision 52 der Datei ausgecheckt, wird hingegen die Revision 51 angefordert, liefert Subversion die Inhalte von Revision 48.

5.3.2 *Tag*- und *Branch*-Konzept von SVN

Während *Tags* und *Branches* in CVS eine klar definierte Bedeutung haben, kennt Subversion nur das Konzept der Kopie, die je nach *Nutzungsart* einen *Tag*- oder *Branch*-Charakter haben kann. Jede Kopie einer Datei oder eines Verzeichnisses ist in Subversion demnach automatisch ein *Branch* dieser Datei oder des Verzeichnisses. *Tags* entstehen in Subversion, wenn man eine Kopie anlegt und später auf ihr keine Änderungen mehr vornimmt. Aufgrund des Fehlens einer *Tag*- und *Branch*-Semantik muss die Strukturierung und Verwaltung von *Tags* und *Branches* der Benutzer und Administrator durchführen.

Üblicherweise werden in einem Projekt die Basisverzeichnisse *Trunk*, *Branches*, und *Tags* angelegt. Ein *Trunk* enthält dabei den Letztstand des Projekts, in *Branches* werden weitere Unterverzeichnisse mit alternativen Entwicklungspfaden verwaltet, und in *Tags* eine Kopie von *Trunk* oder einem der *Branches* als Unterverzeichnis angelegt. Zur besseren Übersicht werden je nach Projektanforderungen *Tags* und *Branches* noch in weitere Unterverzeichnisse unterteilt.

Durch das Fehlen einer festgelegten Semantik für Tags ist die Angabe der Revisionsnummer als Referenz, beispielsweise beim Bug-Tracking oder in der Dokumentation, unabdingbar. Da Dateien in Subversion auch (versionskontrolliert) umbenannt werden können, kann die Projektstruktur jederzeit geänderten Anforderungen angepasst werden.

5.3.3 Der *Merge*-Prozess in SVN

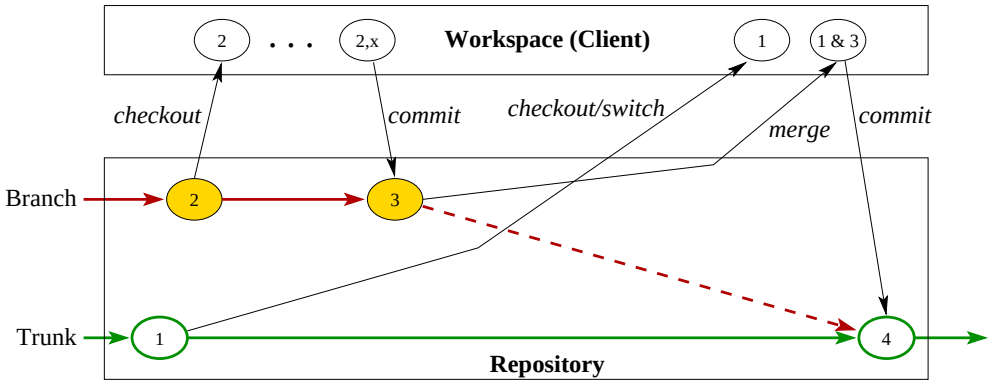

Abbildung 5.5: Der *Merge*-Prozess in SVN. Hier arbeitet der Entwickler an einem *Branch* (Revision 2 und 3); um den Branch an Revision 3 mit dem *Trunk* zusammenzuführen, muss er zunächst zum *Trunk* wechseln (per *checkout* oder kürzer über *switch*), wodurch in seinem Workspace seine *Branch*-Version überschrieben wird; erst durch den *merge*-Befehl stellt er dort die *Merge*-Version („1 & 3“) her, die er dann per *commit* in den Trunk im Repository speichert. Im Repository ist damit am Ende die *Branch*-Version 3 in den Trunk zusammengeführt (gestrichelte Linie).

Kapitel 6

Technische Qualität: Softwaremetriken

You can't control what you can't measure.
Tom DeMarco[1]

Eine *Softwaremetrik* ist ein Maß für eine bestimmte Eigenschaft von Software oder seiner Spezifikationen. Ziel ist es, für eine Software Maßzahlen zu bestimmen und so Eigenschaften zu messen und somit vergleichbar zu machen. Auf diese Weise soll Softwareentwicklung quantitativ planbar und kontrollierbar werden.

Softwaremetriken spielen eine wesentliche Rolle bei der Messung der *technischen Qualität* der Software, oder *inneren Qualität*, die vom Software-Ersteller selbst wahrgenommen wird (im Gegensatz zur „äußeren", fachlichen Qualität, die vom Anwender wahrgenommen wird, s. Def. 7.1 auf S. 48). Die technische Qualität wird bestimmt anhand von Kriterien wie Anpassbarkeit, Erweiterbarkeit, Wiederverwendbarkeit, Administrierbarkeit, Robustheit, Testbarkeit oder Verständlichkeit [4].

Softwaremetriken lassen sich grob in eine der vier Kategorien Quantitätsmetrik, Komplexitätsmetrik, Objektmetrik oder Analysemetrik unterteilen. Sie lassen sich in den meisten Fällen allein anhand des Quelltextes ermitteln.

6.1 Lines of Code (LOC)

Lines of Code (LOC), auch *Source Lines of Code (SLOC)*, gibt die Anzahl der Quelltextzeilen eines Programms von Software an. Es ist eine der gebräuchlichsten Softwaremetriken und wird typischerweise verwendet, um den Aufwand für die Entwicklung eines Programms zu prognostizieren oder um Programmieraufwand oder Softwareproduktivität zu schätzen. LOC ist eine Quantitätsmetrik.

Diese Softwaremetrik ist scheinbar sehr einfach, man muss ja nur die Zeilen zählen. Allerdings beginnen hier die Schwierigkeiten: Was ist denn eigentlich zu zählen? Je nach Definition zählt man …

- nur Zeilen, in denen ausführbare Programmteile stehen,
- auch Kommentarzeilen
- auch Leerzeilen dazwischen.

Zusätzlich muss bei jeder Variante genau festgelegt sein:

- Wie zählt man Anweisungen, die sich über mehrere Zeilen erstrecken (wie Verzweigungen oder Schleifen)?

[1] T. DeMarco *Controlling Software Projects: Management, Measurement and Estimation.* ISBN 0-13-171711-1

- Werden auch Zeilen gezählt, die nur Klammern oder Blockbefehle enthalten?
- Was ist, wenn in einer Zeile mehrere Befehle stehen?

Je nach Definition wird ein und dasselbe Quelltextfragment, gemessen in LOC, „kürzer“ oder „länger“. Das Programm in Abb. 6.1 hat 13 Zeilen, inklusive einer Leerzeile. Ohne

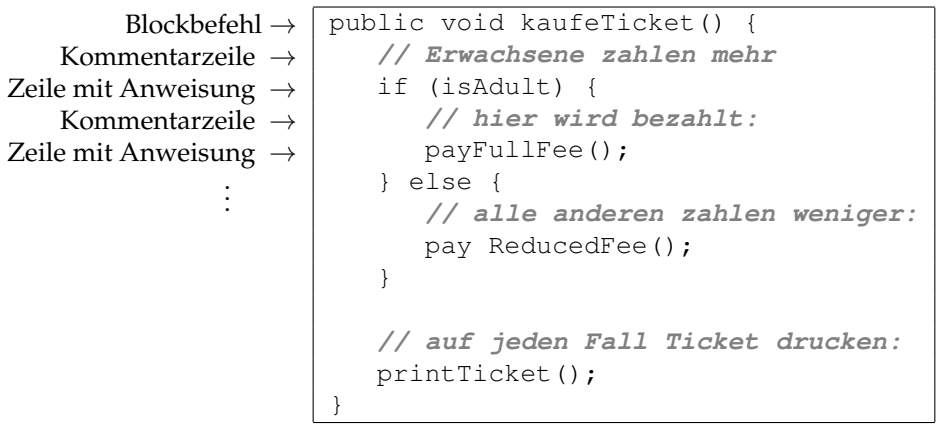

Abbildung 6.1: Ein Quelltextfragment

Leerzeilen sind es entsprechend 12 Zeilen, ohne Kommentare nur noch 8 Zeilen. Welche Zählweise angemessen ist, hängt von der Absicht ab, die man mit der Metrik verfolgt.

- Man möchte abschätzen, wie viele Ordner ein Papierausdruck des Quelltext füllen würde: Dazu müssen alle Zeilen gezählt werden, auch Leerzeilen, denn sie nehmen auf dem Papier gleich viel Raum ein.
- Man möchte bewerten, wieviel ein Programmierteam geleistet hat. Die LOC sollen dann den erstellten Programmumfang messen, d.h. es werden Kommentarzeilen mitgezählt, Leerzeilen allerdings nicht, denn sie stellen keine Leistung dar.
- Man möchte abschätzen, wie schwer es einem Neueinsteiger fallen wird, sich in das Programm einzuarbeiten. Je länger das Programm, desto länger wird die Einarbeitung dauern; allerdings machen Kommentare das Programm eher verständlich, daher sollten sie nicht mitgezählt werden.

Anhand des Beispiels der LOC kann man ein rekursives Phänomen verdeutlichen, das allgemein jede Softwaremetrik betrifft:

Eine Messung beeinflusst den gemessenen Aspekt.

Wenn nämlich beispielsweise Entwickler wissen, dass ihre Leistung nach LOC ohne Leerzeichen und mit Kommentarzeilen bewertet und vielleicht sogar bezahlt wird, werden sie ihr Programmierverhalten danach ausrichten und mehr Kommentare schreiben. Ist das nicht Verzerrung des Ergebnisses, ja Betrug? Keineswegs, eine Metrik *muss* so gestaltet sein, dass die Optimierung ihrer Kennzahl zu einem wünschenswerten Ergebnis führt. Eine Metrik wirkt auf diese Weise handlungsleitend, und wenn sie das „Richtige“ misst, führen die durch sie beeinflussten Handlungen zum Erfolg des Projekts. Was also so unscheinbar mit einer kleinen Routine zur *Messung* von Leistung begonnen hat, *verändert* nun die Softwarequalität, hier beispielsweise den Qualitätsaspekt Verstehbarkeit (durch mehr Kommentare).

Weitere Quantitätsmetriken sind beispielsweise DOC *(inline documentation lines of code)*, das die Anzahl Zeilen von Inline-Dokumentation wiedergibt, oder NCSS *(non commenting source statements)*, das die Anzahl der effektiven Programmanweisungen misst. Für Java sind dies vereinfacht diejenigen Zeilen, in denen eine öffnende geschweifte Klammer `{` oder ein Semikolon `;` steht.

6.2 Zyklomatische Komplexität von McCabe

Von McCabe stammt eine etwas komplizierte Metrik [9, §4.3.2], die *zyklomatische Komplexität* CCN *(cylomatic complexity number)*, oder *McCabe-Metrik*, von Thomas McCabe 1976 eingeführt. Anders als LOC ist sie genau definiert. Der Ausdruck „Komplexität“ sagt aus, was gemessen werden soll, nämlich wie kompliziert die untersuchte Software strukturiert ist. Die Metrik erhält ein Quelltextfragment und liefert eine natürliche Zahl.

Zum Verständnis der zyklomatischen Komplexität muss man sich McCabes Vorstellung von struktureller Schwierigkeit verdeutlichen: Eine einfache Folge von Befehlen ist nicht schwierig zu verstehen, was ein Programm schwierig macht sind bedingte Anweisungen, also Verzweigungen und Schleifen. Die zyklomatische Komplexität ist so definiert, dass man sie aus dem Quelltext einer strukturierten Programmiersprache ablesen kann [9, S. 61]:

Anzahl der Verzweigungen	(`if` und Zweige bei `switch/case`)
+ Anzahl der Schleifen	(`for, while, do/while, ...`)
+ 1	
= zyklomatische Komplexität	

Die stets hinzu addierte 1 ist im Prinzip willkürlich und bewirkt, dass selbst das einfachste Programm (ohne eine Anweisung) mindestens 1 ergibt. Nach McCabe unterscheidet man drei Klassen von zyklomatischer Komplexität, nämlich *niedrig* für eine zyklomatische Komplexität bis 10, *mittel* für eine zyklomatische Komplexität zwischen 10 und 20, *hoch* für eine zyklomatische Komplexität größer als 20 und kleiner als 50, und *undurchschaubar* für eine zyklomatische Komplexität größer als 50.

Interessanterweise würde eine noch so komplizierte mathematische Gleichung[2] die zyklomatische Komplexität *nicht* erhöhen, da dort keine Verzweigung auftaucht. Die derzeit schnellste bekannteste Lösung des Problem des Handlungsreisenden (TSP)[3], eine Lösung über die dynamische Optimierung, besteht aus lediglich 4 Schleifen und einer bedingten Anweisung zum Schreiben einer Tabelle, hat also die zyklomatische Komplexität 5, während die Zeitkomplexität dieses Algorithmus bezüglich der zu besuchenden Orte nicht einmal polynomial ist. Die zyklomatische Komplexität besagt also nichts über die Laufzeit eines Programms, diese Metrik misst lediglich den Verzweigungsgrad von Quelltext.

Betrachten wir beispielsweise die beiden folgenden Methoden, die jeweils eine Jahreszahl n als Eingabe erwarten und genau dann `true` zurück geben, wenn n ein Schaltjahr ist. Hierbei ist n ein Schaltjahr, wenn n durch 4 teilbar ist, außer wenn es durch 100 teilbar ist, es sein denn (Ausnahe von der Ausnahme), wenn n durch 400 teilbar ist.

```
public boolean istSchaltjahr(int n) {
   if (n % 4 != 0) {
      return false;       // Schaltjahre sind immer durch 4 teilbar
   } else if (n % 100 != 0) {
      return true;        // aber nicht durch 100, dann sicher Schaltjahr
   } else if (n % 400 == 0) {
      return true;        // sonst, falls durch 400: doch Schaltjahr
   } else {
      return false;       // alle anderen keine Schaltjahr
   }
```

```
public boolean istSchaltjahr(int n) {
   return ((n % 4 == 0) && (n % 100 != 0)) || (n % 400 == 0);
}
```

[2] Denken Sie an die Formel für die Oberfläche eines Sphäroids im 1. Semester!

[3] §8.4 im Skript *Algorithms and Optimization* (*http://www3.fh-swf.de/fbtbw/devries/download/algorithmics.pdf*)

Das obere Quelltextfragment weist 3 Verweigungen auf und hat damit die zyklomatische Komplexität 4, das untere besteht aus nur einer, wenn auch komplizierteren, Formel und hat somit die zyklomatische Komplexität 1.

Wenn Entwickler wissen, dass ihre Leistungen gemäß dem McCabe-Wert berechnet werden, werden sie ihren Code dahin gehend optimieren, möglichst wenig Verzweigungen oder Schleifen und möglichst viele Formeln zu verwenden. Letztendlich wird er dadurch übersichtlicher, man kann ihn leichter warten, die Fehleranfälligkeit wird geringer und verbliebene Fehler werden einfacher gefunden.

6.3 Objektmetriken

Bei der Verwendung einer objektorientierten Sprache funktioniert die Messung der zyklomatischen Komplexität zwar immer noch, allerdings berücksichtigt sie in keiner Weise Methodenaufrufe. Nun kann ein Programm mit vielen Methodenaufrufen genauso unübersichtlich sein wie eines mit vielen Verzweigungen in einer längeren Methode. Daher wurde für objektorientierte Sprachen verschiedene Metriken eingeführt, die man unter dem Begriff *Objektmetriken* oder *objektorientierte Metriken* zusammenfasst.

6.3.1 WMC

WMC (weighted methods per class) [9, §4.3.4] ist für eine gegebene Klasse definiert durch

$$WMC = \text{Summe der Komplexitäten aller Methoden der Klasse.}$$

Die Komplexität einer einzelnen Methode kann hierbei die zyklomatische Komplexität sein. Dann wird jede Methode zumindest als 1 gezählt, denn das ist der Minimalwert von McCabe, und viele kleine Methoden führen nach der Formel zu hohen ungünstigen WMC-Werten, ebenso wie wenige sehr große Methoden.

Im Sinne dieser Metrik wird also ein Entwickler seinen Code optimieren, indem er die zyklomatische Komplexität jeder einzelnen Methode einer Klasse bestimmt und daraus den WMC der ganzen Klasse berechnet. Wird der WMC zu groß, so wird er Methoden in neue Klassen auslagern und ihn somit vermindern. Per se ist das allerdings nicht unbedingt eine gute Strategie, denn so können fachlich zusammenhängende Methoden unnötig zerschlagen werden. Als Beispiel möge hier die Klasse `Math` der Java SE API dienen, die in sehr sinnvoller Weise gängige mathematische Funktionen in (wenn auch statische) Methoden zusammenfasst und somit einen sehr hohen WMC haben dürfte.

6.3.2 DIT

DIT (depth of inheritance tree) [4] ist für eine gegebene Klasse definiert als ihre Tiefe im Vererbungsbaum, d.h. in welcher Generation sie von einer Klasse erbt, die von einer Klasse erbt, die von einer Klasse erbt, ... Eine hohe Vererbungstiefe spricht prinzipiell für eine höhere Fehleranfälligkeit, denn von jeder vererbenden Klasse werden eben auch noch unentdeckte Bugs geerbt. DIT ist somit ein Maß für die Wiederverwendbarkeit im System.

In Java, wo jede Klasse von der Klasse `Object` erbt, ist der Wert von DIT mindestens 1.

6.3.3 NOC

NOC (number of children) [4] ist als die Anzahl der direkten Unterklassen der betrachteten Klasse definiert. Sie kann einerseits den Grad an Wiederverwendbarkeit messen, aber auch ein Indikator für eine übertriebene Verwendung von Vererbung sein. Ähnlich wie bei PtDIT

gilt, dass eine hohe Anzahl von Unterklassen durch Vererbung unentdeckter Bugs eine hohe Fehleranfälligkeit bewirken kann.

6.3.4 CBO oder Ce

CBO (coupling between object classes) [4] misst die Kopplung einer gegebenen Klasse mit anderen durch ausgehende Beziehungen *Ce („efferent coupling")* zu anderen Klassen, beispielsweise durch Methodenaufrufe, Attributzugriffe, Vererbungen, Eingabeparameter, Rückgabetypen oder Exceptions. Bestehen viele Verbindungen zu anderen Klassen, so kann das der Modularisierung widersprechen und die Wiederverwendung erschweren. Außerdem kann diese Metrik als ein Indikator für den Testumfang dienen, der durch komplexe Beziehungsgeflechte steigt.

6.3.5 RFC

RFC (response for a class) [4] misst die Anzahl der unterschiedlichen Methoden, die direkt in den Methoden der Klasse aufgerufen werden. Je mehr Methoden aufgerufen werden, desto komplexer ist die Implementierung und desto höher Testaufwand.

6.3.6 LCOM

LCOM (lack of cohesion in methods) [4] ist der relative Anteil jeweils zweier Methoden der betrachteten Klasse, die keine oder nur wenige ihrer Attribute gemeinsam nutzen. Diese Metrik misst also den inneren Zusammenhalt der Klasse. Eine Klasse mit einem niedrigen LCOM ist fehleranfällig, da dies auf ein schlechtes Design schließen lässt.

6.4 Analysemetriken

Neben den Softwaremetriken, die eine absolute Zahl für eine einzelne Klasse ausgeben, gibt es die *Analysemetriken*, die im gesamten System Abweichungen von vorgegebenen Regeln oder festgelegten Standards identifizieren oder potenzielle Fehlerstellen entdecken.

Die Metrik *DRY (don't repeat yourself)* [4] überprüft, ob Quelltextstellen doppelt vorhanden sind und kann als Verhältnis zwischen der Anzahl der Klassen mit Code-Duplizierung zu der Anzahl aller Klassen ausgedrückt werden. Es ist damit ein Indikator für die Wartbarkeit und die Wiederverwendbarkeit des Systems.

Die Einhaltung der Programmierrichtlinien und Quelltextkonventionen wird mit Hilfe der Metrik *STY (styleness)* [4] geprüft. Der Messwert kann als Verhältnis der Anzahl aller Klassen mit Verletzungen der Quelltextkonventionen relativ zu der Anzahl aller Klassen berechnet werden.

6.5 Halstead-Maße

Die *Halstead Software Science* ist recht kompliziertes System von Maßen, die ursprünglich dazu dienten, den Programmieraufwand zu quantifizieren.[4] Die Interpretationen der einzelnen Maße sind allerdings oft unklar, daher hat das System heute eher historische Bedeutung. Jedoch verwendet man einige der Kennzahlen des Halstead-Systems, um eine Daumenregel für die Anzahl von Fehlern zu gewinnen, die in einem Programm gefunden werden

[4] *http://yunus.hacettepe.edu.tr/ sencer/complexity.html* [Letzter Abruf 14.4.2008]

sollten. Dazu wird die „Programmlänge“ N und das „Volumen“ V berechnet und daraus die geschätzte Fehlerzahl B durch die Formeln[5] in Abb. 6.2 bestimmt. Hierbei ist ein Ope-

Verwendete Variablen *(primitive Maße)*

N_1: Gesamtzahl der Operatoren
N_2: Gesamtzahl der Operanden
n_1: Gesamtzahl *verschiedener* Operatoren
n_2: Gesamtzahl *verschiedener* Operanden

Halstead-Maße:

Programmlänge:	$N = N_1 + N_2$
Programmvokabular:	$n = n_1 + n_2$
Halstead-Volumen:	$V = (N_1 + N_2)\log_2(n_1 + n_2)$
Halstead-Schwierigkeitssgrad:	$D = \frac{n_1 N_2}{2n_2}$
Halstead-Aufwand:	$E = V \cdot D$
Intelligenzgehalt:	$I = V/D$
geschätzte Fehlerzahl:	$B = V/3000$

Abbildung 6.2: Formeln für Halstead-Maße

rator etwa “+”, “∗”, aber auch ein Anweisungstrenner “;”, Operanden sind u.a. Literale, Konstanten und Variablen.

Als Daumenregel hat sich die Fehlerformel bewährt, allerdings steckt eine gewisse Gefahr in der „Optimierung“ des Quelltexts bezüglich dieser Kennzahl: Man kann das Halstead-Volumen des Programms verkleinern, indem man dieselbe Variable für mehrere Berechnungen verwendet; das *kann* sinnvoll sein (z.B. stets i, j, k als Schleifenindizes zu verwenden), ist aber *per se* kein Selbstzweck, denn durch sinnvolle Namensgebung wird der Quelltext lesbarer und verstehbarer.

Empirisch gilt die *Halstead'sche Längengleichung*[4]

$$N \approx n_1 \log_2 n_1 + n_2 \log_2 n_2.$$

Nach einer Theorie des Psychologen John Stroud ist ein *Moment* definiert als diejenige Zeitspanne, die das menschliche Gehirn für die elementarste Unterscheidungsoperation *(discrimination)* benötigt. Die *Stroud-Zahl* S ist entsprechend die Anzahl der Momente pro Sekunde. Für die meisten Menschen gilt $5 \leqq S \leqq 20$. S hat die Einheit einer Frequenz [$\mathrm{Hz} = \mathrm{sec}^{-1}$]. Mit der Stroud-Zahl ergibt sich die Zeit T, die ein Mensch zum Verständnis eines Quelltextes benötigt, durch die Formel[4]

$$T = \frac{n_1 N_2 \log_2 n}{2\,S} \cdot (n_1 \log_2 n_1 + n_2 \log_2 n_2). \tag{6.1}$$

Bei gegebenem Quelltext gilt also $T \propto 1/S$, d.h. bei gleichem Quelltext benötigt ein Mensch mit einer höheren Stroud-Frequenz, also mehr Momenten pro Sekunde, eine längere Zeitspanne T als jemand mit einer niedrigeren Stroud-Frequenz.

6.6 Funktionspunktanalyse

Die *Funktionspunktanalyse (function point analysis)* ist ein Verfahren zur Bestimmung des fachlich-funktionalen Umfangs eines Softwaresystems oder eines Softwareprojektes und ist ein

[5] *http://www.sei.cmu.edu/str/descriptions/halstead.html* [Letzter Abruf 14.4.2008]

Teilverfahren zur Aufwandsschätzung und Größenbestimmung von Anforderungen sowie zur Ermittlung von Softwareproduktivität. Der Schwerpunkt der Analyse liegt auf den fachlichen Anforderungen des zu erstellenden Systems. Dadurch kann diese Methode schon recht früh, beispielsweise bei Vorlage eines Lastenhefts, eingesetzt werden.

Ein *Funktionspunkt* wird vergeben für jedes Datenelement, das als Eingabe oder Ausgabe einer Endanwenderfunktion der betrachteten Software benötigt wird. Eine solche Funktion kann beispielsweise eine Datenbankabfrage sein. Funktionspunkte werden sin fünf Kategorien eingeteilt, Eingaben *(inputs)*, Ausgaben *(outputs)*, Abfragen *(inquiries)*, Dateien *(files)* und Schnittstellendaten *(interfaces)*.

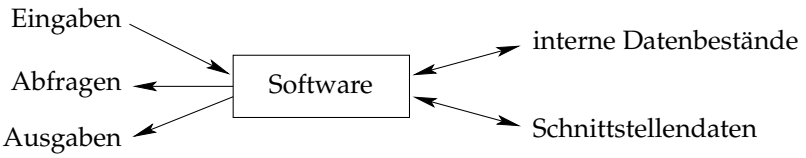

Abbildung 6.3: Die fünf Kategorien der Datenelemente zur Ermittlung der Funktionspunkte

- Datenbestände sind interne Datenspeicher des Systems, also Datenbanken oder Dateien.
- Eingabedaten sind jene Daten oder Kontrolldaten, die in das System eingegeben werden und von einem elementaren Prozess verarbeitet werden. Sie dienen in der Regel zur Pflege der Datenbestände.
- Ausgabedaten sind Daten, die von einem elementaren Prozess nach außen gesendet werden, also z.B. zur Ausgabe auf den Bildschirm. Diese Daten müssen zuvor entweder durch eine mathematische Funktion errechnet oder durch anderweitige Prozesslogik aus den Datenbeständen abgeleitet worden sein.
- Dagegen sind Abfragen sehr einfache, elementare Funktionen, die Daten aus den Datenbeständen ausgeben, ohne diese zuvor verarbeitet zu haben.
- Schnittstellendaten sind Daten, auf die durch das zu entwickelnde System zugegriffen wird. Diese Daten werden aber nicht in diesem System gepflegt, sondern durch ein Fremdsystem. Wurden die Function Points für diese Daten bereits in dem Fremdsystem gezählt, so können diese übernommen werden.

Die Funktionspunkte werden ermittelt, indem man alle Transaktionen (Funktionen) und so alle Datenelemente des Softwaresystems ermittelt. Anschließend ordnet man diesen Datenelementen Komplexitätsgrade (einfach, mittel, komplex) gemäß der folgenden Tabellen („Matrizen der Funktionskomplexitäten" – *function complexity matrices*)[6] zu:

Eingaben	**Anzahl Datenelemente**		
Datentypen	< 5	**5 – 15**	> 15
$= 1$	einfach	einfach	mittel
$= 2$	einfach	mittel	komplex
> 2	mittel	komplex	komplex

Ausgaben & Abfragen	**Anzahl Datenelemente**		
Datentypen	< 6	**6 – 19**	> 19
$= 1$	einfach	einfach	mittel
2 – 3	einfach	mittel	komplex
> 3	mittel	komplex	komplex

Datenbestände & Schnittstellendaten	**Anzahl Datenelemente**		
Datentypen	< 20	**20 – 50**	> 50
$= 1$	einfach	einfach	mittel
2 – 5	einfach	mittel	komplex
> 5	mittel	komplex	komplex

[6] David Longstreet: *Function Points Analysis Training Course.* Online-Tutorial *http://www.softwaremetrics.com/freemanual.htm* (Letzter Abruf: 16.2.2008)

Die Addition der einzelnen Funktionspunkte ergibt dann die ungewichteten Funktionspunkte *(unadjusted function points)* insgesamt des Softwaresystems[6]:

Kategorie	**Komplexität**			**Gesamt**
	einfach	**mittel**	**komplex**	
Eingaben	$\_ \cdot 3 = \_$	$\_ \cdot 4 = \_$	$\_ \cdot 6 = \_$	
Ausgaben	$\_ \cdot 4 = \_$	$\_ \cdot 5 = \_$	$\_ \cdot 7 = \_$	
Abfragen	$\_ \cdot 3 = \_$	$\_ \cdot 4 = \_$	$\_ \cdot 6 = \_$	
Datenbestände	$\_ \cdot 7 = \_$	$\_ \cdot 10 = \_$	$\_ \cdot 15 = \_$	
Schnittstellen	$\_ \cdot 5 = \_$	$\_ \cdot 7 = \_$	$\_ \cdot 10 = \_$	
	Ungewichtete Funktionspunkte			

Als letztes gewichtet man diese mit Bewertungsfaktoren, die technische Einflussfaktoren widerspiegeln. Insgesamt ergibt sich damit das Bewertungsformular der Funktionspunktanalye in Abb. 6.4.

Kategorie	**Anzahl**	**Komplexität**	**Faktor**	**Funktionspunkte**
Eingaben	x_1	einfach	3	$3 \cdot x_1$
	x_2	mittel	4	$4 \cdot x_2$
	...	komplex	6	...
Ausgaben		einfach	4	
		mittel	5	
		komplex	7	
Abfragen		einfach	3	
		mittel	4	
		komplex	6	
Datenbestände		einfach	7	
		mittel	10	
		komplex	15	
Schnittstellen		einfach	5	
		mittel	7	
		komplex	10	
Ungewichtete Funktionspunkte p_u				$p_u = \sum$
Einflussfaktoren	Verflechtungen mit anderen Softwaresystemen (0 – 5)			
	dezentrale Daten, dezentrale Verarbeitung (0 – 5)			
	Anzahl Transaktionen der Anwender (0 – 5)			
	Verarbeitungslogik a) Rechenoperationen (0–10) b) Kontrollverfahren (0–5) c) Ausnahmeregelungen (0–5) d) Logik (0–5)			
	Wiederverwendbarkeit (0 – 5)			
	Konvertierungen des Datenbestands (0 – 5)			
	Anpassbarkeit (0 – 5)			
Summe der Einflussfaktoren				$f = \sum$
bewerteter Einflussfaktor $w = f/100 + 0,7$				w
bewertete Funktionspunkte $p = w \cdot p_u$				p

Abbildung 6.4: Bewertungsformular der Funktionspunktanalyse (nach: *http://danae.uni-muenster.de/~lux/seminar/ss01/Michaelsen.pdf*

Kapitel 7

Softwarequalität

7.1 Qualität und Anforderungen

Wie von jedem anderen Produkt auch erwartet und benötigt der Anwender von Software eine „Qualität“. Dieser Begriff ist allerdings sehr vage. Softwarequalität muss genauer definiert werden und sollte idealerweise quantifizierbar sein, um sie zu messen und bestimmen zu können, ob sie in ausreichendem Maße vorhanden ist oder nicht.

Definition 7.1. *Softwarequalität*, oft auch „äußere Qualität“ oder „fachliche Qualität“ [4], ist die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, bezüglich ihrer Eignung, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen [2]. Insbesondere umfasst sie die Korrektheit, die Zuverlässigkeit und die Bedienbarkeit von Software. □

Wie *Qualität* allgemein hat Softwarequalität also einen subjektiven Wert für den Anwender. Üblicherweise denkt man an Softwarequalität im Alltag nur, wenn etwas mit ihr *nicht* stimmt. Hat man beispielsweise gerade eine Onlineüberweisung abgeschickt und bleibt das Programm plötzlich stecken und reagiert nicht mehr, was ist zu tun? Ist das Geld überwiesen? Wenn nicht, steht eine Mahnung ins Haus, also nochmal überweisen; wenn die Überweisung beim ersten Mal dann aber doch geklappt hat, kommt es zu einer Doppelüberweisung, überzogenem Konto und hohen Zinsen. In diesem Beispiel würde man unter Softwarequalität nicht die Verhinderung von Verbindungsproblemen verstehen, sondern die schnelle und zuverlässige Überprüfbarkeit einer Überweisung.

Zur „Erfüllung der Erfordernisse“ in Definition 7.1 müssen die Erfordernisse der Software, oder auch die Erwartungen an sie, klar definiert sein. Erst dann können sie im Erstellungsprozess nicht falsch verstanden werden und permanent überprüft und gemessen werden.

Mit den „festgelegten“ Erfordernissen ist das schon schwer genug und gelingt nur mit genügender Sorgfalt bei Analyse und Entwurf. Ein weit größeres Problem sind die „vorausgesetzten“ Erfordernisse, hier gibt es die schwerwiegendsten Missverständnisse. Denn was der eine stillschweigend voraussetzt, muss der andere noch lange nicht wissen. Wer sich beispielsweise eine Uhr implementieren lässt, wird kaum explizit fordern, dass sie nach 23:59 Uhr wieder mit 0:00 Uhr beginnt. D.h. ein gewisses Grundverständnis muss man voraussetzen. Aber wo endet es genau? Vermeintliche Selbstverständlichkeiten sind die häufigsten Fallen bei der Anforderungsermittlung.

Umgekehrt beziehen sich viele Anforderungen auf die Qualität von Software und Systemen. Bei Anforderungen denkt man zumeist an die *funktionalen Anforderungen* an die Software, die beschreiben, welche Funktionen sie haben soll. Qualitätsanforderungen aber, die festlegen, wie und in welcher Qualität die Software laufen soll, sind abder ebenso wichtig, auf die Architektur eines Systems haben sie sogar oft größeren Einfluss als funktionale An-

forderungen. Qualitätsanforderungen sind aus Qualitätszielen aufgebaut, die sich ihrerseits auf angestrebte Qualitätsmerkmale beziehen:

> *Qualitätsmerkmale*, oder *Qualitätsaspekte*, sind einzelne Eigenschaften einer Einheit, anhand derer die Qualität beschrieben und beurteilt wird.

$\Downarrow$

> Ein *Qualitätsziel* ist ein *angestrebtes* Merkmal.

$\Downarrow$

> Eine *Qualitätsanforderung* ist eine Menge von zusammengehörigen Qualitätszielen.

Wenn man die gewünschte Ausprägung von Qualitätsmerkmalen angeben möchte, benötigt man *Qualitätsmetriken*. Das sind Maße (Wertebereiche von-bis) oder Indikatoren („trifft zu“ oder „trifft nicht zu“) für bestimmte Qualitätsmerkmale.

Beispielsweise besteht die Qualitätsanforderung „hohe Verfügbarkeit des Rechnersystems“ aus den Qualitätszielen Robustheit bei vielen Internetzugriffen, ausreichende Geschwindigkeit bei hoher Netzlast und einfachen Benutzeroberflächen, d.h. Robustheit, Geschwindigkeit und Gestaltung der Benutzeroberflächen sind die Qualitätsmerkmale. Die entsprechenden Qualitätsmetriken könnten dann die Anzahl paralleler Benutzer, gegen die das System robust sein soll, die Anzahl der Operationen pro Sekunde für die Geschwindigkeit.

Um Qualitätsanforderungen vernünftig zu erheben und zu prüfen, sind wie in anderen Bereichen des Qualitätsmanagements Wokshops und Mitarbeiterbefragungen, Abstimmungsrunden und Validierungen erforderlich. Man kommt dabei oft relativ schnell zu allgemeinen vagen Qualitätsanforderungen, der schwierigste Teil der Arbeit besteht darin, diese auf Qualitätsmerkmale und Qualitätsziele und schließlich auf ihre Qualitätmetriken zu konkretisieren. Meist versucht man schrittweise dahin zu gelangen mit Hilfe eines sogenannten *Qualitätsmodells*, das modelliert, was der Kunde unter den Qualitätzielen konkret versteht [9, §2.7]. Normen und Standards können dabei sehr helfen, in ihnen sind Merkmale einheitlich und in der Regel eindeutig definiert.

7.2 Kosten und Nutzen von Softwarequalität

Qualitätsmaßnahmen müssen sich lohnen. Sie werden letztendlich eingesetzt, um die eigenen Kosten zu sparen. Allerdings kosten sie selber auch, und man muss kaufmännisch gegenrechnen, inwieweit sich die Gesamtsumme aus Fehlervermeidungskosten, also den Ausgaben für Qualitätsmaßnahmen, und den Fehlerkosten, also den Ausgaben für Fehlerbeseitigung, Haftung oder Entschädigung minimieren lässt. Qualitativ kann man in der Realität ein Kurvendiagramm wie in Abb.7.1 beobachten, in dem die Fehlerkosten bei steigendem Qualitätsaufwand (gemessen z.B. in Arbeitszeit oder Ausgaben) sinken, während die Fehlervermeidungskosten nahezu linear mit ihm steigen. Entsprechend gibt es ein globales Minimum der Kurve der Qualitätskosten insgesamt, der den optimalen Qualitätsaufwand bestimmt.

Eine fundamentale Eigenschaft von Qualität offenbart sich in dem harmlosen Diagramm: *Perfekte Qualität gibt es nur mit enorm hohem Qualitätsaufwand.* Im Grenzfall mit unendlichem Aufwand, also nie. Wie in jedem Markt gibt es eine *win-win*-Situation zwischen Nachfrager und Anbieter, wenn der Nachfrager für seinen Preis das ihm erforderliche Maximum

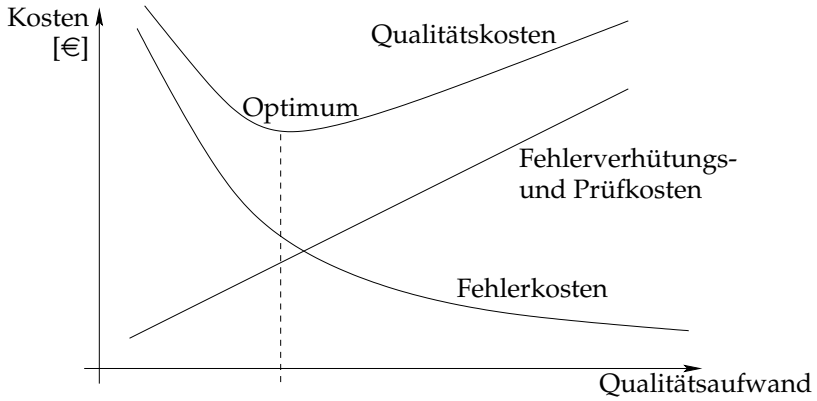

Abbildung 7.1: Optimierung der Qualitätsaufwände. Die Fehlerkosten umfassen beispielsweise Kosten für Rückrufaktionen oder Reparaturen.

an Qualität erlangt und der Anbieter dafür ein Minimum an Qualitätskosten hat. Entsprechend kann das Ziel einer Qualitätssicherung nicht sein, die Qualität auf perfektes Niveau zu bringen, sondern die vom Nachfrager erwartete Qualität zu garantieren.

7.3 Systematisches Testen

Die Qualitätssicherung von Software bedient sich einiger grundlegender Verfahren. Dazu gehört die *Usability*, also die Prüfung der Bedienbarkeit und der Gebrauchstauglichkeit von Software, beispiesweise die Abstimmung der Bedienabläufe und die Gestaltung des gesamten Programms, nicht nur der Bedienoberfläche, auf die Anwender und ihre Aufgaben. Weitere wichtige Verfahren der Qualitätssicherung sind das Testen und Reviews.

Definition 7.2. *Testen* ist die Ausführung von Software mit dem Ziel, Fehler zu finden. □

Diese Definition impliziert, dass durch Testen möglichst viele, schwere Fehler zu finden sind. Ein *Abnahmetest*, also eine Menge zwischen Auftraggeber und Auftragnehmer vereinbarter Prüffälle, denen die Software bei der Übergabe unterzogen wird, gehört damit in diesem Sinne gar nicht zum Testen! Die Zielsetzung eines Abnahmetests ist es nämlich, tatsächlich festzustellen, *dass die Software das Vereinbarte leistet*, also genau entgegengesetzt zur Absicht des Testen. Das Testen entspricht der im Kern sehr ernsten Grundeinstellung von „Murphys Gesetz“, wonach alles, was schiefgehen *kann*, irgendwann auch schiefgeht. Ein noch verborgener Fehler in einer Software wird irgendwann auftreten.

Wenn ein Fehler gefunden wurde, weiß man dadurch noch nicht, wie er zustande kam. Man wird seine Ursache also *nach* dem Testen suchen. Findet man dagegen keinen Fehler, so kann das zweierlei bedeuten: Entweder die Software enthält tatsächlich nur wenige oder keine Fehler, oder der Test war mangelhaft aufgesetzt, so dass er keinen der vorhandenen Fehler gefunden hat. Um die zweite, unangenehme, Ursache auszuschließen, sollte man *systematisch* testen. So werden Fehler zumindest nicht leichtfertig übersehen.

Herumprobieren gilt daher nicht als Test, es kann höchstens die Vorbereitung für einen richtigen Test sein. Für einen systematischen Test kommt es sogar darauf an, sich *schriftlich* vorzubereiten und die Testfälle *schriftlich* zu dokumentieren. Nur so kann man vermeiden, von einem plausiblen, aber trotzdem falschen Ergebnis geblendet zu werden.

7.3.1 Fehler

Aber was ist eigentlich ein *Fehler*? Ein Softwarefehler liegt immer dann vor, wenn sich die Software falsch verhält. Falsch oder richtig bezieht sich in einem Softwareprojekt immer

darauf, was in der Spezifikation steht. Das ist hoffentlich ziemlich genau das, was der Kunde möchte und auch braucht. Aber dessen Gedanken können ungenau sein oder sich ändern.

Definition 7.3. Ein *Fehler* einer Software ist die Abweichung der Software von dem in der Spezifikation vorgeschriebenen Verhalten. □

Folglich gilt automatisch: *Keine Spezifikation – keine Fehler!*

Stellt man beim Testen fest, dass ein Resultat („Ist“) nicht mit dem erwarteten Wert („Soll“) übereinstimmt, muss nicht notwendigerweise ein Softwarefehler der Grund sein. Die theoretisch möglichen Ursachen lauten:

Ist $\neq$ Soll	
Sollwert	**Ursache**
falsch	Kein Sollwert dokumentiert
	Anforderung falsch verstanden
	Sollwert falsch ermittelt
korrekt	Vergleich unangemessen
	Istwert falsch

Nur im letzten Fall liegt definitiv ein Programmierfehler vor, bei allen anderen Fällen könnte die Software sogar korrekt sein, der Fehler liegt in den Analyseschritten vor der Implementierung. Wenn eine Anforderung beim Erstellen der Testfälle falsch verstanden wurde, beim Programmieren aber richtig, kommt es zu einer Abweichung: Die Software ist in Ordnung, der Testfall ist falsch. Gleiches gilt für den Fall, dass zwar die Anforderung richtig verstanden wurde, aber ein Rechenfehler bei der Bestimmung des Sollwerts passierte. Es kommt manchmal auch vor, dass der Vergleich nicht angemessen oder zu streng ist. Wenn komplizierte numerische Algorithmen mit erheblichen Rundungen rechnen, kann es sinnvoller sein, statt eines Sollwerts einen Sollwertbereich anzugeben. Das gilt beispielsweise bei Steuerungen und Regelungen, die von Eingangsgrößen abhängen können, die ihrerseits kaum präzise einzustellen sind (z.B. „eine Spannung von 3,0 Volt $\pm$ 5%“).

Natürlich kann es auch vorkommen, dass sowohl Ist- als auch Sollwert falsch sind, im schlimmsten Fall sogar denselben, falschen Wert annehmen. Das kann passieren, wenn man bei der Programmierung denselben Denkfehler macht wie bei der Erstellung. Gegen diesen Effekt kann helfen, mehrere unabhängige Tester die Testfälle erstelen zu lassen, so dass die Wahrscheinlichkeit derselben Denkfehler minimiert wird.

7.3.2 Erstellen von Testfällen: *Black-Box* und *Glass-Box*

Ein *Testfall* beschreibt, welche Ergebnisse die Software bei einer spezifizerten Eingabe produziert. Man kann Testfälle also eindeutig in einer Tabelle mit Spalten für die Eingaben und für die erwarteten Ergebnisse oder Reaktionen definieren. Sehr verbreitet bei der Entwicklung mit Java ist die Erfassung der Testfälle in ausführbarem Quelltext durch das Open-Source Testframework JUnit.

Man braucht in der Regel ziemlich viele Testfälle, je nach Systemgröße von einigen Dutzenden bis zu tausenden. Testfälle zu erstellen ist die schwierigste, aber auch interessanteste Aufgabe der Testvorbereitung. Man unterscheidet grob zwei Ansätze.

Beim *Black-Box-Test* nimmt man sich die Spezifikation vor und leitet daraus Testfälle ab. In der Spezifikation stehen alle dokumentierten Anforderungen, daher ist sie der Bezugspunkt für die Tests, Motto „Für jede Anforderung mindestens einen Testfall“. Man interessiert sich bei diesem Ansatz nur dafür, wie sich das System laut Spezifikation verhalten soll, nicht aber dafür, wie es intern aufgebaut ist. Man betrachtet das System also als Black Box, deren Inneres man nicht sehen kann. Entsprechend kann man in den Testfällen nur das abdecken, was man ohne Kenntnis des Innern weiß.

Der zweite Ansatz, der *Glass-Box-Test*, oder *White-Box-Test*, der sich an der Struktur der Software orientiert, Motto „Die Testfälle sollen den Quelltext überdecken“.[1] Wo gibt es schwierige oder komplizierte Stellen im Quelltext, an denen leicht Fehler auftreten können? Wurden mit den bisherigen Testfällen alle inneren Kombinationsmöglichkeiten und Abläufe geprüft? Insbesondere bei objektorientierten Sprachen, in denen Objekte oder Klassen andere Objekte oder Klassen benutzen, gilt die Regel: *„Stets das Benutzte vor dem Benutzenden testen.“* D.h. bei Assoziationen und Kompositionen wird die verwendete vor der verwendenden Klasse getestet, bei Vererbungshierarchien die Superklasse vor der erbenden Klasse [9, S. 108f].

Beide Ansätze, Black-Box und Glass-Box, sind systematische Ansätz mit ihren eigenen Vor- und Nachteilen, die sich einander ergänzen, aber nicht ersetzen.

[1] Testfälle sollen also die Anforderungen *ab*decken und den Quelltext *über*decken.

Literaturverzeichnis

[1] BALZERT, H. : *Lehrbuch der Software-Technik. Software-Entwicklung*. Heidelberg Berlin : Spektrum Akademischer Verlag, 1996

[2] BALZERT, H. : *Lehrbuch der Software-Technik. Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Heidelberg Berlin : Spektrum Akademischer Verlag, 1998

[3] BOEHM, B. : 'A spiral model of software development and enhancement'. In: *ACM SIGSOFT* (1986), S. 14–24

[4] FLEISCHER, A. : 'Ist Qualität messbar?'. In: *Java Magazin* 5 (2008), S. 99–104

[5] HAREL, D. ; FELDMAN, Y. : *Algorithmik. Die Kunst des Rechnens*. Berlin Heidelberg : Springer-Verlag, 2006

[6] HINDEL, B. ; HÖRMANN, K. ; MÜLLER, M. ; SCHMIED, J. : *Basiswissen Software-Projektmanagement*. Heidelberg : dpunkt verlag, 2009

[7] RAYMOND, E. S.: 'The cathedral and the bazaar'. In: *First Monday* 3 (1998), Nr. 3. – http://www.uic.edu/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/578/499

[8] ROBLES, G. : 'A Software Engineering approach to Libre Software'. In: GEHRING, R. A. (Hrsg.) ; LUTTERBECK, B. (Hrsg.): *Open Source Jahrbuch 2004 – Zwischen Softwareentwicklung und Gesellschaftsmodell*. Berlin : Lehmanns Media LOB.de, 2004. – ISBN 978–3–93642–778–3, S. 193–208. – http://www.opensourcejahrbuch.de/2004/pdfs/III-3-Robles.pdf

[9] SCHNEIDER, K. : *Abenteuer Software Qualität. Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement*. 1. Heidelberg : dpunkt.verlag, 2007

[10] SEEMAN, J. ; VON GUDENBERG, J. : *Software-Entwurf mit UML 2*. 2. Berlin Heidelberg : Springer-Verlag, 2006. – *http://dx.doi.org/10.1007/3-540-30950-0*

[11] STALLMAN, R. : Why "Open Source" Misses the Point of Free Software. In: LUTTERBECK, B. (Hrsg.) ; BÄRWOLFF, M. (Hrsg.) ; GEHRING, R. A. (Hrsg.): *Open Source Jahrbuch 2007 – Zwischen freier Software und Gesellschaftsmodell*. Berlin : Lehmanns Media LOB.de, 2007, S. 1–7. – http://www.opensourcejahrbuch.de/portal/article_show?article=osjb2007-00-02-en-stallman.pdf

[12] WINTER, M. : *Methodische objektorientierte Softwareentwicklung. Eine Integration klassischer und moderner Entwicklungskonzepte*. Heidelberg : dpunkt.verlag, 2005

Web-Links

[moz] *http://wiki.mozilla.org/ReleaseRoadmap* – Release Roadmap des Mozilla-Projektes [3.3.2008]

[swebok] *http://www.swebok.org/ – SWEBOK, Guide to the Software Engineering Body of Knowledge. 2004 Version*. Handbuch des Gesammelten Wissens zum Software-Engineering der IEEE Computer Society [3.3.2008]

[SVN] *http://svnbook.red-bean.com/* – Online-Version von Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato: *Version Control with Subversion.* O'Reilly (2007) [3.3.2008]

[tigris] *http://tigris.org* – Tigris, eine Open-Source Community, das sich der Entwicklung und Bereitstellung von Tools des kollaborativen Software-Engineering widmet [3.3.2008]

Index

äußere Qualität, 48

Abnahmetest, 50
agile Softwareentwicklung, 22
agile Systementwicklung, 25
Aktivität, 11
Analyse, 12
Analysemetrik, 44
Anforderungen, 48
Anforderungsdefinition, 11, 19
Anforderungsspezifikation, 12
angebotsorientiert, 11
Application Engineering, iv

Basarmodell, 20
Bedarfserfassung, 11
Black-Box-Test, 51
Branching, 35

CBO, 44
CCN, 42
Ce, 44
CVS, 35

Daily Scrum, 16
Delphi-Methode, 21
DIT, 43
DOC, 41
dokumentorientiertes Vorgehensmodell, 13
DRY, 44

eingebettete Systeme, v
Entscheidungspunkt, 25
Entwurf, 12, 19
Erweiterung (Software), 33
eXtreme Programmierung, 23

fachliche Qualität, 48
Fehler, 51
Fiskus (Softwareprojekt), 8
Fork, 35
freie Software, 38
Funktionspunktanalyse, 45

Geschäftsprozessmodellierung, 18
Glass-Box-Test, 52

Halstead Software Science, 44
HEAD, 35

Implementierung, 12, 19
Inbetriebnahme, 19
Individualsoftware, iv
innere Qualität, 40
Integration, 12

JUnit, 51

KISS, 28
kollaborative Softwareentwicklung, 34

LCOM, 44
Lines of Code, 40
Linux, 22
LOC, 40

McCabe-Metrik, 42
Meilenstein, 11, 25
Merge, 34
modifizierte Wasserfallmodell, 14
Modul (CVS), 35
Modularität, 32
Moment (Psychologie), 45
Murphys Gesetz, 50

NCSS, 41
NOC, 43

Objektmetrik, 43
otimistisches Team-Modell, 36

Paarprogrammierung, 23
Patch, 12
Phase, 11
Plug-In, 33
Produkt, 25
Projekt, 13
Projektdurchführungsstrategien, 25
Projektmanagement, 15
Prototyp, 13

Qualität, 48
Qualität, technische -, 40
Qualitätsmetrik, 49
Qualitätsmodell, 49
Qualitätssicherung, 15
Quantitätsmetrik, 40, 41

Rational Unified Process, 18
Refactoring, 23
Release, 23
Repository (CVS), 35
Resource (CVS), 37
Revision, 38
RFC, 44
RUP, 18

Scrum, 15

SLOC, 40
Software, 9
Softwaremetrik, 40
Softwarequalität, 48
Softwarespezifikation, 19
Source Lines of Code, 40
Sprint, 16
Standardsoftware, iv
Story, 15, 24
Stroud-Zahl, 45
STY, 44
Subversion, 38
SVN, 38
Synchronize, 36
Systemsoftware, v

tailoring, 25
technische Qualität, 40
Test, 19
Testen, 50
Testfall, 15, 51
Trunk, 35

Unicode, 30
Usability, iv
User Story, 15

V-Modell, 15
V-Modell XT, 25
Validierung, 15
Verifikation, 15
Version (in CVS), 37
Version Tag (CVS), 37
Vorgehensbaustein, 25
Vorgehensmodell, 11

Wasserfallmodell, 13
White-Box-Test, 52
Wiederverwendung, 19
Wissensmamagement, 15
WMC, 43
Workbench, 36
Workspace, 36

XML, 30
XP, 23

zyklomatische Komplexität, 42