Konzeption und prototypische Implementierung eines Frameworks zur automatisierten Softwaremessung

Von der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von

Bernhard Daubner

geboren in München

1. Gutachter: Prof. Dr. Andreas Henrich

2. Gutachter: Prof. Dr. Bernhard Westfechtel

Tag der Einreichung: 05.03.2008

Tag des Kolloquiums: 09.07.2008

Meinem Vater
Hermann Daubner (⋆1940 †1980)
gewidmet

Conceptual Design and prototypical Implementation of a Framework for automated Software Measurement

Abstract

Within corporate management so called *Management Cockpits* or *Management Support Systems* are well known. These IT systems are designed for the visualisation of decision-relevant information. They are able to provide company management with key performance indicators about the company's operations at a click.

Analogously, *project cockpits* are meant to provide project managers with performance indicators about ongoing IT projects. Examples are test coverage, completion rate, code quality or spent effort. The measurement tools should be able to record these software measures automatically.

Available software measurement applications however either strongly depend on certain process models or development environments, can not perform measures automatically but only visualize manually entered information or can only be configured or extended by the user in a limited way.

Within this work a *framework for automated software measurement* is presented. Its main features are the large degree of measurement automation and the possibility, to define the software measures to collect independently of a concrete project. For this purpose context information is used to create anchor points for software measures. A context can be a certain activity of the underlying process model, a project phase or a functionality that has to be implemented. Thus it is possible, for instance, to restrict size measurements on certain project phases or on components that provide a certain functionality.

Many measurement tools only allow to measure already completed artefacts or only can collect software measures that concern the whole project. In contrast to these tools the approach presented here makes it possible to define software measures with respect to a certain context. The entities that actually have to be measured need not to be known at the time the software measure is defined. Thus software measures can be collected in a standardized way across project borders.

As prototypical implementation of the software measurement approach a framework is presented that is based on an Open-Source project management tool. The framework, which provides API functions to gather simple software measures, allows the flexible definition of complex software measures that are computed automatically at project's runtime and are visualized where appropriate. The definition of the software measures to collect is done by means of the underlying project management tool. Thus already maintained project information can be reused.

In order to proof the potential of this approach the measurement tool is evaluated within the scope of two student software development projects.

Kurzfassung

Im Rahmen der betriebswirtschaftlichen Unternehmenssteuerung sind sogenannte *Management-Cockpits* oder *Managementunterstützungssysteme* schon länger bekannt. Dabei handelt es sich um EDV-Systeme zur Visualisierung entscheidungsrelevanter Daten, welche insbesondere der Unternehmensleitung auf Knopfdruck wesentliche Kenngrößen (*Key Performance Indicators*) des Unternehmens darstellen.

Analog dazu sollen *Projektleitstände* den Projektleitern entsprechende Kenngrößen der laufenden IT-Projekte darstellen. Derartige Kennzahlen sind beispielsweise die Testabdeckung, der Fertigstellungsgrad, die Code-Qualität oder der bereits erbrachte Arbeitsaufwand. Diese Softwaremaße sollten dazu von den Messwerkzeugen automatisiert erfasst werden können.

Aktuell verfügbare Anwendungen zur Softwaremessung sind jedoch entweder stark an bestimmte Prozessmodelle oder Entwicklungsumgebungen gebunden, können teilweise selbst keine Messwerte ermitteln und dienen somit als reine Visualisierungswerkzeuge oder können nur eingeschränkt von den Anwendern konfiguriert oder erweitert werden.

Im Rahmen dieser Arbeit wird daher ein *Framework zur automatisierten Softwaremessung* vorgestellt. Zu dessen wesentlichen Anforderungen gehören die weitgehende Automatisierung des Messprozesses und die Möglichkeit, die zu erhebenden Softwaremaße unabhängig von einem konkreten Projekt definieren zu können. Dazu werden Kontextinformationen bestimmt, welche als Anknüpfungspunkte für die zu erhebenden Softwaremaße dienen. Ein Kontext kann dabei eine bestimmte Aktivität des zugrunde liegenden Vorgehensmodells, eine Projektphase oder eine zu implementierende Funktionalität sein. Dadurch ist es beispielsweise möglich, Umfangsmessungen auf bestimmte Projektphasen oder auf Komponenten, durch die bestimmte Funktionalitäten implementiert werden, einzuschränken.

Im Gegensatz zu vielen anderen Messwerkzeugen, die entweder nur die Vermessung von bereits erstellten Artefakten erlauben oder Softwaremaße nur auf der Ebene des Gesamtprojekts ermitteln können, ermöglicht es der hier dargestellte Ansatz, Softwaremaße in Bezug auf einen bestimmten Kontext zu definieren. Die konkrete Ausprägung der letztendlich zu messenden Entitäten muss zum Zeitpunkt der Definition des Softwaremaßes noch nicht bekannt sein. Dadurch kann erreicht werden, dass Softwaremaße standardisiert und über Projektgrenzen hinweg konsistent ermittelt werden können.

Als prototypische Implementierung dieses Softwaremessungsansatzes wird ein Framework dargestellt, welches auf einer Open-Source-Projektverwaltungssoftware basiert. Dieses Framework, welches entsprechende API-Funktionen zur Ermittlung einfacher Softwaremaße zur Verfügung stellt, erlaubt die flexible Definition komplexer

Softwaremaße, welche dann zur Projektlaufzeit automatisiert ermittelt und gegebenenfalls visualisiert werden können. Zur Definition der zu erhebenden Softwaremaße wird auf die Konfigurationsmöglichkeiten der zugrunde liegenden Projektverwaltungssoftware zurückgegriffen. Dadurch können bei der Softwaremessung bereits vorhandene Projektinformationen wiederverwendet werden.

Um die Tragfähigkeit des in dieser Arbeit dargestellten Ansatzes unter Beweis zu stellen, wird die Evaluierung dieses Messwerkzeuges anhand zweier studentischer Softwareentwicklungsprojekte dargestellt.

Vorwort

Im Prinzip erwies sich die wirtschaftliche Schieflage, in welche die SchmidtBank KGaA in Hof als mein ehemaliger Arbeitgeber im Jahr 2001 geraten war, als Auslöser für mich, ein Promotionsvorhaben zu beginnen. Ich stand damals nach mehreren Jahren Berufstätigkeit im Banken-IT-Umfeld vor der Wahl, einen unsicher gewordenen Arbeitsplatz gegen ein Projekt mit unbekanntem Ziel und noch weniger bekannten Weg dorthin einzutauschen.

Herr Prof. Dr. Andreas Henrich, der damals den neu gegründeten Lehrstuhl für Softwaretechnik an der Universität Bayreuth innehatte, hat mich in persönlichen Gesprächen und per eMail ermutigt, diesen Weg zu gehen. An dieser Stelle möchte ich ihm nicht nur für die Motivation und die Betreuung dieser Arbeit als Doktorvater meinen Dank aussprechen, sondern auch dafür, dass er es mir durch die Anstellung als wissenschaftlichen Mitarbeiter an seinem Lehrstuhl überhaupt erst ermöglicht hat, ein Promotionsvorhaben anzugehen. Auch wenn das Thema eigentlich von Anfang an im Bereich der Werkzeugunterstützung für die Softwaremessung angesiedelt war, lies mir mein Doktorvater die Freiheit, mich zunächst ausgiebig mit benachbarten Themen, wie dem Requirements-Engineering oder der Fortschrittsmessung von SAP-R/3-Projekten zu beschäftigen, um dann nach ca. zwei Jahren doch noch zum letztendlichen Thema dieser Arbeit vorzudringen.

Genauso möchte ich mich bei Herrn Prof. Dr. Bernhard Westfechtel nicht nur für die Übernahme des Zweitgutachtens auf das herzlichste bedanken, sondern auch dafür, dass er sich als Betreuer vor Ort geduldig Zeit genommen hat, um mit mir Hindernisse und neue Aspekte zu diskutieren, aber auch, um kritische Anmerkungen und wertvolle Anregungen zu machen. Auch am Lehrstuhl von Prof. Westfechtel fand ich ein anregendes und harmonisches Arbeitsklima vor, welches es mir ermöglichte, meine Doktorarbeit erfolgreich abzuschließen.

Einen wesentlichen Anteil an dem fruchtbaren Arbeitsumfeld hatten auch meine zahlreichen Lehrstuhlkollegen, welche auf die Standorte Bayreuth und Bamberg verteilt waren. Namentlich möchte ich hier insbesondere Sabrina Uhrig, Thomas Buchmann, Alexander Dotor, Martin Eisenhardt, Volker Lüdecke, Dr. Karlheinz Morgenroth, Dr. Wolfang Müller und Dr. Günter Robbert erwähnen. Darüber hinaus danke ich auch den weiteren Lehrstuhlmitgliedern Monika Glaser und Bernd Schlesier, die als gute Seelen im Hintergrund für einen reibungslosen Betrieb des Lehrstuhls und seiner Infrastruktur sorgten.

Zum Schluss möchte ich mich noch sehr bei meiner Ehefrau Heike bedanken, welche die nicht beneidenswerte Aufgabe übernommen hat, diese Arbeit auf typographische Fehler und unklare Formulierungen durchzusehen.

Hof, im Juli 2008 — Bernhard Daubner

Die meisten Produktbezeichnungen von Hard- und Software sowie von Firmennamen, die in dieser Arbeit genannt werden, sind in der Regel auch eingetragene Warenzeichen. Der Autor folgt bei den Produktbezeichnungen im Wesentlichen den Schreibweisen der Hersteller.

Verweise auf Online-Referenzen beziehen sich auf den Stand vom 29.02.2008.

Inhaltsverzeichnis

1 Einleitung

> It's supposed to be automatic, but actually you have to push this button.
>
> *(John Brunner)*

1.1 Problemstellung und Motivation

Softwaremessung als Schlüsseltechnologie

Software-Messverfahren und Softwaremaße werden als Schlüsseltechnologien für das Controlling und das Management von Softwareentwicklungsprozessen betrachtet. Durch Messen im Bereich des Software Engineerings kann ein besseres und tieferes Verständnis der Entwicklungsprozesse und der dabei erstellten Produkte erlangt werden (vgl. [ABBD06], S. vii). Insbesondere wird Softwaremessung als Voraussetzung für die Beurteilung der Softwarequalität und die Vorausberechenbarkeit von Softwareprojekten gesehen (vgl. [Zus98]). Diese beiden letztgenannten Aspekte sind insbesondere bei der kommerziellen Softwareentwicklung von großer Bedeutung.

Gestaltung von IT-Verträgen

Sowohl für Softwareunternehmen als auch für unternehmensinterne IT-Abteilungen ist eine zumindest näherungsweise Vorausberechenbarkeit von IT-Projekten von essentieller Bedeutung bei der Vertragsgestaltung. Bei Verträgen über die Neu- oder Weiterentwicklung von Softwareprodukten sind in der Regel mehr oder weniger verbindliche Angaben zu Projektumfang, Projektdauer, Gesamtkosten und Qualität des auszuliefernden Softwareprodukts wichtige Vertragsbestandteile (vgl. [Zah05]). Nicht minder komplex erweist sich die Vertragsgestaltung bei Wartungsverträgen für Unternehmenssoftware zwischen IT-Systemhäusern und deren Kunden. Derartige Verträge umfassen beispielsweise das Betreiben einer Unternehmensanwendung (z. B. *SAP R/3*) im Rahmen des *Application Service Providing* durch einen Dienstleister, dem *Application Service Provider* (ASP, vgl. [Brä04]). Der ASP kümmert sich dabei nicht nur um Hosting, Betrieb und Administration der Software, sondern muss diese durch regelmäßiges Implementieren der vom Hersteller der Software zur Verfügung gestellten Software-Updates auf den aktuellen Stand halten. Da es dabei häufig zu Konflikten zwischen den Hersteller-Updates und eigenen Veränderungen oder Erweiterungen der ursprünglich ausgelieferten Software kommt, ist das Implementieren der Hersteller-Updates meistens mit umfangreichen Programmiertätigkeiten verbunden (vgl. [Nie07]).

Vorausberechenbarkeit von IT-Projekten

Diese Beispiele verdeutlichen die Notwendigkeit für Softwareunternehmen, zu denen aufgrund der obigen Ausführungen auch Application Service Provider und sonstige IT-Systemhäuser gehören, zum einen den Umfang und die Kosten von Softwareentwicklungen möglichst genau vorhersagen und zum anderen Abweichungen im Bereich der geplanten Kosten, Termine und Qualitätsanforderungen zeitnah erkennen

zu können. Für die Aufwandsschätzungen wiederum sind, neben den Anforderungen an die zu entwickelnde Software, genaue Kenntnisse über die eingesetzten Softwareentwicklungsprozesse notwendig. Dabei ist sowohl in der Literatur als auch bei den operativ tätigen Praktikern akzeptiert, dass Softwaremessung sowohl taktisch im Rahmen der Planung, der Bewertung und des Controllings einzelner Softwareprojekte als auch strategisch zur Gewinnung von Erfahrungen mit den eingesetzten Entwicklungsprozessen und zur Prozessverbesserung erfolgreich eingesetzt werden kann.

Die in den Nachrichten vielfach gemeldeten Probleme bei der Umsetzung von IT-Projekten (z. B. LKW-Maut, Job-Portal der Bundesagentur für Arbeit, Bundeswehr-IT-Projekt *Herkules*) verdeutlichen jedoch, wie schwer es den Beteiligten fällt, die vertraglich vereinbarten Anforderungen an Umfang, Qualität und Kosten der IT-Projekte auch zu erfüllen. Dabei dringen üblicherweise nur die Probleme der populärsten IT-Projekte, welche auch für das Allgemeininteresse von Bedeutung sind, ans Licht der Öffentlichkeit.

Die Ursachen für den häufig unbefriedigenden Verlauf oder gar das Scheitern von Softwareentwicklungsprojekten sind vielfältig. Ein Grund ist sicherlich auch in der ungenügenden Anwendung von Software-Messverfahren zur Erkennung von Planabweichungen bei Softwareprojekten und zum Aufbau einer Wissens- und Erfahrungsbasis für die verwendeten Softwareprozesse zu suchen.

Softwaremessung als Kostenfaktor

Der Nutzen der Softwaremessung wird zwar auch von den meisten Praktikern grundsätzlich nicht in Frage gestellt, jedoch tragen sämliche Aktivitäten im Rahmen der Softwaremessung, wie auch alle übrigen Tätigkeiten im Rahmen des unternehmensweiten Controllings, erst einmal nichts zur Produktivität bei. Daher gehören Aktivitäten zur Softwaremessung zu den ersten Tätigkeiten, auf die verzichtet wird, falls sich innerhalb eines Projekts terminliche oder personelle Engpässe abzeichnen. Oder es werden zwar im Rahmen des Softwareentwicklungsprojektes Techniken zur Softwaremessung angewendet, dabei aber nicht die geeigneten Softwaremaße erhoben, die Messverfahren fehlerhaft durchgeführt oder die falschen Schlüsse aus den Messergebnissen gezogen.

1.2 Zielsetzung und Aufbau der Arbeit

Das Ziel der vorliegenden Arbeit ist daher, ein *Framework zur Softwaremessung* zu entwickeln, welches gerade den Praktikern die Anwendung von Software-Messverfahren erleichtern soll.

Automatisierung der Softwaremessung

Dies beinhaltet zunächst, die Softwaremessung so weit wie möglich zu automatisieren und damit gleichsam zu standardisieren. Das bedeutet, dass der operativ tätige Projektmitarbeiter so wenig wie möglich mit Routinetätigkeiten wie dem manuellen Führen von Prüf- und Messlisten belastet werden soll, um sich seinen eigentlichen Aufgaben widmen zu können. Stattdessen sollen durch dieses Framework vordefinierte Softwaremaße (z. B. Produktivitätsmaße) zusammen mit ihrem Kontext (z. B. die zugehörige Aktivität im Rahmen des Softwareentwicklungsprozesses) regelmäßig

Kontext der Softwaremessung

und automatisiert erhoben und die Ergebnisse geeignet aufbereitet werden. Zu diesem Zweck müssen vorher entsprechende Informationsbedürfnisse identifiziert und daraus die relevanten Softwaremaße abgeleitet werden.

Dabei sollen in dieser Arbeit weder neue Softwaremaße entwickelt noch gänzlich neue Verfahren zur Erhebung dieser Softwaremaße dargestellt werden. Das hier erarbeitete Framework soll stattdessen bewährte Ansätze der Softwaremessung miteinander verknüpfen und mittels Automatisierung eine Möglichkeit zur standardisierten und konsistenten Durchführung bereits bekannter Software-Messverfahren bieten.

Kombination bewährter Ansätze

1.2.1 Anforderungen an das Softwaremessungs-Framework

Konkret soll das zu entwickelnde Framework zur Softwaremessung die folgenden Anforderungen erfüllen:

- **Automatisierung der Messdurchführung**
  Die eigentliche Softwaremessung soll möglichst automatisiert erfolgen, so dass die Anwender (Projektmanager, Projektmitarbeiter) von entsprechenden Routine-Tätigkeiten entlastet werden. Für Projektinformationen, welche nicht maschinell erfasst werden können (z. B. Arbeitszeiten für bestimmte Tätigkeiten), sollen entsprechende Erfassungmasken zur Verfügung gestellt werden.

- **Dynamische Definition von Softwaremaßen**
  Softwaremaße sollen von den verantwortlichen Projektmitarbeitern dynamisch zur Projektlaufzeit definiert werden können. Dabei soll auch die Definition komplexer (d. h. mittels mathematischer Formeln verknüpfter) Softwaremaße möglich sein.

- **Wiederverwendung von Softwaremaßen**
  Einmal definierte Softwaremaße sollen auch in nachfolgenden Projekten wiederverwendet werden können. Daher muss die Erhebung der Softwaremaße in gleicher Weise erfolgen, um die Messergebnisse vergleichbar zu machen. Diese Anforderung impliziert auch, dass es möglich sein muss, die zu erhebenden Softwaremaße unabhängig von einem konkreten Projekt zu definieren.

- **Automatische Identifikation der zu messenden Entitäten**
  Aus den Forderungen nach der automatisierten Messdurchführung und der Wiederverwendbarkeit von Softwaremaßen folgt unmittelbar, dass das Framework in der Lage sein muss, die letztendlich zu vermessenden Entitäten maschinell zu identifizieren. Bei Softwaremaßen, die sinnvoll auf verschiedene Projekte angewendet werden können, muss dieser Identifikationsmechanismus auch entsprechend über Projektgrenzen hinweg funktionieren.

- **Visuelle Aufbereitung der Messergebnisse**
  Die Messergebnisse sollen visuell aufbereitet und in übersichtlicher Form dargestellt werden können.

- **Persistierung der Messergebnisse**
  Die Messergebnisse sollen dauerhaft gespeichert werden können, um bei der Durchführung von Messungen Vergleichsmöglichkeiten mit früheren Messungen zu haben.

- **Einfache Handhabung**
  Der Einarbeitungsaufwand in die Benutzung des Softwaremessungs-Frameworks sollte niedrig genug sein, um potentielle Anwender nicht abzuschrecken. Insbesondere sollen durch eine einfache Handhabbarkeit des Systems Softwareentwickler, die bis jetzt keine Softwaremessverfahren eingesetzt haben, ermutigt werden, Erfahrungen mit der Anwendung von Softwaremessungstechniken zu sammeln.

- **Integrierbarkeit in bestehende Umgebungen**
  Das Softwaremessungs-Framework sollte sich einfach in bestehende Entwicklungsumgebungen integrieren lassen, da es insbesondere bei großen Unternehmen wahrscheinlich ist, dass dort bereits umfangreiche Systeme zur Projektverwaltung und zur Softwareentwicklung im Einsatz sind.

- **Niedrige Anschaffungs- und Betriebskosten**
  Da, wie weiter oben gefordert, die Hemmschwelle zur Anwendung von Verfahren der Softwaremessung mittels des in dieser Arbeit entwickelten Softwaremessungs-Frameworks möglichst klein sein soll, sollten auch keine oder nur geringe Anschaffungs- und Betriebskosten anfallen. Diese Forderung lässt sich im Allgemeinen durch die Verwendung von Open-Source-Komponenten erfüllen.

- **Plattformunabhängigkeit**
  Schließlich sollte das Softwaremessungs-Framework auf möglichst vielen verschiedenen Betriebssystemplattformen einsetzbar sein. Dies ist insbesondere in den Fällen relevant, in denen das Framework in eine bestehende heterogene Systemlandschaft integriert werden soll.

Leichtgewichtiges Framework zur Softwaremessung

Die letzten vier Punkte können auch unter dem Schlagwort „Leichtgewichtigkeit“ subsumiert werden, weil die Anwender des Softwaremessungs-Frameworks nicht gezwungen werden sollen, ihre Prozesse zu ändern oder auf andere Entwicklungswerkzeuge zu wechseln.

Insgesamt ist somit das Ziel dieser Arbeit, ein leichtgewichtiges Framework für die Softwaremessung zu entwickeln, welches es erlaubt, die für die jeweilige Organisation als wichtig erkannten Softwaremaße standardisiert und automatisiert zu erheben. Dieses Framework besteht aus einem methodischen Konzept zur Definition der zu erhebenden Softwaremaße und einer prototypischen Implementierung einer entsprechenden Werkzeugunterstützung.

Der weitere Aufbau dieser Arbeit orientiert sich daher an der Motivation, Herleitung und Implementierung dieses leichtgewichtigen Ansatzes.

1.2.2 Aufbau der Arbeit

Im **Kapitel 2** wird zuerst der Themenbereich der Softwaremessung, welcher auch mit dem immer häufiger verwendeten Begriff *Softwaremetrie* bezeichnet wird, untersucht und näher bestimmt. Dabei sollen insbesondere die Begriffe *Softwaremaß* und *Softwaremetrik* gegeneinander abgegrenzt werden. Daneben wird ein grundlegender Überblick über die Ziele und Einsatzmöglichkeiten der Softwaremessung gegeben. Weiter werden Kriterien aufgezeigt, anhand derer die unüberschaubare Menge der in der Literatur beschriebenen Softwaremaße näher klassifiziert und geordnet werden kann. So enthält beispielsweise allein die Softwaremaß-Sammlung *ZD-MIS* (Zuse/Drabe – Measure-Information-System) [Zus98] über 1500 Softwaremaße zusammen mit einer einheitlichen Beschreibung.

Definition und Ziele der Softwaremessung

Auch wenn somit an verfügbaren Softwaremaßen kein Mangel herrscht, macht es verständlicherweise wenig Sinn, im Rahmen eines Softwaremessungsvorhabens willkürlich einige Softwaremaße auszuwählen und diese während des Projektes zu erfassen. Stattdessen erfolgt der Einsatz von Softwaremessverfahren üblicherweise aufgrund unternehmerischer Ziele (z. B. Verbesserung der Kundenzufriedenheit) oder zumindest auf Basis von Zielsetzungen innerhalb des Software Engineerings (z. B. Reduzierung des Wartungsaufwandes für die Software), so dass die hierfür relevanten und sinnvoll anwendbaren Softwaremaße entsprechend ausgewählt werden müssen. Daher sollen verschiedene Strategien für eine zielorientierte Auswahl von Softwaremaßen bzw. für einen sinnvollen Einsatz von Softwaremessverfahren dargestellt werden. Zusätzlich wird untersucht, ob es Standard-Softwaremaße gibt, welche für die meisten Softwareprojekte relevant sind und daher grundsätzlich erhoben werden sollten.

Auswahl geeigneter Softwaremaße

Anschließend wird in **Kapitel 3** der aktuelle Forschungsstand bzw. der aktuelle Stand der Technik auf dem Gebiet der Messwerkzeuge und -verfahren dargestellt. Die Messwerkzeuge lassen sich dabei folgendermaßen gliedern:

Stand der Technik

- Messwerkzeuge, welche in Standard-Entwicklungsumgebungen integriert sind, dienen hauptsächlich zur Anwendung einfacher Softwaremaße auf den Programmcode bzw. auf UML-Diagramme.
- Sogenannte *Dashboards* können die Ergebnisse bestimmter, vordefinierter Softwaremaße visualisieren. Dabei handelt es sich meistens um Softwaremaße, welche auf Managementebene relevant sind und deren Ausprägungen häufig plakativ in Form einer Ampel dargestellt werden.
- Sogenannte Projektleitstände versuchen, Softwaremaße und sonstige Projektinformationen zielgerichtet zu sammeln und entsprechend aufbereitet zu visualisieren. Jedoch sind hier die Softwaremaße im Gegensatz zu den Dashboards nicht fest vorgegeben, sondern frei konfigurierbar.

Nachdem in **Kapitel 4** zunächst kurz dargestellt wird, welche Softwaremaße insbesondere auf Management-Ebene von Interesse sind, wird anschließend die Kernidee dieser Arbeit erläutert.

Softwaremaße auf Managementebene

Zu den Hauptanforderungen an das innerhalb dieser Arbeit entwickelte Framework zur Softwaremessung gehört, dass die zu erhebenden Softwaremaße frei definiert und zur Projektlaufzeit automatisiert erfasst werden können. Damit ergibt sich die Anforderung, dass die Identifikation der zu messenden Entitäten zur Projektlaufzeit maschinell erfolgen muss. Daher wird untersucht, welche Elemente des Softwareentwicklungsprozesses es gibt, die als Anknüpfungspunkte für Softwaremaße dienen können. Die Verwendung dieser Anknüpfungspunkte ermöglicht es, zur Laufzeit die zu messenden Entitäten automatisiert zu identifizieren.

Anknüpfungspunkte für Softwaremaße

In diese Betrachtungen werden verschiedene Vorgehensmodelle mit einbezogen und auf die Anwendbarkeit dieses Ansatzes hin untersucht. Dabei werden konkrete Handlungsanweisungen für die Definition von Softwaremaßen erarbeitet, welche dann letztendlich zu dem Softwaremessungs-Framework führen.

Eine wichtige Messgröße, welche insbesondere im Rahmen des Projektmanagements von Bedeutung ist, ist der Projektfortschritt. Daher soll am Beispiel des *Feature Driven Developments* eine Möglichkeit der Fortschrittsmessung dargestellt werden, welche sich auch (zumindest teilweise) automatisieren lässt.

Fortschrittsmessung

Die Ermittlung des Projektfortschritts setzt grundsätzlich die Kenntnis des geplanten Aufwandes voraus. Da dieser insbesondere zu Beginn des Projekts und damit zum Zeitpunkt des Vertragsabschlusses nur schwer und mit entsprechenden Fehlerwahrscheinlichkeiten zu bestimmen ist, sollen auch alternative Bepreisungsmodelle für Softwareprojekte dargestellt werden.

Vertragsmodelle

Zum Abschluss dieses Kapitels wird der *Live-Ansatz* vorgestellt. Dieser wurde entworfen, um die Eignung einer Softwareentwicklungsumgebung für die Anwendung agiler Methoden zu beurteilen. Es wird jedoch gezeigt, dass mittels der im Live-Ansatz dargestellten Konzepte auch die Effektivität von Softwaremessungsmaßnahmen beurteilt werden kann.

Reifegradmodell zur Effektivität von Softwaremessung

Daran anschließend wird in **Kapitel 5** das Konzept für das Softwaremessungs-Framework erläutert und eine konkrete Implementierungsmöglichkeit mittels der Open Source Software *Maven* beschrieben, welches ein Framework zur Automatisierung von Aufgaben im Rahmen der Softwareentwicklung ist. Zu diesem Zweck wird zunächst die grundlegende Architektur von Maven und dessen Erweiterungsmöglichkeiten mittels Plugins untersucht. Anschließend wird die Architektur des Softwaremessungs-Frameworks beschrieben.

Implementierung des Ansatzes

Es folgt in **Kapitel 6** eine Evaluierung des innerhalb dieser Arbeit entwickelten Softwaremessungs-Frameworks anhand zweier studentischer Softwareentwicklungsprojekte. An dieser Stelle werden auch die Grenzen des beschriebenen Ansatzes erläutert.

Evaluierung

Abschließend werden in **Kapitel 7** die Ergebnisse dieser Arbeit zusammengefasst und es wird ein Ausblick auf weitere Forschungsaspekte bzw. noch offene Problemfelder gegeben.

2 Softwaremessung

You cannot have a science
without measurement.

(Richard W. Hamming)

2.1 Definition und Ziele der Softwaremessung

2.1.1 Grundbegriffe

Die Softwaremessung beschäftigt sich als Teilgebiet der Softwaretechnik mit der Messung von Eigenschaften von Software und den bei der Erstellung der Software verwendeten Prozessen. Alternativ wird immer häufiger auch der Begriff *Softwaremetrie* (vgl. [Wik]) verwendet, welcher laut Zuse erstmals im Jahre 1987 eingeführt wurde (vgl. [Zus98], S. 15).

Softwaremetrie

Der Begriff *Messung* bezeichnet dabei nach [FP98] einen Prozess, bei dem Attributen von Entitäten aus der realen Welt (den Messobjekten) Messwerte nach klar definierten Regeln zugeordnet werden.

Die Softwaremaße beziehen sich dabei auf Attribute eines oder mehrerer Objekte der Softwareanwendung, des Softwareentwicklungsprozesses oder der Softwarewartung. Daher verwenden wir in Anlehnung an [Dum01] die folgende Definition:

Definition 2.1 *Die **Softwaremessung** ist der Prozess der Quantifizierung von Attributen der Objekte bzw. Komponenten des Software Engineerings mit der Ausrichtung auf spezielle Messziele und ggf. der Einbeziehung von Messwerkzeugen.*

Bei der Softwaremessung werden also Objekte aus der Welt des Software Engineerings auf Elemente der Mathematik und dabei insbesondere auf Zahlen und Symbole abgebildet, mit dem Ziel, eine Eigenschaft oder ein Attribut einer Entität aus dem Software Engineering quantitativ zu bestimmen. Beispielsweise wird durch das Softwaremaß *Lines of Code* (LOC) der Eigenschaft *Länge* eines Software-Sourcecodes ein Wert, nämlich die Anzahl der Code-Zeilen, zugeordnet.

Softwaremessung als Abbildung

Entität

Daher definieren sowohl [FP98] als auch [Zus98] eine Messung als Abbildung von den Eigenschaften von Objekten der empirischen Welt auf formale Objekte (Zahlen oder Symbole). Insbesondere erscheint an dieser Stelle der Begriff der *Abbildung* von Bedeutung, da sich diese auch bei der Definition des *Maßes* in der Mathematik finden lässt. Mathematisch ist ein Maß eine Abbildung von einem Teilmengensystem über einer Grundmenge auf die Menge der reellen und komplexen Zahlen. Daher definieren wir das *Softwaremaß* innerhalb dieser Arbeit folgendermaßen:

Softwaremaß

Definition 2.2 *Ein **Softwaremaß** ist eine Abbildungsvorschrift, welche Attributen von Elementen des Software Engineerings einen Zahlenwert zuweist, um dieses Attribut näher zu charakterisieren. Dieser Zahlenwert wird als **Messwert** bezeichnet.*

Daher muss beispielsweise bei dem Softwaremaß *LOC* (siehe oben) auch angegeben werden, welche Sourcecode-Zeilen tatsächlich zu zählen sind. In der Literatur und in der praktischen Anwendung differieren hier die Meinungen erheblich. Insbesondere geht es dabei darum, ob auch Kommentar- bzw. Leerzeilen zu zählen sind oder nicht (siehe dazu auch Abschnitt 2.3).

Mess-Skala

Sowohl [FP98] als auch [Zus98] weisen an dieser Stelle darauf hin, dass diese Messwerte sich auf verschiedene Mess-Skalen (z.B. nominale, ordinale oder absolute) beziehen können. Diese Detaillierung ist jedoch für das weitere Verständnis dieser Arbeit nicht notwendig. Für eine tiefergehende Betrachtung wird daher auf die genannten Monographien verwiesen.

Metrik

Viel wichtiger erscheint es stattdessen, an dieser Stelle den Begriff des Maßes von dem der *Metrik* abzugrenzen, auch oder gerade weil die Begriffe Softwaremaß und Softwaremetrik sowohl in der deutschsprachigen als auch in der englischsprachigen Literatur oft synonym verwendet werden. Eine Metrik im mathematischen Sinn ist folgendermaßen definiert:

Definition 2.3 *Sei X eine beliebige Menge. Eine Abbildung $d : X \times X \longrightarrow \mathbb{R}$ heißt **Metrik**, wenn für beliebige x, y, $z \in X$ die folgenden Bedingungen erfüllt sind:*

$$
\begin{aligned}
&d(x,y) = 0 \Leftrightarrow x = y && \textit{(Definitheit)}\\
&d(x,y) = d(y,x) && \textit{(Symmetrie)}\\
&d(x,y) \leq d(x,z) + d(z,y) && \textit{(Dreiecksungleichung)}
\end{aligned}
$$

Softwaremetrik

Der Wert der Abbildung $d(x, y)$ kann als der Abstand der Elemente x und y voneinander aufgefasst werden. Somit handelt es sich also bei einer *Softwaremetrik* eigentlich um ein Abstandsmaß:

Definition 2.4 *Eine **Softwaremetrik** ist eine Abstandsfunktion, die Paaren von Attributen von Softwarekomponenten Zahlen zuordnet.* (vgl. [Dum01])

Beipiele für Softwaremetriken

In [FP98] (S. 103) finden sich auch entsprechende Beispiele für Softwaremetriken, welche auch der mathematischen Definition einer Metrik Genüge tun. Beispielsweise werden bei der *N-Versionen-Programmierung* (vgl. [Avi95], [AC77]) n verschiedene Versionen einer kritischen Softwarekomponente unabhängig voneinander implementiert. Die Annahme dabei ist, dass die Wahrscheinlichkeit, dass die n verschiedenen Entwicklerteams dabei jeweils den gleichen Fehler bei der Implementierung begehen, gering ist. Die n Programmversionen werden dann gleichzeitig ausgeführt und falls sich das Verhalten der verschiedenen Versionen unterscheiden sollte, wird maschinell das Ergebnis ausgewählt, welches die Mehrheit der Programme liefert. Dadurch soll insgesamt die Verlässlichkeit der Anwendung erhöht werden. Wenn man sich nun

dafür interessiert, wie unterschiedlich die einzelnen Implementierungen tatsächlich ausgefallen sind, kann man eine *Verschiedenheitsmetrik* definieren, welche den Unterschied zwischen zwei Implementierungen bestimmt. Unabhängig davon, dass dazu zuerst festgelegt werden müsste, wie der Unterschied zwischen zwei Programmen definiert sein soll, handelt es sich bei dieser Softwaremetrik um ein echtes Abstandsmaß.

Als zweites Beispiel wird in [FP98] eine Softwaremetrik genannt, welche den Unterschied zwischen der Spezifikation eines Programmes und dem implementierten Programm selbst bestimmt. Dazu muss man diese Metrik als Abstandsmaß zwischen zwei Produkten definieren und sowohl das Programm als auch die Spezifikation jeweils als Produkt betrachten. Der Abstand zwischen dem Programm und der Spezifikation nimmt z. B. zu, wenn das Programm eine spezifizierte Funktionalität nicht aufweist.

Ein interessanter Aspekt an dem letztgenannten Beispiel ist, dass hier zwar wiederum eine Softwaremetrik definiert wurde, welche die Distanz zwischen einer Programmspezifikation und dem eigentlichen Programm berechnet, jedoch bei einer gegebenen Spezifikation der Wert der Softwaremetrik als Maß für die „Richtigkeit“ eines Programmes betrachtet werden kann. Bzgl. dieses Softwaremaßes ist ein Programm also umso „richtiger“, je kleiner seine Distanz zur Spezifikation ist.

Definition eines Maßes auf Basis einer Metrik

2.1.2 Objekte und Ziele der Softwaremessung

Softwaremaße können zunächst nach der Art der zu vermessenden Objekte unterschieden werden. Da sowohl materielle Objekte (z. B. Personen), ideelle Objekte (z. B. Programme) als auch konzeptionelle Objekte (z. B. Prozesse) vermessen werden können, werden diese verschiedenen Arten häufig unter dem Begriff *Entität* subsumiert (vgl. [FP98], S. 5, [Sim01], S. 21).

Klassifizierung von Softwaremaßen

Entsprechend der während der Softwarebearbeitung identifizierbaren, messbaren Entitäten hat sich die folgende in [FP98] dargestellte Klassifizierung allgemein etabliert (vgl. [ABCR94], [PGF96], [Zus98]):

- **Prozessmaße** beziehen sich auf Aktivitäten innerhalb des Softwareentwicklungsprozesses.
- **Produktmaße** beziehen sich auf jede Art von Arbeitsergebnissen (z. B. Programme, Dokumente, Testergebnisse), welche aus den Prozessaktivitäten resultieren.
- **Ressourcenmaße** beziehen sich auf Mittel, die für die einzelnen Prozesse benötigt werden.

Ziele der Softwaremessung

Gründe, diese Softwaremaße innerhalb des Software Engineerings zu erheben, sind laut [PGF96] die Bedürfnisse, die Prozesse, Produkte und Ressourcen zu charakterisieren, zu beurteilen, zu prognostizieren und zu verbessern:

Charakterisierung

Die genannten Entitätstypen werden **charakterisiert**, um ein Verständnis von Prozessen, Produkten und Ressourcen zu erlangen und eine Vergleichsgrundlage für zukünftige Bewertungen zu schaffen.

Beurteilung

Während des Projektverlaufs kann mittels Softwaremessung regelmäßig **beurteilt** werden, ob der aktuelle Status mit den Planvorgaben übereinstimmt. Softwaremaße fungieren diesbezüglich als entsprechende Sensoren. Auch dienen Softwaremaße dazu, die Erreichung von Qualitätszielen und die Wirkungen von Technologie- und Prozessverbesserungen in Bezug auf die Produkte und Prozesse zu beurteilen.

Prognose

Softwaremessung dient der **Prognose**, da sie eingesetzt wird, um ein Verständnis der Beziehungen zwischen den Prozessen und den dabei erzeugten Produkten zu schaffen, und um Modelle zur Beschreibung dieser Beziehungen erstellen zu können. Dadurch können bestimmte Attribute prognostiziert werden, indem die gemessenen Werte bestimmter anderer Attribute verwendet werden. Dies ist notwendig, um realistische Kostenschätzungen, Zeitpläne und Qualitätsziele aufzustellen, und um entsprechende Ressourcen einzuplanen. Prognosemaße sind auch die Basis, um Trends durch Extrapolation von Messwerten zu erkennen, wodurch wiederum die vorhanden Kosten-, Aufwands- und Qualitätsabschätzungen aufgrund der aktuellen Daten überarbeitet werden können. Hochrechnungen und Abschätzungen auf Basis historischer Daten helfen bei der Risikoanalyse und bei der Lösung von Zielkonflikten in Bezug auf Design- und Kostenanforderungen.

Verbesserungsziele

Softwaremessung dient **Verbesserungszielen**, da sie eingesetzt wird, um quantitative Informationen zu sammeln, mittels derer Hindernisse, grundlegende Probleme oder Ineffizienzen erkannt und andere Möglichkeiten identifiziert werden können, um die Produktqualität oder die Performance von Prozessen zu verbessern. Softwaremessung hilft auch bei der Planung und dem Controlling von Verbesserungsbestrebungen. Die Messung der aktuellen Performance liefert die Messlatte, anhand derer beurteilt werden kann, ob Verbesserungsbemühungen die geplante Wirkung zeigen und ob dabei Nebeneffekte auftreten. Geeignete Softwaremaße helfen auch bei der Vermittlung von Zielen und Gründen für Verbesserungsbemühungen, um dadurch eine breite Unterstützung durch die Projektbeteiligten zu erreichen (vgl. [PGF96]).

2.1.3 Klassifizierung von Softwaremaßen

Wie im Abschnitt 2.1.2 bereits erwähnt, können Softwaremaße zunächst aufgrund der zu messenden Objekte in Prozess-, Produkt- und Ressourcenmaße kategorisiert werden. Darüber hinaus können Softwaremaße auch dahingehend differenziert werden, ob die zu messenden Attribute einzelner Entitäten unmittelbar messtechnisch erfasst werden können, oder ob eine wertmäßige Bestimmung einzelner Merkmale nur im Kontext mit anderen Merkmalen oder Entitäten sinnvoll ist. In [FP98] werden für diese Unterscheidung die Begriffe *interne Attribute*und *externe Attribute* verwendet.

Direkte und indirekte Softwaremaße

Da jedoch die Softwaremaße selbst klassifiziert werden sollen und dabei der Unterschied herausgestellt werden soll, ob eine Messung unmittelbar möglich ist oder nicht, werden im Weiteren in Anlehnung an [Sim01] die Begriffe *direkte* und *indirekte*

Softwaremaße verwendet.

- **Direkte Softwaremaße** beziehen sich auf Prozess-, Produkt- oder Ressourcen-Attribute, welche unmittelbar durch alleinige Untersuchung der entsprechenden Entität messbar sind.
- **Indirekte Softwaremaße** beziehen sich auf Attribute von Prozess-, Produkt- oder Ressourcen-Entitäten, welche von anderen Entitäten abhängig sind. Hier muss die zu untersuchende Entität in ihrem Kontext betrachtet werden; und das Messergebnis hängt mehr vom Verhalten der Entität als von dieser selbst ab.

Beispielsweise können viele Attribute des Sourcecodes unmittelbar durch einfaches Messen ermittelt werden. Hierzug gehören die Länge (z. B. in LOC gemessen) oder die Komplexität (z. B. Anzahl der Entscheidungspunkte), die somit zu den direkten Produktmaßen zählen. Anders verhält es sich z. B. mit der Anzahl der in einem Sourcecode-Objekt enthaltenen Fehlern. Abgesehen von den eher einfach zu ermittelnden Syntaxfehlern, welche üblicherweise vom Compiler entdeckt werden, muss zum Erkennen von Fehlfunktionen der Code ausgeführt werden und z. B. mittels eines Unit-Test-Mechanismus überprüft werden. Hierbei hängt die Anzahl der gefunden Fehler auch stark von der Güte, d. h. dem Abdeckungsgrad der Unit-Tests ab. Noch weniger unmittelbar zugänglich ist die Anzahl der auf Programmierfehler zurückzuführenden Fehlfunktionen, die erst beim Kunden bzw. beim Anwender entdeckt werden, und somit die Zuverlässigkeit der Software. Dieses Softwaremaß hängt von der Umgebung, in der die fertige Software eingesetzt wird, ab und davon, welche Funktionen der Software im Produktivbetrieb tatächlich aufgerufen werden.

Beispiele indirekter Softwaremaße

Auf Managementebene sind diese indirekten Softwaremaße oftmals von großem Interesse. So soll beispielsweise die Kosten-Effizienz von Aktivitäten (z. B. von Code-Inspektionen) oder die Produktivität von Entwicklerteams ermittelt werden, um Rückschlüsse auf Kostensenkungsmöglichkeiten zu erhalten. Auch der Kunde bzw. der Anwender ist hauptsächlich an indirekten Softwaremaßen wie der Zuverlässigkeit, der Benutzerfreundlichkeit oder der Wartbarkeit interessiert, da er von diesen Aspekten unmittelbar betroffen ist. Diese Faktoren haben unmittelbaren Einfluss auf die *Total Cost of Ownership* (TCO), d. h. die Gesamtheit der Kosten einer Investition, die über ihren kompletten Lebenszyklus hinweg anfallen, und sind somit für die Kaufentscheidung relevant.

Managementrelevante Informationen mittels indirekter Softwaremaße

Total Cost of Ownership

Allerdings sind diese indirekten Softwaremaße in der Regel schwieriger zu messen als die direkten. Zusätzlich können sie oft erst relativ spät im Rahmen des Entwicklungsprozesses erfasst werden. So kann z. B. die Zuverlässigkeit erst getestet werden, nachdem die Entwicklung abgeschlossen und das System betriebsbereit ist (vgl. [FP98], S. 74 f.).

2.1.3.1 Prozessmaße

Wie im Abschnitt 2.1.2 angeführt, dienen Prozessmaße dazu, ein besseres Verständnis für die verwendeten Prozesse und deren Performance zu erhalten. Allerdings

können hierbei nur eine beschränkte Anzahl von Prozessmerkmalen direkt gemessen werden. Dazu zählen z. B.

- die Dauer eines Prozesses bzw. einer seiner Aktivitäten oder
- der mit einem Prozess bzw. einer bestimmten Aktivität verbundene Aufwand.

Da jedoch Prozesse und deren Effizienz häufig im Verhälnis zur aufgewendeten Zeit oder den angefallenen Kosten betrachtet werden, können die meisten Prozessmerkmale nur indirekt gemessen werden (vgl. [FP98], S. 74 ff.). Die folgenden Beispiele indirekter Prozessmaße sollen dies illustrieren:

Prozentsatzmethode zur Aufwandsschätzung

- Insbesondere im Rahmen der Prognose zukünftiger Projektaufwände ist man häufig daran interessiert, wie sich die Aufwände auf die einzelnen Phasen oder Prozesse eines Projektes verteilen und ob diese Aufwandsverteilung über verschiedene Projekte hinweg stabil bleibt. In diesem Fall, der auch bereits von Boehm [Boe81] so beschrieben wurde und als Prozentsatzmethode zur Aufwandsschätzung bekannt ist (vgl. [Hen02], S. 217 f.), könnte man über die bekannten Aufwände bereits abgeschlossener Projektphasen auf die noch erforderlichen Aufwände für die kommenden Projektphasen schließen.

- Die genannte Prozentsatzmethode wurde hauptsächlich für wasserfallartige Vorgehensmodelle entwickelt und ist auch sehr grobgranular, so dass eine Anwendbarkeit für heutige, oftmals iterativ durchgeführte Projekte nicht unbedingt ratsam erscheint. Es wird jedoch im Abschnitt 4.2.4.4 ein Verfahren dargestellt, bei dem diese Prozentsatzmethode bei der Beurteilung des Fertigstellungsgrades einzelner *Features* einer Anwendung eingesetzt wird. Dabei ist es notwendig, die Aufwandsverteilung für die verschiedenen Design- und Programmiertätigkeiten bei der Implementierung eines Features bestimmen zu können.

Bestimmung des Projektfortschritts

- Allgemein gehört die Fortschrittsmessung zu den indirekten Prozessmaßen. Üblicherweise werden die folgenden Indikatoren für die Bestimmung des aktuellen Projektfortschrittes verwendet (vgl. [Cla02]):
  - **Abgeschlossene Aktivitäten:** Hierbei wird der Projektfortschritt als Verhältnis der bereits abgeschlossenen Projektaktivitäten zu den insgesamt geplanten Projektaktivitäten angegeben. Zusätzlich können die Aktivitäten dabei gewichtet werden, um genauere Fortschrittsabschätzungen zu erhalten.
  - **Vollständige Artefakte:** Der Projektfortschritt wird als Prozentsatz der bereits fertiggestellten Artefakte in Relation auf die Gesamtmenge der zu erstellenden Artefakte angegeben.

  Ein Nachteil der ersten Methode ist, dass das Projektteam zwar Fortschritte für durchgeführte Aktivitäten melden kann, aber dabei nicht notwendigerweise auch Produkte erzeugt haben muss. Der Nachteil der zweiten Methode ist, dass nur vollständige Produkte in die Fortschrittsberechnung eingehen. Auch

gehen unter Umständen verschieden aufwändig zu erstellende Artefakte in gleicher Weise in die Bewertung ein. Auch hier soll auf die im Abschnitt 4.2.4.4 dargestellte Fortschrittsmessung verwiesen werden.

- In [TT06] wird ein Softwaremessungsvorhaben beschrieben, bei dem unter anderem die Effektivität von *Peer Reviews* sowohl von Design-Dokumenten als auch von Programmcode untersucht werden sollte. Dazu wurde die Verteilung der Herkunft von Softwaremängeln in Bezug auf die einzelnen Phasen des Entwicklungsprozesses und der Aufwand für die jeweilige Fehlerbeseitigung untersucht. Es zeigte sich, dass nach Einführung der Peer Reviews und zusätzlicher Checklisten zur Fehlererkennung die Softwaremängel selbst um 33 % und der Aufwand zur Fehlerbeseitigung um durchschnittlich 61 % abnahm.

Messung der Methoden-effektivität

2.1.3.2 Produktmaße

Die zu messenden Produkte beschränken sich nicht nur auf diejenigen, welche an den Kunden ausgeliefert werden sollen. Jedes im Rahmen des Softwareentwicklungsprozesses erzeugte Artefakt kann vermessen und beurteilt werden. So kann sowohl der Umfang der von den Entwicklern erzeugten und nur intern verwendeten Prototypen als auch der Umfang von Unit-Test-Implementierungen bei der Beurteilung der Entwicklerproduktivität mit eingehen. Aber auch der Umfang und die Qualität von Check-Listen und Prüfanweisungen kann gemessen werden.

Softwaremessung nicht auf Deliverables beschränkt

Auch hier kann wieder zwischen internen und externen Attributen und somit zwischen direkten und indirekten Softwaremaßen unterschieden werden. Da ein externes Attribut sowohl vom Produktverhalten als auch der Umgebung oder dem Kontext des Produkts abhängt, muss dies bei der Messung berücksichtigt werden. Beispielsweise mag die Reaktionsgeschwindigkeit einer Web-Anwendung gut sein, so lange auf diese nur von einem Anwender zugegriffen wird. Jedoch kann die Reaktionsgeschwindigkeit auf ein nicht mehr ausreichendes Maß sinken, sobald parallele Zugriffe auf die Anwendung erfolgen.

Benutzerfreundlichkeit, Vollständigkeit, Effizienz, Prüfbarkeit, Portabilität, Wiederverwendbarkeit und Interoperabilität sind weitere externe Attribute. Diese Attribute beschreiben nicht nur Eigenschaften von Softwareprogrammen, sondern sind auch Merkmale anderer Artefakte des Softwareentwicklungsprozesses und können daher beispielsweise auch bei Spezifikations- und Designdokumenten gemessen werden (vgl. [FP98], S. 78).

Indirekte Produktmaße

Interne Produktattribute, wie der Umfang, der dafür erbrachte Aufwand und die Kosten, sind oft einfach zu messen. Jedoch gibt es für bestimmte Eigenschaften mehrere Definitionen und verschiedene Anweisungen, wie diese Eigenschaften zu messen sind. Dies trifft sowohl für die Komplexität von Software als auch für das relativ einfache Umfangsmaß LOC zu.

Ein Hauptziel beim Messen interner Produktattribute ist es, dadurch auf die externen Produkteigenschaften zu schließen. Indirekte Qualitätsmaße wie Zuverlässigkeit, Wartbarkeit oder Portierbarkeit sollen mittels der direkt messbaren inter-

Abschätzung externer Produktmerkmale mittels direkter Softwaremaße

nen Eigenschaften abgeschätzt werden oder es sollen zumindest diejenigen Produkte identifiziert werden, die aufgrund ihrer internen Attribute als Problemfälle bzgl. der externen Qualitätsmaße einzuschätzen sind.

Als Indikatoren für eine schlechte Wartbarkeit und teilweise für eine erhöhte Fehleranfälligkeit gelten beispielsweise

- ein hoher Grad an Sourcecode-Komplexität,
- ein großer Umfang einzelner Module oder
- ein hoher Abhängigkeitsgrad zwischen einzelnen Programmmodulen.

2.1.3.3 Ressourcenmaße

Während die Produkte das Ergebnis von Prozessaktivitäten sind, werden unter Ressourcen diejenigen Entitäten verstanden, die zur Durchführung des Prozesses notwendig sind. Dazu gehören insbesondere Personal, Gerätschaften, verwendete Software und Verfahren. Ressourcen können beispielsweise in Bezug auf ihren Umfang (Personalstärke), Kosten (Softwarelizenzen) und Qualität (Güte verschiedener Testverfahren) untersucht und beurteilt werden.

Beurteilung der Kosteneffizienz

Insbesondere die Kosten werden üblicherweise für alle Arten von Ressourcen gemessen und dabei häufig ins Verhältnis zum produzierten Ergebnis gesetzt, um festzustellen, wie die Kosten der einzelnen Ressourcen die Ergebnisse beeinflussen. Beispielsweise können auf diese Weise die Kosten für das Durchführen von *Peer Reviews* der Veränderung in der Anzahl der Softwaremängel (siehe oben) gegenübergestellt werden.

Untersuchung der Produktivität

Parallel zu den Kosten wird häufig auch die Produktivität von Entwicklerteams bzw. deren Veränderung gemessen. Die Produktivität wird dabei meistens als Quotient aus Ergebnisumfang und Aufwand ermittelt (z. B. erstellte Code-Zeilen pro Personentag). Bei diesem indirekten Ressourcenmaß wird also ein Produktmaß (z. B. LOC) mit einem Prozessmaß (Projektaufwand in Personentagen) kombiniert. Dieses Produktivitätsmaß kann zum einen verwendet werden, um verschiedene Entwicklerteams zu vergleichen. Hierbei ist jedoch große Sorgfalt geboten, da unter Umständen die Komplexität der Entwicklungsaufgaben zweier Teams sehr unterschiedlich sein kann. Viel interessanter erscheint hingegen die Untersuchung der Veränderung der Produktivität, wenn andere Ressourcen verändert werden. Beispielsweise kann die Entwicklung der Produktivität nach Einführung einer neuen Werkzeugumgebung für die professionelle Softwareentwicklung untersucht werden. Im Idealfall wird sich die Produktivität dabei infolge des Einarbeitungsaufwands für die neue Umgebung kurzzeitig verschlechtern, um dann anschließend ein höheres Niveau zu erreichen.

Da die Ressourcen in der Regel als Input für die Prozesse des Software Engineerings betrachtet werden, überwiegen hier, wie auch an obigen Beispielen ersichtlich ist, die indirekten Softwaremaße, da diese meistens von zwei Eigenschaften (Kosten/Nutzen, Aufwand/Ergebnis) abhängen.

2.1.4 Softwaremessung und Prozessverbesserung

2.1.4.1 Das CMMI-Reifegradmodell

In den vorhergehenden Abschnitten wurde dargestellt, wie durch Softwaremessung Erkenntnisse über Prozesse, Produkte und Ressourcen des Software Engineerings gewonnen werden können, mit dem Ziel, bestimmte Elemente der Softwareentwicklung zu charakterisieren, zu beurteilen, zu prognostizieren und zu verbessern. Um nicht nur einzelne Elemente des Softwareentwicklungsprozesses, sondern die Qualität des Prozesses selbst beurteilen zu können, wurde am *Software Engineering Institut* (SEI) der *Carnegie Mellon University* in Pittsburgh mit dem *Software Capability Maturity Model* (vgl. [PCCW93]) ein Prozessmodell zur Beurteilung und Verbesserung der Qualität („Reife“) des Software-Entwicklungsprozesses von Organisationen entwickelt. Dieses Prozessmodell wurde später zum Prozessreifegradmodell *Capability Maturity Model Integration* (CMMI) (vgl. [CKS03]) weiterentwickelt. Dabei wurden nicht nur andere Produktentwicklungsprozesse (z. B. Hardwareentwicklung), sondern z. B. auch Service-Prozesse in das Reifegrad-Framework integriert.

SW-CMM und CMMI

Die kontinuierliche Verbesserung von Entwicklungsprozessen (vgl. [Ima02]) basiert auf kleinen, evolutionären Schritten. Die Grundidee des CMMI ist, das erreichte Niveau der Prozessqualität mittels fünf Reifegradstufen zu bewerten, deren Definition in Tabelle 2.1 beschrieben ist.

Jedem dieser Reifegrade (ausgenommen Reifegrad 1) ist eine Reihe von Prozessgebieten mit konkreten Anforderungen zugeordnet, deren Erfüllung zur Erreichung des jeweiligen Reifegrades erforderlich ist (vgl. Tabelle 2.1). Ein Prozessgebiet[1] ist jeweils eine Zusammenfassung aller Anforderungen zu einem Thema (z. B. zu Projektplanung, Anforderungsmanagement oder Konfigurationsmanagement) und umfasst eine Reihe von Zielen, die dabei erreicht werden sollen (vgl. [Kne06], S. 14). Dabei schließen die höheren Reifegrade die Anforderungen an die darunterliegenden mit ein (vgl. [Kne06], S. 17 ff.).

Prozessgebiete

Beim Reifegrad 1 (*Initial*) des CMMI sind die Prozesse entweder gar nicht oder nur *ad hoc* definiert. Der Erfolg eines Projektes hängt in erster Linie vom Einsatz und der Kompetenz einzelner Mitarbeiter ab. Dem Reifegrad 1 sind keine Anforderungen und damit keine Prozessgebiete zugeordnet.

Reifegrad 1

Beim Reifegrad 2 (*Managed*[2]) wird der Einsatz wesentlicher Elemente des Projektmanagements wie Anforderungsmanagement, Projektplanung und Projektcontrolling gefordert, um Kosten, Zeitplan und Umfang von Projekten zu planen und zu steuern. Auch enthält das CMMI ab dieser Reifegradstufe explizit die Forderung, die Anforderungen des Prozessgebiets *Messung und Analyse* zu erfüllen. Bei Softwareentwicklungsprojekten wird somit die Anwendung von Methoden der Softwaremessung gefordert.

Reifegrad 2: Projektmanagement

[1] Das CMM spricht noch von *Key Process Areas*, während die Themenbereiche im CMMI nur noch als *Process Areas* bezeichnet werden. In [SSM06] wird angemerkt, dass die Bezeichnung *„key“* zum Ausdruck bringen soll, dass die Erfüllung dieser Anforderungen zwar notwendig aber nicht hinreichend für einen guten Prozess sei (vgl. [SSM06], S. 65 f. [PCCW93], S. 32).

[2] Beim älteren CMM wurde der Reifegrad 2 noch als *Defined* bezeichnet.

Stufe	Name	Prozessgebiete
1	Initial	keine
2	Managed	• Anforderungsmanagement • Projektplanung • Projektverfolgung und -steuerung • Management von Lieferantenvereinbarungen • Messung und Analyse • Qualitätssicherung von Prozessen und Produkten • Konfigurationsmanagement
3	Defined	• Anforderungsentwicklung • Technische Umsetzung • Produktintegration • Verifikation und Validation • Organisationsweiter Prozessfokus • Organisationsweite Prozessdefinition • Organisationsweites Training • Integriertes Projektmanagement • Risikomanagement • Entscheidungsanalyse und -findung
4	Quantitatively Managed	• Organisationsweite Prozess-Performance-Messung • Quantitatives Projektmanagement
5	Optimizing	• Organisationsweite Auswahl und Verbreitung von Innovationen • Ursachenanalyse und Problemlösung

Tabelle 2.1: CMMI-Reifegradstufen (in Anlehnung an [Kne06] und [CMM06])

Reifegrad 3: Definierte Prozesse

Auf der Reifegradstufe 3 (*Defined*) liegt der Schwerpunkt der Anforderungen auf unternehmensweiten Prozessdefinitionen. Der Hauptunterschied zum Reifegrad 2 besteht somit im Geltungsbereich der verwendeten Standards, Prozessbeschreibungen und Vorgehensweisen. Beim Reifegrad 2 kann jedes einzelne Projekt gemäß anderer Prozessbeschreibungen durchgeführt werden. Beim Reifegrad 3 müssen organisationsweite Standards und Prozessbeschreibungen implementiert sein, welche dann für konkrete Projekte nur noch durch *Tailoring* angepasst werden. Ein weiterer Unterschied zum Reifegrad 2 ist, dass Prozesse beim Reifegrad 3 exakter beschrieben werden inklusive aller Prozessaktivitäten. Aufgrund des erreichten Verständnisses der Beziehungen zwischen den Prozessaktivitäten und unter Verwendung geeigneter Prozessmaße soll ein proaktives Management der Prozesse ermöglicht werden (vgl. [CMM06], S. 33 f.).

Reifegrad 4: Systematische Verwendung von Kennzahlen

Beim Reifegrad 4 fordert das CMMI die systematische Verwendung von Softwaremaßen und Kennzahlen, um Ergebnisse von Arbeitsabläufen besser vorhersagen zu können (Aufwand, Fehlerquote, etc.) (vgl. [Kne06], S. 19). Die Voraussetzungen dafür wurden durch die beim Reifegrad 3 implementierten detaillierten Unternehmensprozesse geschaffen. Diese organisationsweiten Prozesse ermöglichen erst den Vergleich von Kennzahlen über Projektgrenzen hinweg. Insbesondere soll Softwaremetrie angewendet werden, um Prozessinstanzen zu erkennen, die ein ungewöhnliches Verhalten aufweisen, oder um Eigenschaften von Prozessen zu identifizieren, welche innerhalb der organisationsweiten Prozessrahmenwerke verbessert werden sollten (vgl. [CMM06], S. 261 f.).

Reifegrad 5: Kontinuierliche Verbesserung

Der Reifegrad 5 schließlich stellt die höchste Stufe im CMMI-Reifegradmodell dar. Bei diesem Reifegrad wird der Schwerpunkt auf eine kontinuierliche Verbesserung der Prozessumsetzung mittels schrittweiser und innovativer Prozess- und Technologieverbesserungen gelegt. Dabei werden quantitative Verbesserungsziele für die Organisation festgelegt, welche regelmäßig mit den sich ändernden Unternehmenszielen abgeglichen und mittels Softwaremessung den tatsächlich erreichten Prozessverbesserungen gegenüber gestellt werden (vgl. [CMM06], S. 38).

Der Hauptunterschied zum Reifegrad 4 liegt in der Art und Weise, wie mit Schwankungen bei der Prozessumsetzung umgegangen wird. Bei Reifegrad 4 soll das Unternehmen in der Lage sein, Prozessinstanzen mit unterdurchschnittlicher Performance zu erkennen. Beim Reifegrad 5 sollen zusätzlich die grundlegenden und projektübergreifend relevanten Ursachen für die Minderperformance erkannt und im organisationsweiten Prozessrahmenwerk korrigiert werden können.

2.1.4.2 Softwaremessung innerhalb des CMMI

Die Umsetzung der Aktivitäten des Prozessgebietes *Messung und Analyse* wird innerhalb des CMMI bereits für die Erreichung des Reifegrades 2 (*Managed*) gefordert. Das Ziel dieses Prozessgebietes ist es, sowohl für das Management als auch innerhalb der einzelnen Projekte Informationen als Basis für Entscheidungen bereitzustellen. Diese Informationen dienen dann

- der objektiven Planung und Schätzung,

- dem Vergleich der tatsächlichen Prozessumsetzung mit den festgelegten Plänen und Zielen,
- der Identifikation und Behebung prozessbezogener Probleme und
- der Bereitstellung einer Datenbasis, um Softwaremessung auch in zukünftigen Prozessen implementieren zu können (vgl. [CMM06], S. 178).

Einblickmöglichkeit in den Entwicklungsprozess

Der Grad der möglichen Anwendbarkeit von Softwaremaßen hängt aus Sicht des CMMI von den Einblickmöglichkeiten in den Entwicklungsprozess (*Visibility Into the Software Process*, vgl. [PCCW93], S. 19 ff.) ab. So stellt im Reifegrad 1 der Softwareentwicklungsprozess eine Art *Black Box* dar, in welche man Ressourcen hineinsteckt und am Ende (hoffentlich) entsprechende Projektergebnisse bekommt. Diese Einblickmöglichkeiten nehmen mit fortschreitendem Reifegrad zu. So ist beispielsweise beim Reifegrad 3 die innere Struktur der Prozesse bekannt und schließlich kann beim Reifegrad 5 nicht nur in existierende Projekte Einblick genommen werden, sondern es sollten sich auch die Auswirkungen potentieller Prozessveränderungen beurteilen lassen.

Anwendung der Softwaremetrie im CMMI

Im Einzelnen ergeben sich damit die folgenden Möglichkeiten der Softwaremetrie in Abhängigkeit vom erreichten Prozessreifegrad (vgl. [FP98], S. 89 ff., [Kne06], S. 44 ff.).

- Da beim Reifegrad 1 der Entwicklungsprozess nur als *ad hoc* definiert ist, können mangels Vergleichsmöglichkeiten die hier erhobenen Softwaremaße nur als Basis dienen, um zukünftige Prozessverbesserungen beurteilen zu können. Beispielsweise können zu diesem Zweck Maße wie Projektdauer, Aufwand und Produktivität erhoben werden.

- Der Reifegrad 2 zeichnet sich durch das Vorhandensein von Projektmanagement aus. Daher sind die Input- und Output-Größen der Softwareprojekte bekannt und können somit gemessen werden. Sinnvoll erscheint beispielsweise die Messung von Umfang und Volatilität der Anforderungen. Auch kann mittels entsprechender Messungen die Einhaltung von Produktgrößen-, Aufwands-, Kosten- und Terminschätzungen überwacht werden.

- Beim Reifegrad 3 sind definierte Prozesse vorhanden. Somit sind auch die einzelnen Aktivitäten des Entwicklungsprozesses bekannt und können vermessen werden. Beispielsweise kann jetzt die Verteilung von Aufwänden auf die einzelnen Prozessaktivitäten gemessen und mit anderen Projekten verglichen werden. Durch die definierten Prozesse ergibt sich auch eine Traceability zwischen den End- und Zwischenprodukten bzw. zu den Anforderungen. Diese kann benutzt werden, um die Entstehung, die Verweildauer und die Behebung von Fehlern zu untersuchen.

- Auf Reifegrad 4 wird das *Quantitative Projektmanagement* unter Anwendung von Verfahren zur Softwaremessung explizit vom CMMI-Modell gefordert. Der Fokus der Softwaremessung liegt darauf, die Varianz in den wesentlichen Kennzahlen zu reduzieren, indem Prozessfehler erkannt und behoben werden.

- Beim *optimierenden* Reifegrad 5 sollen Kennzahlen und Metriken zur Verbesserung der Prozesse eingesetzt werden. Dabei ist es beispielsweise denkbar, verschiedene Varianten einer Prozessdefinition durchzurechnen, um auf diese Weise die effizienteste Prozessvariante zu finden. Ein weiterer Aspekt ist auch die Erhebung von Benchmarks, um die eigenen Prozesse mit denen der Wettbewerber zu vergleichen (vgl. [CMM06], [Jon00]).

2.1.5 Statistische Prozessregelung

Wie bereits im Rahmen der Beschreibung des CMMI-Modells (Abschnitt 2.1.4.2) erwähnt, ist eine Anwendung der Softwaremetrie das *Quantitative Prozessmanagement*. Dieses verwendet Methoden der *Statistischen Prozessregelung* (*Statistical Process Control* (SPC), vgl. [Oak96], [FC04], [CMM06], S. 375 ff.), welche aus dem *Total Quality Management* stammen und auf das Management von Softwareprozessen übertragen wurden (vgl. [Sei03]).

Natürliche und außergewöhnliche Abweichungen bei der Prozess-Performance

Dabei handelt es sich um ein Verfahren, mittels statistischer Methoden verschiedene Abweichungen bei der Umsetzung von Prozessen zu erkennen. Grundsätzlich kann es bei der Abarbeitung von Prozessen immer zu Abweichungen vom erwarteten Sollwert in Folge der natürlichen Streuung kommen. Diese entsteht aus einer Vielzahl kleiner und erwarteter Einzeleinflüsse bei der Interaktion zwischen den Komponenten eines Prozesses und ist relativ gut vorhersehbar. Zusätzlich können außerordentliche Einflüsse den Prozessverlauf beeinflussen. Diese treten nur unregelmäßig auf, haben größere Auswirkungen und machen einen Prozess instabil. Bei ihrem Auftreten bedarf es des Eingreifens des Managements. Bei der natürlichen Streuung sind demgegenüber störende Überreaktionen zu vermeiden (vgl. [Sei03]).

Eine Möglichkeit, natürliche und außerordentliche Faktoren voneinander zu unterscheiden, stellt die Verwendung sogenannter Kontrolldiagramme (*Control Charts*, vgl. Abbildung 2.1) dar. In diesen werden beispielsweise die Messwerte, der Mittelwert aller Messwerte und die ein- bis dreifache Standardabweichung eingetragen. Anschließend lässt sich mittels verschiedener Regeln überprüfen, ob aufgrund dieser Messwerte der Prozess noch unter statistischer Kontrolle ist, da die Abweichungen noch im Rahmen der natürlichen Streuung sind.

Beispiel 1 (Statistische Prozessregelung) *In einer Firma soll der Aufwand untersucht werden, den die Mitarbeiter jeweils für den Support eines bestimmten Produkts erbringen müssen. In Tabelle 2.2 sind die ingesamt pro Tag angefallenen Aufwände x_i in Stunden für den abgelaufenen Monat mit $n = 20$ Werktagen angegeben. Zusätzlich wurde der absolute Betrag δ_j der jeweiligen Veränderung angegeben.*

Aus den Einzelaufwänden x_i wird zunächst der Durchschnittsaufwand $\bar{x}$ berechnet:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \approx 19,2 \quad (n = 20)$$

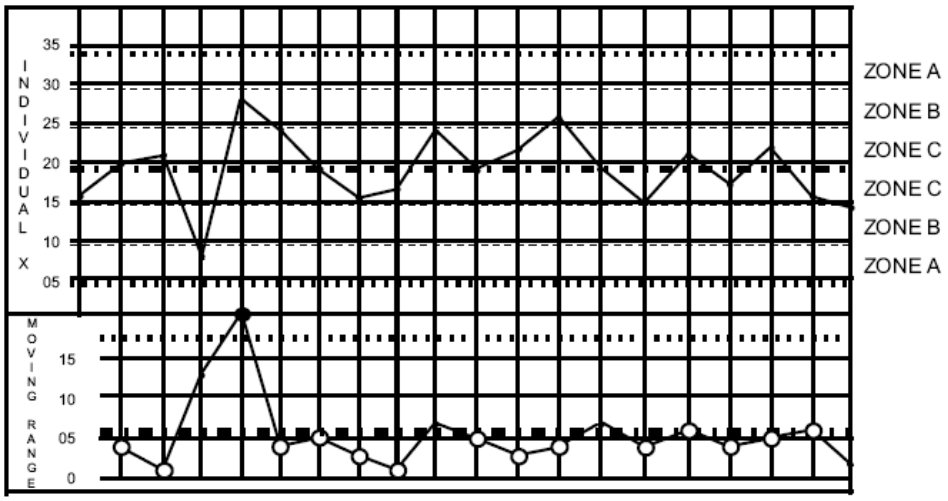


Abbildung 2.1: X/MR-Control-Chart zur Statistischen Prozessregelung

Tag	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Aufwand	16	20	21	8	28	24	19	16	17	24	19	22	26	19	15	21	17	22	16	14
δ		4	1	13	20	4	5	3	1	7	5	3	4	7	4	6	4	5	6	2

Tabelle 2.2: Tabelle mit Aufwänden für den Support eines best. Produkts

*Anschließend wird aus den täglichen Veränderungen δ_i (*moving range*) die durchschnittliche Veränderung $\bar{\delta}$ berechnet:*

$$\bar{\delta} = \frac{1}{n-1} \sum_{j=2}^{n} \delta_j \approx 5,5$$

Die Messwerte x_i, die Veränderungen δ_j und die Durchschnittsmaße $\bar{x}$ und $\bar{\delta}$ werden in das Kontrolldiagramm eingezeichnet (vgl. Abbildung 2.1).

Wenn man nun davon ausgeht, dass die Messwerte normalverteilt sind, dann gilt für die Standardabweichung σ der Messwertverteilung, dass

- *ca. 68,0 % der Messwerte im Bereich $\bar{x} \pm \sigma$,*
- *ca. 95.0 % der Messwerte im Bereich $\bar{x} \pm 2\sigma$ und*
- *ca. 99,7 % der Messwerte im Bereich $\bar{x} \pm 3\sigma$*

liegen (vgl. [Sta05], S. 255 f.).

Im Rahmen des SPC ist es üblich, mit einer Näherung s für die Standardabweichung σ zu arbeiten, die sich aus der durchschnittlichen Veränderung $\bar{\delta}$ der Messwerte ergibt:

$$s = \frac{\bar{\delta}}{d_2} = \frac{5,5}{1,128} = 4,88$$

Hierbei ist d_2 ein sogenannter bias correction factor *nach Harter [Har60], der im obigen Fall den Wert $d_2 = 1,128$ hat.*

*Damit lassen sich nun die obere und die untere Kontrollschranke UCL (*Upper Control Limit*) bzw. LCL (*Lower Control Limit*) zu*

$$UCL = \bar{x} + 3s \quad und \quad LCL = \bar{x} - 3s$$

berechnen und in das Kontrolldiagramm einzeichnen. Der Bereich zwischen LCL und UCL wird in sechs Zonen der Breite s gegliedert (vgl. Abbildung 2.1).

Die Messwerte sollten dabei zufällig, d. h. ohne bestimmte Muster zu bilden, um den Durchnittswert verteilt sein und sich um den Durchschnittswert herum häufen. Anderenfalls wird der Prozess als nicht unter statistischer Kontrolle betrachtet. Um nun Messwerte und Muster von Messwerten zu erkennen, die auf einen unkontrollierten Prozess hindeuten, gibt es verschiedene Empfehlungen. In [Sta05] werden beispielsweise die folgenden Kriterien vorgeschlagen:

Identifikation unkontrollierter Prozesse

- Ein Messwert befindet sich außerhalb der beiden Kontrollschranken. Bei einem kontrollierten Prozess beträgt die Wahrscheinlichkeit hierfür nur 0,3 %.

- Sieben oder mehr aufeinander folgende Messwerte befinden sich auf derselben Seite oberhalb oder unterhalb des Durchschnittswertes. Die Wahrscheinlichkeit p dafür ergibt sich z. B. für $k = 7$ Messwerte aus einer Bernoulli-Kette der Länge k als $p = \left(\frac{1}{2}\right)^k = 0,78\,\%$.

- Mindestens sieben aufeinander folgende Messwerte bilden ein alternierendes Auf-Ab-Muster. Die Wahrscheinlichkeit dafür berechnet sich in analoger Weise als Bernoulli-Kette.

- Zwei aufeinander folgende Messwerte befinden sich außerhalb der Warnschranken. Diese werden meistens im Abstand des zweifachen Standardabweichung um den Durchschnittswert festgelegt.

Die Anwendung der Statistischen Prozessregelung ist somit eine Möglichkeit, Unregelmäßigkeiten bei der Durchführung von Entwicklungsprozessen zu erkennen. Sie kann verwendet werden, um verschiedene Professionalitätsgrade bei der Prozessumsetzung zu beurteilen, und gibt somit Hinweise darauf, ob Maßnahmen zur Prozessverbesserung angezeigt sind.

Eine Weiterentwicklung der Prozessbeurteilung mittels Statistischer Prozessregelung ist die *Six-Sigma-Methode* (vgl. [Töp04]). Dabei werden bestimmte Prozesse ausgewählt und u. a. mittels Kontrolldiagrammen analysiert. Das Ziel dabei ist, die Prozesse soweit zu kontrollieren, dass die Standardabweichung der Prozessergebnisse langfristig auf jeder Seite des Durchschnittswertes sechs Mal ($6\,\sigma$) in die Toleranzgrenzwerte passen (vgl. [Sta05], S. 423 ff.).

Six Sigma

2.2 Ansätze zur Definition von Strategien zur Softwaremessung

Nachdem im vorhergehenden Abschnitt die Ziele der Softwaremessung erläutert wurden, sollen im Folgenden verschiedene Ansätze zur konsistenten Implementierung von Strategien zur zielorientierten Auswahl von Sofwaremessungsmaßnahmen dargestellt werden. Hierbei kann im Wesentlichen zwischen sogenannten *Top-Down*- und *Bottom-Up*-Ansätzen unterschieden werden. Bei den Ersteren wird ein vorgegebenes Ziel oder ein Problem durch schrittweise Zerlegung soweit verfeinert, bis mögliche Lösungsansätze sichtbar werden. Bei der Bottom-Up-Methode wird dagegen zunächst der zerzeitige Ist-Zustand analysiert, um evtl. vorhandene Probleme zu entdecken oder wünschenswerte Veränderungen zu definieren.

2.2.1 Top-Down-Ansätze zur Definition von Strategien zur Softwaremessung

2.2.1.1 Die *Goal-Question-Metric*-Methode

Eine typische Umsetzung des Top-Down-Ansatzes ist die sehr verbreitete *Goal-Question-Metric*-Methode (GQM) [BR88], [Bas92], [BCRS94], [SB99]. Wie in [SB99] ausgeführt wird, wurde diese Methode gegen Ende der 1970er Jahre von V. Basili und D. Weiss im Rahmen verschiedener Industrieprojekte entwickelt und später um weitere, auf D. Rombach zurückgehende, Konzepte erweitert. Die Grundidee dieser Methode besteht darin, dass Softwaremessung immer in Hinblick auf ein oder mehrere konkrete Ziele durchgeführt werden sollte. Diese Ziele werden durch Verfeinerung auf quantifizierbare Fragen überführt, welche wiederum mittels geeigneter Softwaremaße beantwortet werden können (vgl. [BR88]).

Zielgesteuerte Auswahl von Softwaremaßen

Bei der Betrachtung der Ziele muss zwischen *Zielen der Softwaremessung* und *(Qualitäts-)Verbesserungszielen* unterschieden werden. Die GQM-Methode spezifiert und unterstützt Messziele, d. h. Zielsetzungen des Softwaremessungsvorhabens (vgl. [BCRS94]). Diese Ziele können jedoch auf Verbesserungszielen basieren bzw. von diesen abgeleitet sein. Verbesserungsziele richten sich dagegen auf konkrete Zielsetzungen innerhalb eines Unternehmens oder im Rahmen eines Projektes. Dazu gehören beispielsweise die Steigerung der Produkt- oder Servicequalität, Kostenreduzierung, Verringerung der *Time-to-Market*-Zeitspanne oder Verringerung des Projektrisikos. Eine Anleitung zur Bestimmung von Managementzielen und deren Verfeinerung zu Verbesserungs- und Messzielen findet sich unter anderem in [PGF96].

Ziele der Softwaremessung vs. strategische Ziele

Die Messziele selbst führen nicht unmittelbar zu einer etwaigen Qualitätsverbesserung oder Verringerung von Entwicklungsaufwand. Stattdessen sollen dadurch Informationen zur Erreichung der Verbesserungsziele gewonnen werden. Insbesondere soll die GQM-Methode einen Lernprozess der Beteiligten über die Zusammenhänge im Rahmen der Softwareentwicklung bewirken ([BCRS94]).

Konkret beschreibt die GQM-Methode drei Ebenen der Messausführung:

- **Goal – Konzeptuelle Ebene**
  Diese Ebene behandelt die Frage, welche Ziele mit der Softwaremessung erreicht werden sollen. Diese Ziele werden in Bezug auf einen Zweck (welches Objekt und warum), einer Perspektive (welcher Aspekt und wer) und einem Kontext definiert.

- **Question – Operationale Ebene**
  Auf dieser Ebene wird versucht, die Ziele mittels verschiedener quantifizierbarer Fragen näher zu gliedern und genauer zu bestimmen (vgl. [BR88]). Dabei müssen die Fragen so gewählt werden, dass durch deren Beantwortung der Grad der Zielerreichung beurteilt werden kann (vgl. [BCRS94]). Die Fragen dienen somit der Operationalisierung der Ziele, d. h. den Zielen werden dadurch Indikatoren zugeordnet, um sie damit der Analyse zugänglich zu machen. Jeder Frage wird auch gleich eine erwartete Antwort in Form eine Hypothese zugeordnet. Dadurch soll das Projektteam dazu angeregt werden, über die aktuelle Situation nachzudenken, wodurch wiederum ein besseres Verständnis für die Prozesse bzw. das Produkt stimuliert werden soll. (*„Comparison of real answers with these hypotheses will create deep understanding of implicit knowledge and therefore largely contribute to the learning effects of measurement.“*, [BCRS94], S. 579)

- **Metric – Quantitative Ebene**
  Nachdem die Ziele mittels Fragen operationalisiert wurden, werden auf dieser Ebene diejenigen Softwaremaße bestimmt, die zur Beantwortung der Fragen beitragen sollen.

Beispiel 2 (Anwendung der GQM-Methode) *In Abbildung 2.2 ist als Beispiel für die Anwendung der GQM-Methode die Ableitung von Fragen und Softwaremaßen ausgehend von einem konkreten Ziel dargestellt (vgl. [FP98], S. 85). Das Messziel besteht darin, die Wirksamkeit von Code-Standards zu überprüfen, d. h. ob Sourcecode, der unter Berücksichtigung dieser Standards erstellt wurde, in irgendeiner Art und Weise demjenigen überlegen ist, der ohne die Anwendung von Code-Standards erstellt wurde.*

Dabei ergibt sich zunächst die Frage, welche Programmierer sich überhaupt an den Code-Standard halten. Anschließend ist zu untersuchen, wie sich die Produktivität von Programmierern, die sich an den Standard halten, zu derjenigen der übrigen verhält. Ein ähnlicher Vergleich muss auch bzgl. der Qualität des dabei erzeugten Sourcecodes durchgeführt werden.

Nachdem die Fragen definiert worden sind, müssen diese dahingehend untersucht werden, welche Entitäten gemessen werden müssen, um die Fragen zu beantworten. Beispielsweise muss, vorzugsweise durch eine maschinelle Analyse des Sourcecode-Repositorys, festgestellt werden, welche Programmierer die vorgegebenen Code-Standards anwenden. Um jedoch die Wirksamkeit von Code-Standards tatsächlich beurteilen zu können, benötigt man zusätzlich Informationen darüber, welche Erfahrungen die Programmierer mit dem Standard, der Programmiersprache selbst und der

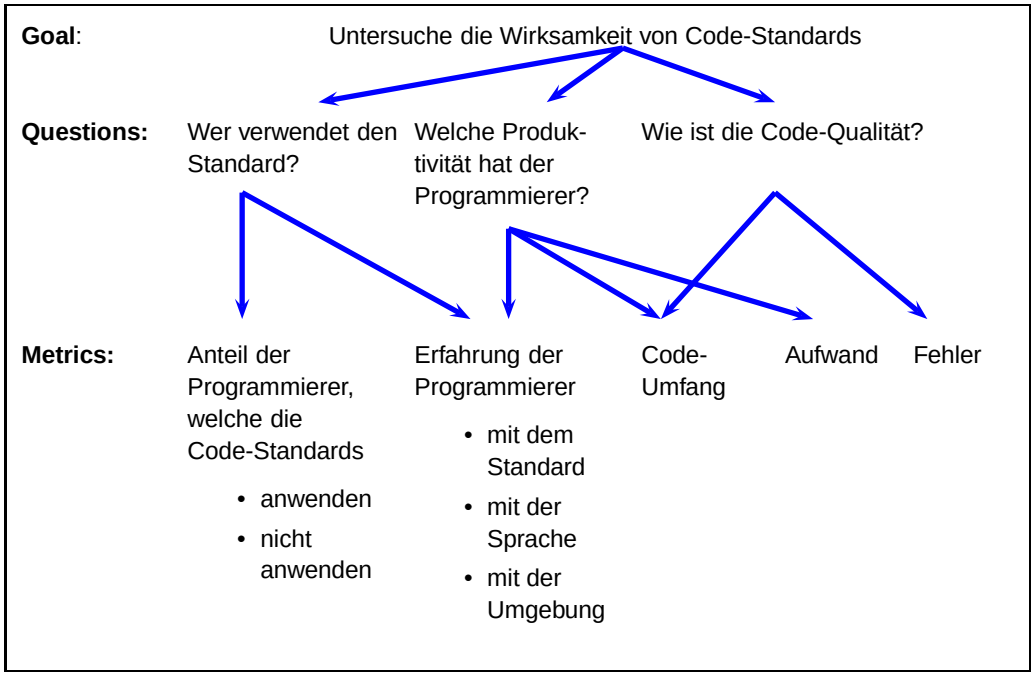


Abbildung 2.2: Beispielhafte Anwendung des GQM-Verfahrens

Entwicklungsumgebung haben. Die Frage nach der Produktivität wiederum bedarf einer entsprechenden Definition von Entwicklerproduktivität*, welche in der Regel zu dem Quotienten aus entsprechenden Maßen für den Produktumfang und den Entwicklungsaufwand führt.*

Kombination verschiedener Softwaremaße

Das Beispiel illustriert, dass im Allgemeinen mehrere Softwaremaße erforderlich sind, um eine bestimmte Frage zu beantworten. Umgekehrt kann ein bestimmtes Softwaremaß auch als Indikator für verschiedene Fragen dienen. Viel bedeutsamer ist jedoch, dass zusätzlich zu den Fragen und den Softwaremaßen ein Verfahren oder eine definierte Vorgehensweise benötigt wird, wie die Softwaremaße sinnvoll zu kombinieren sind, um dadurch tatsächlich Antworten auf die Fragen zu liefern (vgl. [FP98]).

Der GQM-Plan

Diese Vorgehensweise wird in einem sogenannten *GQM-Plan* definiert (vgl. [SB99]). Der GQM-Plan ist ein Dokument, welches Beschreibungen aller Ziele, Fragen, Hypothesen und Softwaremaße für ein Softwaremessungsvorhaben beinhaltet. Er beschreibt die Zerlegung von Messzielen in Fragen und deren Zuordnung zu geeigneten Softwaremaßen und stellt somit die formale Dokumentation des Softwaremessungsvorhabens dar. Dieses Dokument dient als Basis für die Messanleitung, welche Vorschriften für das Erfassen der benötigten Softwaremaße enthält. Der GQM-Plan dient als Leitfaden für die Interpretation der zu erhebenden Daten. Die Messergebnisse sollen dabei Anworten auf die im GQM-Plan definierten Fragen geben, so dass Rückschlüsse auf die GQM-Ziele möglich sind.

GQM-Plan als Leitfaden zur Messdurchführung

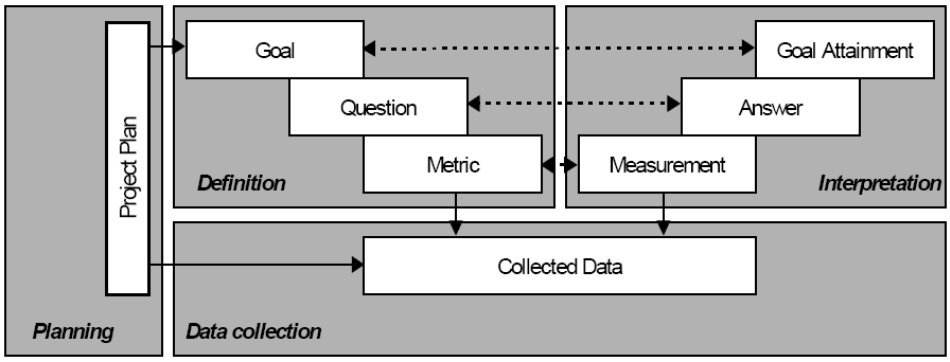


Abbildung 2.3: Phasen der GQM-Methode

Phasen eines GQM-Projekts

Die Durchführung eines GQM-Projekts erfolgt in vier Phasen, welche nachfolgend beschrieben und in Abbildung 2.3 illustriert werden (vgl. [SB99]):

1. In der *Planungsphase* werden die nötigen Informationen für die Vorbereitung und Durchführung eines Softwaremessungsvorhabens gesammelt. Hierzu gehört insbesondere die Auswahl des (Qualitäts-)Verbesserungszieles, d. h. des Bereiches, der aus Managementsicht Optimierungspotenziale aufweist. Zusätzlich wird in dieser Phase ein dediziertes GQM-Team aufgestellt und ein Beispielprojekt zur Anwendung der GQM-Methode ausgewählt.

2. In der *Definitionsphase* wird das Softwaremessungsvorhaben definiert und dokumentiert, d. h. es werden die Ziele, Fragen, Hypothesen und die zu verwendenden Softwaremaße festgelegt. Hierbei wird auch der GQM-Plan erstellt.

3. Während der *Datensammlungsphase* findet die eigentliche Erhebung der einzelnen Softwaremaße statt. Die erfassten Maße werden in einer Datenbank gespeichert. In [SB99] finden sich Anleitungen, wie die Datensammlung geplant werden kann, d. h. wie organisatorisch festgelegt werden kann, durch wen, wie und wann bestimmte Softwaremaße erhoben werden sollen.

4. Schließlich werden in der *Interpretationsphase* die Messergebnisse verwendet, um die gestellten Fragen zu beantworten. Anschließend wird mittels dieser Antworten überprüft, inwieweit die ursprünglich gestellten Ziele erreicht wurden. Die Diskussion der Ergebnisse findet gemäß [SB99] in sogenannten *Feedback Sessions* statt.

2.2.1.2 Kritik am Top-Down-Ansatz der GQM-Methode

Die von Basili, Rombach und Kollegen entwickelte GQM-Methode wurde von der Softwareindustrie bereitwillig aufgenommen und vielfach angewandt (vgl. [HF94]),

so dass die meisten publizierten Softwaremessungsvorhaben zumindest ein Lippenbekenntnis bzgl. der Anwendung der GQM-Methode ablegen (vgl. [FN99]). Dennoch gibt es an der beschriebenen Top-Down-Vorgehensweise auch einige Kritik (vgl. [Het93], [BN95]).

Probleme bei der Zieldefinition

Als grundsätzliches Problem für jeden Top-Down-Ansatz wird gesehen, dass das zu lösende Problem hinreichend genau definiert sein muss, um es durch Verfeinerung in kleinere, leichter lösbare Teilaufgaben zerlegen zu können (vgl. [BN95]). Es wird argumentiert, dass insbesondere im Bereich des Software-Engineerings kein hinreichend genaues Wissen über die Prozesse und die zu erstellenden Produkte vorhanden sei, um die GQM-Ziele korrekt definieren zu können. Aus diesem Grund würden Softwaremanager unter Umständen unsinnige oder nicht erreichbare Ziele definieren.

Probleme bei der Erfassung der Softwaremaße

Insbesondere wird als Problem angeführt, dass nicht für alle Ziele bzw. Fragen effiziente Techniken der Softwaremessung zur Verfügung stünden. Dies könne zu dem Versuch führen, Softwaremaße auf Bereiche anzuwenden, für die sie nur wenig geeignet sind, da sie zu subjektiv, schlecht definiert oder zu schwierig zu erheben sind (vgl. [BN95]). Es gebe Ziele und Fragen, die relativ einfach zu definieren aber extrem schwierig nachzuprüfen und zu messen seien. Praktiker würden in diesen Fällen bald erkennen, dass die Messergebnisse nur Teilaspekte der Ziele addressierten, und somit befürchten, dass die Messergebnisse nicht korrekt interpretiert werden würden (vgl. [Het93]). So wird auch in [Sim01] die explizite Benennung von Interpretationshilfen und die Darstellung von Abhängigkeiten zwischen Qualitätsmerkmalen und Softwaremaßen vermisst.

Strikte hierarchische Dekomposition

Auch macht es der Top-Down-Ansatz schwierig, Softwaremaße oder Fragestellungen, die sich in anderen Organisationen oder Projekten bewährt haben, zu integrieren. Die Bottom-Up-Informationen stehen in Konflikt mit der strikt hierarchischen Dekomposition von Zielen zu Softwaremaßen. Damit ist es schwierig, die mittels der GQM-Methode abgeleiteten Fragen und Softwaremaße gegen andere, evtl. bewährtere auszutauschen (vgl. [BN95]).

Uneinheitliche Bestimmung von Messgrößen

Weiter wird angeführt, dass die Konstruktion von GQM-Hierarchien zu schlechten Definitionen von zu messenden Attributen und Softwaremaßen führen könne. Hierbei könnten bei verschiedenen Projekten im Rahmen des GQM-Prozesses unterschiedliche Definitionen und Softwaremaße für die gleichen Attribute aufgestellt werden. Dadurch werden Vergleiche von Projekten – ein wichtiger Aspekt der Softwaremessung – verhindert (vgl. [BN95]).

Fixe Anzahl von Verfeinerungsebenen

Bzgl. der GQM-Hierarchien wird von Simon angeführt, dass die fixe Anzahl von drei Verfeinerungsschichten für viele Anwendungen zu restriktiv sei, und plädiert für die Verwendung detaillierterer Verfeinerungsketten. Hierbei habe sich in der Praxis das Vorgehen bewährt, die Ziele der GQM-Methode in hierarchisch angeordnete *Subgoals* zu verfeinern, die ihrerseits wiederum durch *Questions* und *Sub-Questions* hin zu Maßen verfeinert werden (vgl. [Sim01]). Allerdings wird durch eine hierarchische Anwendung der GQM-Methode das oben angeführte Problem der schlechten Projektvergleichbarkeit auf Grund unterschiedlicher Softwaremaßdefinitionen nicht kleiner.

Der Umstand, dass die GQM-Methode, welche eigentlich für die Umsetzung von

Messzielen entwickelt worden ist (siehe Abschnitt 2.2.1.1), in der Praxis letztendlich Managementziele als Ausgangspunkt für die Softwaremessung verwendet, verleite zu der Ansicht, dass Softwaremaße nicht unbedingt einer strikten, mathematischen Definition bedürfen. Deshalb seien formlose und ungenaue Definitionen an der Tagesordnung, da von Seiten des Managements die Zeit für eine genauere Spezifikation fehle. Darüber hinaus seien die vom Management gesetzten Ziele häufig von kurzfristigen Zwängen beeinflusst, während durch Softwaremessung eher langfristig orientierte Verbesserungsziele unterstützt werden können (vgl. [BN95]).

Taktische vs. strategische Ziele

Durch die Anknüpfung der Softwaremessung an Managementziele bestehe darüber hinaus die Gefahr, dass die Messergebnisse derart manipuliert werden würden, dass sie entsprechende „Fortschritte“ dokumentierten bzw. Defizite verschleierten (vgl. [Het93]).

Das Hauptproblem der GQM-Methode ist jedoch sicherlich, erst einmal die „richtigen“ Ziele zu finden und zu dokumentieren. Viele Firmen müssten zunächst einmal Softwaremessungsvorhaben durchführen, um den aktuellen Zustand herauszufinden (vgl. [Het93]), da dies eine notwendige Voraussetzung dafür ist, um überhaupt sinnvolle Ziele definieren zu können (vgl. [BN95]).

2.2.2 Bottom-Up-Strategien zur Softwaremessung

2.2.2.1 Das MQG-Verfahren

In der Praxis haben die geschilderten Probleme zur Entwicklung eines umgekehrten GQM-Prozesses geführt, bei dem mittels eines Bottom-Up-Vorgehens zunächst der Fokus darauf gelegt wird, die aktuellen Entwicklungsprozesse und die dabei entwickelten Produkte zu verstehen (vgl. [BN95]).

Sowohl von Hetzel [Het93] als auch von Bache und Neil [BN95] wird dazu das MQG-Verfahren (*Metric/Question/Goal*) vorgeschlagen, welches gemäß folgender Schritte abläuft:

1. **Metrics:**
   Zunächst werden Softwaremaße erhoben und die Ergebnisse analysiert. Der Zweck der Softwaremessung liegt dabei in der Stimulierung von Fragen und der Bereitstellung von Einblicken in die Entwicklungsaktivitäten (vgl. [Het93]).

2. **Questions:**
   Anschließend wird untersucht, wie die erhaltenen Ergebnisse zu erklären sind und ob es Verbesserungsbedarf gibt. Außerdem wird geprüft, ob die Messwerte von den Erwartungen bzw. Vorstellungen abweichen.

3. **Goals:**
   Da nun mögliche Probleme oder Mängel bekannt sind, können entsprechende (Verbesserungs-)Ziele aufgestellt werden.

Dem ersten Schritt geht dabei das implizit formulierte Ziel voraus, den aktuellen Zustand herauszufinden. Dieses Ziel unterscheidet sich von denen des dritten Schritts

und von denen der GQM-Methode dadurch, dass es sich eben nicht um ein Verbesserungsziel handelt (vgl. [BN95]), sondern die Voraussetzungen für die Definition von Zielen schaffen soll.

Spiralartige Vorgehensweise

Beiden Ansätzen gemeinsam ist auch, dass das Verfahren zyklisch angewendet werden soll. Hierbei geht Hetzel jedoch nur von einer spiralartigen Vorgehensweise aus, bei denen die festgelegten Ziele unmittelbar zu Veränderungen in den Abläufen führen und somit entsprechend überarbeitete Softwaremaße bedürfen (vgl. [Het93], S. 30). Demgegenüber beschreiben Bache und Neil auch eine alternierende Anwendung des Verfahrens. Hier startet das Verfahren zwar zunächst ebenfalls mit der Erfassung von Softwaremaßen und führt über die Fragen zu Verbesserungszielen, anschließend können aber auch von den Zielen gemäß der GQM-Methode neue Fragen und schließlich zu erhebende Softwaremaße abgeleitet werden (vgl. [BN95], S. 67). Diese alternierende Vorgehensweise wird insbesondere empfohlen, wenn durch Anwendungen des einfachen MQG-Verfahrens in der betroffenen Organisation bereits entsprechende Erfahrungen auf dem Gebiet der Softwaremessung gewonnen wurden.

2.2.2.2 Geeignete Softwaremaße für das MQG-Verfahren

Die Entwickler des MQG-Verfahrens betonen, dass verschiedene Merkmale bei den Softwareprozessen und Produkten immer wieder auftauchen würden, auch wenn sich Software-Organisationen hinsichtlich ihrer Entwicklungsmethoden, Programmiersprachen und Managementkultur unterscheiden mögen.

Einschlägige Softwaremaße

Bache und Neil schlagen deshalb vor, diese gemeinsamen Merkmale mit Softwaremaßen, welche von ihnen als „einschlägig“ bezeichnet werden, zu erfassen. Dabei werden die *einschlägigen Softwaremaße* folgendermaßen charakterisiert (vgl. [BN95], S. 65):

- Sie sind wohldefiniert, objektiv und erfassen wohlverstandene Merkmale.
- Es werden Merkmale gemessen, welche nachgewiesenermaßen einen Kosten- oder Qualitätsaspekt repräsentieren.
- Sie sollten einfach zu erheben und soweit möglich mittels Werkzeugunterstützung automatisiert erfassbar sein.
- Sie sollten entweder durch Anwendung in der Industrie oder durch wissenschaftliche Untersuchungen hinreichend erprobt sein.

Grundmaße

Hetzel hingegen definiert einige „Grundmaße“, welche für jedes am Softwareentwicklungsprozess beteiligte Arbeitsprodukt erhoben werden sollten. Diese Grundmaße werden in Input-, Output- und Ergebnismaße gegliedert (vgl. [Het93], S. 29):

- **Input-Maße:**
  Informationen über Ressourcen, Aktivitäten oder andere Arbeitsprodukte, die zur Erstellung eines bestimmten Arbeitsproduktes aufgewendet wurden.

- **Output-Maße:**
  Maße, welche die erzeugten Arbeitsprodukte beschreiben oder quantifizieren.

- **Ergebnismaße:**
  Die Ergebnismaße sollen laut Hetzel „*the usage and effectiveness (perceived and actual) of the deliverables and work products in fulfilling their requirements*“ quantifizieren. Diese Eigenschaften werden üblicherweise unter dem gebräuchlicheren Begriff der *Produktqualität* subsumiert.

Hetzel beschreibt für verschiedene Arten von Arbeitsprodukten, welche er in die Kategorien *Spezifikation/Design*, *Code/Implementierung* und *Test/Evaluierung* unterteilt, geeignete Grundmaße gemäß seiner Typisierung nach Input, Output und Ergebnis (vgl. [Het93], S. 87 ff.).

2.2.3 Bidirektionale Strategien

Eine bidirektionale Strategie, bei der zwischen einer Management-Sicht und einer operativen Sicht auf die Softwaremessung unterschieden wird, ist in [SK06] beschrieben. Diese Arbeit basiert auf dem Forschungsprojekt *IndexQBench* ([QBe08]), welches „Entwicklung und Einsatz eines ganzheitlichen Ansatzes zur konstruktions- und evolutionsbegleitenden Sicherung der inneren Qualität von objektorientierter Software“ als Ziel hat, und dessen Ergebnisse in [SSM06] in Form eines Leitfadens veröffentlicht wurden.

Ganzheitlicher Qualitätsbegriff

Dabei ist auch der Qualitätsbegriff des QBench-Projekts „ganzheitlich“ zu verstehen. Dieser entspricht der Sichtweise von Garvin [Gar84], gemäß dessen Definition des *wertorientierten* und des *fertigungsbezogenen Qualitätsverständnis* sich der Qualitätsbegriff nicht nur auf das Produkt sondern auch auf den gesamten Herstellungsprozess erstreckt. Daher betrachtet das QBench-Projekt nicht nur die Software-Qualität im engeren Sinn, sondern die Ansätze lassen sich auf alle Merkmale des Softwareentwicklungsprozesses anwenden. Insbesondere wird in [SK06] auch explizit ein Indikator für den *Fertigstellungsgrad*, d. h. ein Indikator für eine Projekt-Eigenschaft, aus dem nachfolgend dargestellten *bidirektionalem Qualitätsmodell* abgeleitet.

Somit wollen wir im Weiteren auch von einer *bidirektionalen Strategie zur Softwaremessung* sprechen. Genauso wollen wir mit den in [SSM06] und [SK06] verwendeten Begriffen *Qualitätseigenschaft* und *Qualitätsmerkmal* verfahren und nur von *Eigenschaften* und *Merkmalen* sprechen. In der genannten Literatur wird der Zusatz „Qualität“ in Abgrenzung zu den allgemeineren Begriffen verwendet, um die Unterscheidbarkeit der Eigenschaften bzw. der Merkmale deutlich zu machen.

Qualitäts-eigenschaft

Nach [SSM06] ist eine *Qualitätseigenschaft* eine Eigenschaft, die zur Unterscheidung von Produkten, Bausteinen oder Herstellungsprozessen in qualitativer (subjektiver) Hinsicht herangezogen werden kann.

Die Wahrnehmung von Qualitätseigenschaften ist immer subjektiv bzw. unscharf, da eine gewisse Interpretation des Wahrnehmenden (z.B. bei der Wartbarkeit oder der Verstehbarkeit) erfolgt. Damit ist ihre Relevanz auch vom jeweiligen Kontext abhängig. Eigenschaften lassen sich erst durch Operationalisierung eindeutig erfassen

und bewerten. Dazu muss Konsens hinsichtlich der Bedeutung einzelner Eigenschaften einer Entität bestehen (vgl. [SQS05]).

Qualitätsmerkmal

In [SSM06] wird ein *Qualitätsmerkmal* als Merkmal definiert, welches zur Unterscheidung der betrachteten Entitäten herangezogen werden kann. Ein Qualitätsmerkmal ist somit eine Eigenschaft, welche eine Entität aus sich selbst heraus objektiv besitzt (z. B. Software besteht aus *Lines of Code*) (vgl. [SQS05]).

Die der bidirektionalen Betrachtungsweise zugrunde liegenden Überlegungen sind, dass das IT-Management weniger an konkreten Softwaremaßen und deren Werten als an einer Einschätzung der Erfüllung strategischer Ziele interessiert ist. Demgegenüber sind auf operativer Ebene eher konkrete Softwaremaße von Interesse, wobei jedoch die einzelnen Softwaremaße jeweils nur für einen kleinen Adressatenkreis einen Mehrwert darstellen. Wie in [SK06] dargestellt wird, korreliert im Allgemeinen der Adressatenkreis eines Softwaremaßes eng mit dem Typ der bei der Softwaremessung betrachteten Artefakte. So sind z. B. Code-Maße hauptsächlich für den Entwickler interessant, während dagegen Test-Maße eher für die Qualitätssicherung von Bedeutung sind.

Adressatenkreis der Softwaremetrie

Mittels einer bidirektionalen Sichtweise soll nun die Traceability zwischen Softwaremaßen und IT-Strategien hergestellt werden. Dabei bedient man sich in der Management-Sicht einer der GQM-Methode angelehnten Verfeinerung abstrakter IT-Strategien hin zu Eigenschaften des Entwicklungsprozesses oder der erstellten Produkte. Auf der operativen Ebene werden Merkmale vorhandener Daten und Artefakte mittels Softwaremaße repräsentiert. Hier werden ähnlich wie beim MQG-Verfahren Softwaremaße erhoben und anschließend wird versucht, diese zu interpretieren, um mögliche Probleme oder Mängel zu erkennen.

Qualitätsindikatoren

Als Schnittstelle zwischen den aus der strategischen Sicht abgeleiteten Prozess- und Produkteigenschaften und den Softwaremaßen der operativen Sichtweise fungieren sogenannte *Qualitätsindikatoren* (siehe Abbildung 2.4), die bei bestimmten Ausprägungen der Softwaremaße Rückschlüsse auf bestimmte Eigenschaften erlauben. So ist z. B. nach [SK06] „eine sinkende Fehlererkennungsrate bei gleich bleibendem Testaufwand und sinkender Code-Änderungsrate ein guter Indikator für einen zunehmenden Fertigstellungsgrad“ (siehe [SK06], S. 43).

Die Qualitätsindikatoren repräsentieren somit einen Konsens bzw. eine Interpretationsvorschrift, wie sich Werte und Wertetupel üblicherweise verhalten und welche Implikationen Abweichungen von dieser Erwartungshaltung besitzen.

Bei der bidirektionalen Vorgehensweise wird also zum einen versucht, aus strategischen Zielen mittels Verfeinerung Eigenschaften abzuleiten, welche die Prozesse und Produkte erfüllen müssten, damit die Ziele erfüllt werden. Auf der anderen Seite wird untersucht, ob man diese Eigenschaften mittels der vorhandenen Softwaremaße operationalisieren kann. Die Qualitätsindikatoren stellen somit eine mögliche Zuordnung zwischen den Softwaremaßen und den Prozess- und Produkteigenschaften dar.

Eine konkrete, individuelle Instantiierung dieser bidirektionalen Strategie zur Softwaremessung ist nach [SK06] in jedem Fall das Ergebnis eines projektspezifischen Abstimmungsprozesses. In [SSM06] wird betont, dass sich eine bestimmte Eigen-

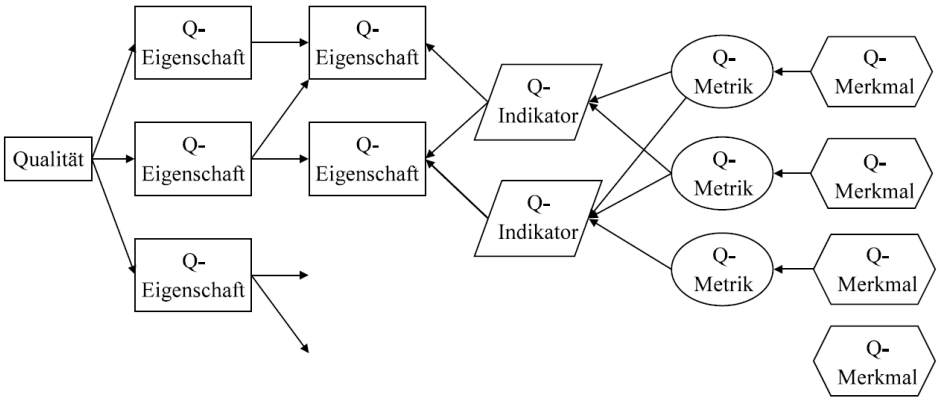


Abbildung 2.4: Struktur des bidirektionalen Qualitätsmodells (vgl. [SK06])

schaft in der Regel erst durch die Aggregation verschiedener Qualitätsindikatoren zuverlässig ausdrücken lasse, d. h. es existierten deutlich mehr Qualitätsindikatoren als Eigenschaften. Da verschiedene Qualitätsindikatoren häufig unterschiedlich starken Einfluss auf die Qualitätseigenschaften besäßen, sei es häufig ratsam, eine Gewichtung der Qualitätsindikatoren vorzunehmen, so dass sich eine konkrete Eigenschaft aus der gewichteten Summe der sie beeinflussenden Qualitätsindikatoren ergibt. Dabei müsse jeder Einfluss separat begründet werden, damit die Nachvollziehbarkeit des bidirektionalen Modells gewahrt bleibe (vgl. [SSM06]).

Aufbauend auf diesen Überlegungen wurde in [SSM06] ein Katalog mit 52 Qualitätsindikatoren vorgestellt. Damit soll die Qualität der erzeugten Softwareprodukte mittels eines sogenannten *Code-Quality-Index* messbar gemacht werden, welcher auf einer standardisierten Menge dieser Qualitätsindikatoren basiert. Entgegen unserer obigen Anmerkung, dass dieser bidirektionale Ansatz allgemein auf den gesamten Softwareentwicklungsprozess anwendbar ist, beschränken sich die in [SSM06] vorgeschlagenen Qualitätsindikatoren jedoch auf reine Code-Eigenschaften.

Kombination der GQM- und MQG-Methode

Letztendlich stellt also der bidirektionale Ansatz eine Kombination der GQM- und der MQG-Methode dar. Aus Sicht des Managements werden wie bei der GQM-Methode von Management-Zielen mittels Verfeinerung Eigenschaften abgeleitet. Dabei werden nur solche Eigenschaften betrachtet, die innerhalb des jeweiligen Projektes relevant sind. Nun wird aber nicht wie bei der GQM-Methode versucht, zu diesen Eigenschaften passende Softwaremaße zu finden. Damit umgeht man die in Abschnitt 2.2.1.2 dargestellte Problematik, dass nicht immer für alle Ziele bzw. Fragen effiziente Techniken der Softwaremessung zur Verfügung stehen. Stattdessen werden gemäß dem MQG-Ansatz diejenigen Merkmale betrachtet, die im jeweiligen Kontext überhaupt ermittelbar sind (vgl. [SSM06]). Gleichzeitig wird jedoch auch das bloße Anwenden von Metrikwerkzeugen ohne Zielvorstellung vermieden, da eine derartige Vorgehensweise in der Literatur als nicht erfolgversprechend angesehen wird (vgl. [FN99]).

Laut [SSM06] ermöglicht die bidirektionale Vorgehensweise somit eine maximale Berücksichtigung von Projektspezifika, da sie systematisch Gewünschtes (in Form von Eigenschaften) mit Möglichem (in Form der messbaren Merkmale) kombiniere. Die strikte Trennung zwischen technikabhängigen und managementabhängigen Teilen erlaube unternehmensweite, technikunabhängige Zielvorgaben, die für jedes System im Unternehmen mittels jeweils spezifischer Qualitätsindikatoren ermittelt werden könnten.

2.3 Softwaremaße auf Managementebene

Unabhängig von den genannten Strategien zur zielgerichteten Auswahl von Softwaremaßen gibt es einige Kerngrößen in der Softwareentwicklung, die regelmäßig in der Literatur als untersuchenswert dargestellt werden. Dazu gehören insbesondere die Beurteilung

- des Umfangs des Softwareprojektes,
- des Projektaufwands und somit der Projektkosten,
- der Qualität,
- der kalendermäßigen Zeitdauer und
- der Produktivität.

Five Core Metrics

Deshalb werden diese Messgrößen auch als die *„Five Core Metrics“* (vgl. [PM02], [PM03]) oder als *„Key Measures“* (vgl. [Rus02]) bezeichnet.

Auch wird regelmäßig darauf verwiesen, dass man sich insbesondere bei der Einführung von Maßnahmen zu Softwaremessung auf wenige Softwaremaße beschränken sollte, um diese konsistent erfassen zu können (vgl. [Rus02], [KO06]).

Kernattribute

In dieser Arbeit sollen obige Messgrößen jedoch als *Kernattribute* bezeichnet werden, da einige dieser Attribute im Kontext verschiedener Entitäten untersucht werden können. Insbesondere kann sich z. B. die Qualität auf das erstellte Produkt, auf die Projektdurchführung oder auf den bei der Projektdurchführung angewandten Prozess beziehen. Genauso lässt sich die Produktivität in Bezug auf einzelne Mitarbeiter, aber auch für ganze Projekte bestimmen. Welche Softwaremaße bezüglich dieser Kernattribute dann erhoben werden können, soll im Weiteren dargestellt werden.

„A defect is anything that results in customer dissatisfaction“ (siehe [MK00])

Im Rahmen der Six-Sigma-Vorgehensweise wird eine „Nullfehler-Strategie“ bei der Softwareentwicklung angestrebt (vgl. [Feh05], S. 20 ff.). Dabei gelten als Fehler oder Mängel alle während des Projektverlaufs auftretenden Unzulänglichkeiten, die beim Kunden zu Unzufriedenheit führen (vgl. [MK00]). Dazu gehören neben mangelnder Qualität des erstellten Produktes auch Lieferverzögerungen oder die Nichteinhaltung von Kostenschätzungen. Da Lieferverzögerungen letztendlich über die kalendermäßige Zeitdauer des Entwicklungsprojektes und Kostenüberschreitungen über den Aufwand kontrolliert werden können, ist somit auch für diese Quellen möglicher Mängel eine Überwachung durch Messung obiger *Kernattribute* möglich.

2.3.1 Messung des Softwareumfangs

LOC und KDSI

Seit den Anfängen der Softwaremetrie wurden die *Anzahl der Programmzeilen* (LOC) oder das vergleichbare Maß der *„kilos of delivered source instructions“* (KDSI) als Größenmaß für den Umfang von Softwareprojekten verwendet (vgl. [FN99]) und dabei auch in frühen Modellen zur Aufwandsschätzung wie *COCOMO* (vgl. [Boe81]) oder *SLIM* (vgl. [Put78]) eingesetzt.

Kritik am LOC-Maß

Es zeigte sich jedoch, dass das LOC-Softwaremaß mehrere Unzulänglichkeiten hinsichtlich der Größenmessung aufwies, welche insbesondere mit der steigenden Vielfalt an Programmiersprachen deutlich wurden. So sind die Anzahl der Codezeilen einer Assemblersprache nicht mit denen einer Hochsprache vergleichbar. Aber auch innerhalb einer Programmiersprache hängt die Anzahl der zur Implementierung einer bestimmten Funktionalität benötigten Programmzeilen oder Programminstruktionen stark vom Stil des jeweiligen Programmierers ab. Hinzu kommt, dass es auch unterschiedliche Meinungen darüber gibt, ob Leer- oder Kommentarzeilen mitzuzählen sind. So sind diese beispielsweise für die Code-Verständlichkeit im Rahmen von Wartungsarbeiten durchaus wertvoll. Und in Sprachen wie *Java* können Kommentarzeilen sogar Annotationen enthalten, welche die Code-Generierung beeinflussen. Ein weiterer Kritikpunkt für die genannten Maße ist, dass beim Einsatz entsprechender graphisch orientierter Entwicklungsumgebungen bzw. bei der Verwendung vorgefertigter Softwarekomponenten der Anteil der manuell erzeugten Codezeilen von immer geringerer Bedeutung ist.

LOC als Normalisierungsmaß

Trotz seiner Schwächen wird das LOC-Softwaremaß immer noch regelmäßig angewendet (vgl. [FN99]). So dient es zur Normalisierung der Fehlerhäufigkeit durch Angabe der *Fehler pro KLOC* (Anzahl von Fehlern auf 1000 Codezeilen) und zur Abschätzung der Produktivität (*LOC pro Personenmonat*). Daneben ist es nach wie vor ein zwar grobes aber einfach zu ermittelndes Maß für den Programmumfang, da die meisten Programmierer eine Vorstellung davon haben, welchen Umfang ein Programm von z. B. 10 KLOC in Bezug auf eine bestimmte Programmiersprache hat.

NCSS

Ähnlich den LOC kann auch das Maß *NCSS (non-commenting source statements)* eingesetzt werden, bei dem die Programmiersprachen-Statements ohne Kommentare aber einschließlich der Compiler-Anweisungen, Datendeklarationen und ausführbaren Anweisungen gezählt werden (vgl. [GC87], S. 58, [Gra92], S. 233). Durch diese exaktere Definition und die Zählung von Programmiersprachen-Statements ist dieses Maß auch bei moderneren Sprachen einsetzbar, bei denen eine Programmzeile oder auch ein einzelnener Ausdruck mehrere Anweisungen enthalten kann. Grady empfiehlt dieses Softwaremaß, mit dem er bei *Hewlett Packard* umfangreiche Erfahrungen gesammelt hat (vgl. [GC87]), insbesondere für Darstellungen des Projektstatus auf Managementebene (vgl. [Gra92], S. 91 ff.). Die Gründe für die Verwendung dieses Maßes sind einerseits die automatisierbare Messbarkeit und die Verwendbarkeit als Normalisierungsmaß insbesondere in Verbindung mit der Messung von Fehlern. Außerdem entwickelt sich (ähnlich wie bei den oben genannten LOC) mit der Zeit innerhalb der Organisation eine Vorstellung davon, welchen Umfang ein Projekt mit

einer in NCSS angegebenen Größe hat. Dazu ist jedoch erforderlich, die Größenmessungen regelmäßig und automatisiert durchzuführen, um auf diese Weise eine Erfahrungsdatenbank aufzubauen.

Halsteads und McCabes Softwaremaße

Wegen der Ungenauigkeiten und der Abhängigkeit von der verwendeten Programmiersprache des LOC-Softwaremaßes wurden ab Mitte der 1970er Jahre weitere Softwaremaße entwickelt, welche die Bewertung des Umfanges und der Komplexität von Software ermöglichen sollten (vgl. [FN99]). Als Beispiele seien hier die *software science measures* von Halstead ([Hal77]) und die Komplexitätsmaße von McCabe ([McC76]) genannt. Nach Halstead können der Programmieraufwand und die Implementierungszeit aus den Maßen *Programm-Länge* und *Programm-Volumen* abgeleitet werden, welche aus der Anzahl und der Verwendungshäufigkeit der Operanden und Operationen der Programmiersprache berechnet werden (vgl. [Zus98], S. 59 f.). McCabe berechnete unter Anwendung von Methoden der Graphentheorie im Kontrollflussgraphen eines Programmes eine *minimale Anzahl von Pfaden*, welche er als *Cyclomatische Komplexität* der Software interpretierte (vgl. [Zus98], S. 60).

Die Softwaremaße LOC, NCSS, Halstead-Länge bzw. -Volumen und teilweise auch McCabes Software-Komplexität können erst ermittelt werden, wenn der Quellcode der Software bereits vorliegt. Damit ist ihre Verwendbarkeit zur vorausschauenden Abschätzung des Programmieraufwandes eingeschränkt. Trotzdem werden sie im Original-COCOMO-Modell genau zu diesem Zweck verwendet (vgl. [Boe81]).[3] In der Weiterentwicklung zu COCOMO II (vgl. [B$^+$00]) ist es jedoch möglich, die Aufwandsschätzung auf Basis sogenannter *Object-Points*[4] durchzuführen.

Function-Point-Methode

Im Jahr 1979 veröffentlichte Albrecht die *Function-Point-Methode*, welche es ermöglichte, die Anforderungen an die zu entwickelnde Software mit sogenannten *Function Points* zu bewerten und daraus den Projektaufwand abzuleiten ([Alb79], [AG83]). Dabei wird der Umfang der Anwendung gemäß der vom Kunden geforderten Funktionalität gemessen. Diese Funktionalität wird vom Kunden (z. B. bei den Fachabteilungen) erfragt und anhand eines logischen Systementwurfs quantifiziert. Das Ergebnis ist unabhängig von der für die Implementierung verwendeten Technologie, da die technische Realsierung der Anwendung nicht betrachtet wird (vgl. [BF00], S. 179).

IFPUG

Die Function-Point-Methode wurde im Laufe der Zeit beständig weiterentwickelt, so dass es heute mehrere Function-Point-Zählmethoden gibt. Die bekannteste davon ist wohl die von der *International Function Point Users Group* derzeit in der Version IFPUG 4.2 [IFP04] veröffentlichte.

Funktionstypen

Bei der Function-Point-Analyse (FPA) werden im Prinzip für die zu entwickelnde Anwendung die Zugriffe auf logische Datenbestände und die Durchführung von Transaktionen herausgearbeitet. Diesen als *Funktionstypen* bezeichneten Funktionen der Anwendung (vgl. Tabelle 2.3) werden Daten-Function-Points bzw. Transaktions-Function-Points zugeordnet.

[3] Die Beispiele in der Beschreibung zum Original-COCOMO-Modell beziehen sich auf die Programmierung von Großrechner-Software, wo eine Abschätzung der (COBOL-)Programmzeilen eher möglich ist, als bei modernen objekt-orientierten Programmiersprachen.

[4] Diese sind nicht identisch mit denen der Object-Point-Methode nach H. Sneed (vgl. S. 36).

Daten-Function-Points	**Internal Logical Files (ILF)** Interne Datenbestände, die innerhalb der Systemgrenzen der Anwendung gepflegt werden
	External Interface Files (EIF) Externe Schnittstellen-Datenbestände, die außerhalb der Systemgrenzen (von anderen Anwendungen) gepflegt werden
Transaktions-Function-Points	**External Inputs (EI)** Externe Eingabedaten mit ihren logischen Datengruppen und Datenelementen
	External Output (EO) Externe Ausgabedaten mit ihren logischen Datengruppen und Datenelementen
	External Inquiries (EQ) Externe Abfragen mit ihren logischen Datengruppen und Datenelementen

Tabelle 2.3: Bei der FPA bewertete Funktionstypen (vgl. [BF00], S. 190 f.)

FTRs/DETs	**1–4 DETs**	**5–15 DETs**	**> 15 DETs**
0–1 FTR	gering	gering	mittel
2 FTR	gering	mittel	hoch
> 2 FTR	mittel	hoch	hoch

Tabelle 2.4: Komplexitätsgrade für externe Eingaben nach IFPUG 4.2 ([IFP04])

Die Komplexität der Funktionstypen wird dabei anhand einer Komplexitätsmatrix in „gering“, „mittel“ oder „hoch“ eingestuft und entsprechend mit einer unterschiedlichen Anzahl Funktion-Points bewertet.

Beispiel 3 (Function-Points) *Ein Kundenverwaltungssystem soll eine Eingabemaske zum Hinzufügen weiterer Datensätze erhalten. Diese Transaktion stellt aus der Sicht der FPA einen EI-Funktionstyp dar, da es sich um externe Eingabedaten handelt.*

*Nun werden die Datenelemente (Name, Vorname, Telefonnummer, usw.) gezählt und diese Anzahl mit DET (*data element type*) bezeichnet. In diesem Fall gelte beispielsweise* $DET = 8$*. Anschließend wird die Anzahl FTR der referenzierten logischen Datenbestände (*file type records*) ermittelt. Diese gehen in diesem Fall mit* $FTR = 1$ *in die Berechnung ein, da logisch nur auf den Datenbestand „Kundendaten“, in denen der neue Datensatz abgelegt wird, zugegriffen wird. Dass die Kundenstammdaten und die Kundenadressdaten evtl. in unterschiedlichen Tabellen abgelegt sind, wird an dieser Stelle nicht berücksichtigt.*

Mit $DET = 8$ *und* $FTR = 1$ *wird gemäß Tabelle 2.4 die Komplexität dieser EI-Transaktion als „gering“ eingestuft und in diesem Fall mit 3 Function-Points bewertet.*

Ungewichtete und gewichtete Function-Points

Die so ermittelten Function-Points werden als *ungewichtete Function-Points* bezeichnet. Anhand von 14 Einflussfaktoren (z. B. *Datenkommunikation*, *Transakti-*

onsrate, *Benutzerfreundlichkeit*, etc.) wird ein sogenannter *Value Adjustment Factor* (VAF) berechnet, der dann schließlich zu den *gewichteten Function-Points* führt. Der Schritt von den ungewichteten Function-Points zu den gewichteten entspricht der Unterscheidung zwischen dem Umfang der zu entwickelnden Anwendung und dem damit verbundenen Aufwand, da bei diesem auch noch weitere Einflüsse auf die Systementwicklung (z. B. Erfahrung der Mitarbeiter, Klarheit der Anforderungen, Prozessreife, etc.) berücksichtigt werden müssen (vgl. [BF00], S. 192, [BF00], S. 22 f.). Allerdings wird auch angeführt, dass diese VAF stark abhängig von der Beurteilung von Experten seien und deshalb wenig für konkrete und eindeutige Aussagen zu den Kosten einer Software geeignet wären. Daher könne man alternativ auch mittels der FPA nur den Umfang der Software in Form von ungewichteten Function-Points bestimmen und mittels parametrisierbarer Modelle, die für verschiedene Industrien entsprechend angepasst sind, ähnlich wie beim COCOMO-II-Verfahren ([B+00]) auf den Aufwand und die Kosten schließen (vgl. [Feh05], S. 170).

Value Adjustment Factor

Die Function-Point-Methode wird zwar nur von einem Teil der industriellen Praktiker verwendet (vgl. [Feh05], S. 166 f.), jedoch ist die Aussagekraft ihrer Ergebnisse grundsätzlich anerkannt (vgl. [HF94]). Leider erfolgt das Zählen der Function-Points grundsätzlich manuell und darf nach Meinung vieler Function-Point-Anwender auch nicht automatisiert werden, da die Zählung auf den Benutzeranforderungen an die Software basiert und jeder Zählschritt dokumentiert und begründet werden muss (vgl. [BF00], S. 132).

Manuelle Zählung der Function-Points

In der Literatur wird berichtet, dass diese Einschränkung für die *COSMIC Full Function Points* (CFFP, [ADO+03]) nicht mehr gelte (vgl. [Feh05], S. 171). Bei dieser Variante der Function-Point-Methode, welche in Konkurrenz zur IFPUG-Version steht, werden Datenbewegungen gezählt, was auch durch die Analyse von UML-Diagrammen möglich sein soll. Auch seien CFFP besser für die Analyse aktueller Multi-Tier-Systeme und Komponenten-basierender Anwendungen geeignet. In diesen seien die für die FPA verwendeten Zählgrößen nur schwer zu finden, da diese noch aus der Mainframe-Programmierung stammten (vgl. [Dek04]).

COSMIC Full Function Points

Ebenfalls den Bedürfnissen der objekt-orientierten Programmierung Rechnung tragend wurde von Sneed die *Object-Point-Methode* [Sne96] veröffentlicht. Dabei wird das objekt-orientierte Design der zu implementierenden Anwendung mit sogenannten *Class-*, *Message-* und *Process-Points* bewertet, welche mit dem Klassen-Entwicklungsaufwand, dem Integrationsaufwand bzw. mit dem Systemtestaufwand korresponieren. Das gesamte Aufwandsmaß *Object-Points* ergibt sich als Summe der drei Bestandteile, wobei anschließend noch eine Justierung auf der Basis von Qualitätsanforderungen und Projekt-Einflussfaktoren durchgeführt wird. Grundsätzlich lässt sich die Zählung der Object-Points somit auch automatisieren. Jedoch gibt es kaum industrielle Erfahrungen mit der Anwendbarkeit dieser Methode (vgl. [BF00], S. 211). Auch muss für die Anwendung dieser Methode bereits ein objekt-orientiertes Design-Modell der Anwendung vorliegen, so dass sie in der Angebotsphase eines Softwareprojektes üblicherweise noch nicht eingesetzt werden kann.

Object-Point-Methode

Für den letztgenannten Fall der frühzeitigen Aufwandsschätzung innerhalb industrieller Softwareentwicklungsprojekte erscheint die Aufwandsschätzung mittels

Use-Case-Points (UCP, vgl. [SW01]) als vielversprechende Alternative. Dabei werden die in den Use-Cases auftretenden Akteure und Transaktionen, d. h. Ereignisse zwischen dem Akteur und dem System, bewertet. Es wird berichtet, dass die Methode eine recht genaue Aufwandsschätzung erlaubt, wenn sie in Bezug auf die jeweilige Organisation entsprechend kalibriert wurde (vgl. [ADSJ01], [Car05]). Die Firma *sd&m* berichtet in diesem Zusammenhang von einer eigenen Erweiterung der UCP-Methode, durch die sich die Schätzgenauigkeit für bestimmte Projektarten signifikant verbessern ließ (vgl. [FJE06]).

Use-Case-Points

Schließlich wird in Abschnitt 4.2.4 (S. 98 ff.) mit der Darstellung des *Feature-Driven Developments* eine weitere Möglichkeit gezeigt, den Projektumfang zu spezifizieren. Dabei wird die Anwendung in einzelne *Features* (vgl. [PF02]) zerlegt, welche dann gewichtet und gezählt werden können.

Features

2.3.2 Bestimmung von Aufwand und Kosten

Betriebswirtschaftlich wird unter dem *Aufwand* der bewertete Verbrauch aller Güter und Dienstleistungen in einer bestimmten Periode verstanden (vgl. [RRK06]). Dabei wird zwischen Aufwendungen, die unmittelbar mit der Leistungserstellung im Rahmen des Betriebszwecks zusammenhängen, den sog. *Betriebsaufwendungen*, und den *neutralen Aufwendungen* (z. B. außerordentliche Aufwendungen) unterschieden. Zu den Betriebsaufwendungen zählen dagegen beispielsweise Gehälter, Energie, Lizenzkosten und Abschreibungen. Die Betriebsaufwendungen sind im Wesentlichen deckungsgleich mit den *Kosten* in der Kosten- und Leistungsrechnung.[5]

Aufwand und Kosten

Unter dem Aufwand für ein Softwareentwicklungsprojekt wird dagegen meistens nur der *Zeitaufwand* in Personentagen (oder Personenmonaten) verstanden, der zur Durchführung des Projektes erforderlich ist. Der Projektaufwand ist somit zunächst einmal die tatsächliche Zeit, die alle Mitarbeiter eines Projektes für dieses aufwenden. Dieser Aufwand enthält keinen Urlaub und keinen Krankenstand, sehr wohl aber Schulungen und Reise- bzw. Fahrzeiten (vgl. [BF00]).

Projektaufwand

Für die Berechnung der projektbezogenen Kosten des eigenen Personals wird der erbrachte Projektaufwand mit entsprechenden Stundenverrechnungssätzen multipliziert, welche im Rahmen des betrieblichen Rechnungswesens ermittelt werden (vgl. [Bur02], S. 345 ff.). Dabei wird üblicherweise zwischen internen und externen Stundenverrechnungssätzen unterschieden. Die internen Stundenverrechnungssätze sind ein Maß dafür, welcher betriebswirtschaftliche Aufwand mit dem Einsatz eines Mitarbeiters in einem Projekt verbunden ist. Mittels der externen Stundensätze wird entweder die Tätigkeit der Mitarbeiter dem Kunden in Rechnung gestellt, oder die externen Stundensätze fließen in die Berechnung eines Festpreises bei der industriellen Auftragsvergabe ein.

Stunden-verrechnungssätze

[5] Bei den Abschreibungen kann es beispielsweise zu Differenzen zwischen den bilanziellen Abschreibungen in der Betriebsbuchhaltung und den kalkulatorischen Abschreibungen in der Kostenrechnung kommen.

Aufwandserfassung

Entwicklungs-adäquate Stundenkontierung

Voraussetzung für jede Aufwands- und Kostenkontrolle ist das „entwicklungsadäquate“ Erfassen des Personalaufwands, d. h. eine regelmäßige und vollständige Stundenaufschreibung entsprechend der Projektstruktur und – wenn möglich – auch entsprechend der Prozessstruktur. Um ein wirkungsvolles Projektcontrolling zu ermöglichen, sollte eine Detaillierung des Personalaufwandes nach

- Arbeitspakten,
- Meilensteinen,
- Entwicklungsphasen und
- Tätigkeitsarten

erfolgen (vgl. [Bur02], S. 341).

SAP-Arbeitszeitblatt

Wegen des großen Datenumfangs, der somit bei der Stundenkontierung anfällt, ist eine praktikable Aufwandserfassung i. Allg. nur mit Hilfe eines EDV-gestützten Verfahrens möglich. Als Beispiel sei hier das SAP-Arbeitszeitblatt *SAP CATS* (*cross-application time sheet*, [SAP03]) angeführt. Dabei handelt es sich um eine Anwendung zur manuellen Erfassung von Arbeitsleistungen auf Basis eines SAP ERP-Systems. Dabei können die benötigten Kontierungsobjekte und die anderen Arbeitszeitattribute vom Administrator entsprechend der Organisationsbedürfnisse konfiguriert werden. Die einzelnen Projektmitarbeiter bekommen dann bei der Tätigkeitserfassung nur diese Kontierungsobjekte (Projekte, Arbeitspakete) zur Kontierung angeboten, welche für sie relevant sind.

Qualität der Zeiterfassung

Die Qualität der Stundenaufschreibung wird nach [Bur02] von drei Kriterien bestimmt:

- Genauigkeit
- Vollständigkeit
- Ehrlichkeit

Die *Genauigkeit* und die *Vollständigkeit* könne dabei positiv durch Einbinden der Kontenstruktur in einen Netzplan beeinflusst werden, da dadurch eine Plausibilisierung der kontierten Arbeitspakete mit den Netzplandaten möglich ist. Außerdem würden dadurch Kontierungen auf falsche Arbeitspakete und damit auf falsche Konten erheblich verrringert werden.

Die *Ehrlichkeit* würde dagegen ganz entscheidend vom Verhalten der Leitung beeinflusst werden. Denn grundsätzlich sei es auch möglich, dass die kontierten Stunden nicht nur zur Projektbeurteilung, sondern auch zur Mitarbeiterbeurteilung verwendet werden. Daher bestünde zwangsläufig die Gefahr der Manipulation, falls bei Planabweichungen Sanktionen drohten (vgl. [Bur02], S. 345). Eine Möglichkeit, dieses Problem zu mindern, besteht darin, die kontierten Stunden nur auf Team-Ebene auszuwerten.

Häufig sei nach [Bur02] auch ein bewusstes „zu viel Kontieren“ auf einzelne, noch nicht ausgeschöpfte Projektkonten zu beobachten. Dieses Kontieren nach dem „Tragfähigkeitsprinzip“ beeinträchtigt natürlich ebenfalls jede zielorientierte Projektkontrolle (vgl. [Bur02], S. 345).

2.3.3 Die Produktivität

Ökonomen verstehen unter der *Produktivität* üblicherweise das Verhältnis zwischen der produzierten Ausbringungsmenge und den dafür beim Produktionsprozess eingesetzten Mitteln. Bezogen auf die Softwareentwicklung ergibt sich somit die Produktivität als Verhältnis zwischen dem erzeugten Produktumfang und den dafür erbrachten Aufwand:

$$\text{Produktivität} = \frac{\text{Produktumfang}}{\text{Aufwand}}$$

Um die Produktivität tatsächlich bestimmen zu können, müssen in dieser Gleichung noch die Variablen durch entsprechende Softwaremaße ersetzt werden. Oft wird beispielsweise eine der folgenden Definitionen verwendet (vgl. [FP98], S. 408 ff., [Kan03], S. 343 ff., [GC87], S. 19 ff.)

$$\text{Produktivität} = \frac{\text{LOC}}{\text{Personenmonate}} \quad \text{oder} \quad \text{Produktivität} = \frac{\text{NCSS}}{\text{Personenmonate}}$$

LOC-Produktivität

Als Nachteil dieses Softwaremaßes wird gewöhnlich angeführt, dass sowohl für den Zähler als auch für den Nenner die Bestimmung der genannten Softwaremaße problematisch sei (vgl. [FP98], ebd.). Die Messung des Aufwandes in Personenmonaten mache insbesondere die organisationsübergreifende Vergleichbarkeit problematisch, da ein Personenmonat je nach Unternehmen eine unterschiedliche Anzahl von Stunden bedeuten könne. Innerhalb einer Organisation sollte jedoch die Messung des Aufwandes, wie in Abschnitt 2.3.2 beschrieben, in konsistenter Weise möglich sein.

Problematischer ist dagegen, wie im Abschnitt 2.3.1 dargestellt, die Messung des Softwareumfangs, da weder LOC noch NCSS wirklich ein Maß für den Umfang der mit dem Softwareprodukt ausgelieferten Funktionalität darstellen (vgl. [FP98], [Kan03]). Zudem gibt es auch hier wieder Probleme mit der Vergleichbarkeit über verschiedene Projekte und Organisationen hinweg. So formuliert Jones, dass „using lines of code for productivity studies involving multiple languages and full life cycle activities should be viewed as professional malpractice“ (siehe [Jon00], S. 72).

Auch kann es vorkommen, dass die nachträgliche Vereinfachung von Sourcecode-Abschnitten dahingehend, dass der resultierende Code zwar die gleiche Funktionalität aufweist, aber kompakter, effizienter und evtl. sogar verständlicher ist, zu einem Rückgang der Produktivität führt.

In [FP98] wird daher zumindest empfohlen, den Softwareumfang in Function-Points zu messen, was dann zu der folgenden Formel für die Produktivität führt:

$$\text{Produktivität} = \frac{\text{implementierte Function-Points}}{\text{Personenmonate}}$$

Function-Point-Produktivität

Allerdings würden sich nach [FP98] viele Manager weigern, Function-Points zu verwenden, da ihnen dieses Maß nicht „griffig“ genug sei. Diese könnten sich zwar unter der Anzahl der Programmzeilen etwas vorstellen, hätten aber nur eine vage Vorstellung von der Bedeutung von Function-Points.

Kritik an Function-Point-Produktivität

In älteren Publikationen wird als Nachteil der Function-Point-Produktivität angeführt, dass das Function-Point-Verfahren fehlerhafte Ergebnisse liefere, wenn damit Software mit einer hohen algorithmischen Komplexität bewertet werden soll (vgl. [MWD96]). Dies liege daran, dass diese Methode für kommerzielle Software entwickelt wurde, bei der der Schwerpunkt in der Manipulation großer Datenbestände, in der Dateneingabe und der Datenausgabe liege (vgl. [BF00], S. 182). So wird auch von Balzert in [Bal98] die LOC-Produktivität als die am besten bewährte dargestellt, wobei sich diese Empfehlung auf den genannten Artikel von Maxwell et al. [MWD96] bezieht. Die genannten Schwächen der Function-Point-Methode waren auch Auslöser für die Entwicklung der verschiedenen Function-Point-Varianten, wie beispielsweise den *COSMIC Full Function Points*. In neueren Publikationen wird auch von Maxwell die Function-Point-Produktivität angewendet, wobei eine Zählweise verwendet wird, die in etwa den ungewichteten IFPUG 4.0 Function-Points entspricht (vgl. [MF00]). Allerdings wird auch die LOC-Produktivität nicht explizit als ungeeignet dargestellt.

Einflussfaktoren der Produktivität

Ein interessantes Ergebnis dieser Studie ist, dass die Produktivtät je nach betrachteter Branche von unterschiedlichen Faktoren positiv oder negativ beeinflusst wird. So konnten im Bankenbereich Skaleneffekte bei der Produktivität mit zunehmendem Softwareumfang festgestellt werden. Dies widerspricht der sonst gängigen Meinung, dass mit steigendem Programmumfang die Produktivität sinke (vgl. [Bal98], S. 13). Demgegenüber nahm im Bereich der öffentlichen Verwaltungen die Produktivität mit zunehmender Anzahl an Abfrage-Funktionen zu. Gleichzeitig nahm sie ab, je mehr die Anwender an der Entwicklung mitwirkten.

Unabhängig von den genauen Auswirkungen der verschiedenen Einflussgrößen auf die Produktivität zeigt dieses Beispiel, dass Produktivitätsmessungen sich nur mit großer Vorsicht vergleichen lassen.

Dreidimensionales Produktivitäts-konzept

Einen gänzlich anderen Aspekt der Produktivitätsbewertung betrachten dagegen Kan [Kan03], Putnam und Myers [PM03]. Sie gehen nicht von einer zweidimensionalen Produktivitätsdefinition wie bei den obigen Gleichungen aus, sondern führen aus, dass für Softwareprojekte ein dreidimensionales Produktivitätskonzept angemessen sei, welches neben Umfang und Aufwand auch die kalendermäßige Entwicklungszeit mit einschließt. Dies sei deshalb gerechtfertigt, da bei einer Verknappung der zur Verfügung stehenden Zeit sich der Aufwand überlinear vergrößere (vgl. [Kan03], S. 346).

Prozess-produktivität

Genauer wird dieser Zusammenhang in [PM03] beschrieben. Im Gegensatz zu obigen Produktivitätskonzepten, welche sowohl auf einen einzelnen Mitarbeiter als auch auf ein ganzes Projekt angewendet werden können, definieren Putnam und Myers die *Prozessproduktivität* wie folgt, wobei sie die Exponenten durch Analyse einer sehr großen Menge empirischer Daten ermittelt hätten (vgl. [PM03], S. 92):

$$\text{Prozessproduktivität} = \frac{\text{Produktumfang (mit best. Qualität)}}{\left(\frac{\text{Aufwand}}{\beta}\right)^{\frac{1}{3}} \cdot \text{Zeit}^{\frac{4}{3}}} \qquad (*)$$

Dabei ist der Faktor β ein vom Produktumfang abhängiger Parameter, welcher

dem Aufwand bei kleineren Projekten ein größeres Gewicht gibt.

In [PM02] werden der Prozessproduktivität folgende Eigenschaften zugeschrieben:

- Bei der Prozessproduktivität handelt es sich um die Produktivität einer Projektorganisation, welche gemäß eines bestimmten Vorgehensmodells arbeitet, entsprechende Werkzeuge verwendet und dabei durch geeignete Managementpraktiken geführt wird. Die Prozessproduktivität berücksichtigt daher explizit alle Aktivitäten der Softwareentwicklung inklusive Design und Test und beschränkt sich daher nicht auf reine Implementierungsaktivitäten.
- Sie unterscheidet sich von der konventionellen Produktivität hauptsächlich dadurch, dass auch die kalendermäßige Entwicklungszeit einen Faktor bei der Produktivitätsermittlung darstellt.
- Sie definiert eine nicht-lineare Beziehung zwischen den Maßen Produktumfang, Aufwand und Zeit.

Aufgelöst nach dem Produktumfang ergibt sich die folgende Darstellung:

$$\text{Produktumfang} = \left(\frac{\text{Aufwand}}{\beta}\right)^{\frac{1}{3}} \cdot \text{Zeit}^{\frac{4}{3}} \cdot \text{Prozessproduktivität}$$

Aus dieser Darstellung wird ersichtlich, dass die benötigte Zeit sublinear, der Aufwand hingegen supralinear mit dem Produktumfang zunimmt. In [PM03] wird jedoch darauf hingewiesen, dass obige Gleichungen kein Naturgesetz darstellten und die beiden Exponenten Näherungen seien, mit denen in der Praxis sehr gute Erfahrungen gemacht worden wären.

Obige Gleichung $(*)$ lässt sich auf beiden Seiten mit 3 potenzieren. Anschließend erhält man durch entsprechendes Umformen die folgende Gestalt:

$$\frac{\text{Prozessproduktivität}^3 \cdot \text{Zeit}^4}{\text{Produktumfang}^2 \cdot \beta} = \frac{\text{Produktumfang}}{\text{Aufwand}} = \text{konventionelle Produktivität}$$

Prozessproduktivität vs. konventionelle Produktivität

Damit variiert in diesem Modell die konventionelle Produktivität mit der dritten Potenz der Prozessproduktivität und mit der vierten Potenz der Entwicklungszeit. Daher würde eine Verschlechterung der Prozessproduktivität oder eine zu knapp bemessene Entwicklungszeit zu einer überproportionalen Verringerung der konventionellen Produktivität führen.

Eine weitere Umformung von Gleichung $(*)$ führt zu folgender Darstellung:

$$\left(\frac{\text{Aufwand}}{\beta}\right)^{\frac{1}{3}} \cdot \text{Zeit}^{\frac{4}{3}} = \frac{\text{Produktumfang}}{\text{Prozessproduktivität}}$$

Aufwand abhängig von der Entwicklungszeit

Wenn man die Prozessproduktivität einer Organisation als konstant betrachtet, so ist die gesamte rechte Seite obiger Gleichung für eine bestimmte Produktgröße konstant. Daher sind laut [PM03] der Aufwand und die Entwicklungszeit direkt voneinander abhängig, da der Faktor β für einen gegebenen Produktumfang eine

feste Größe hat.[6] Insbesondere würde bei einer Verringerung der kalendermäßigen Entwicklungszeit der Aufwand überproportional steigen.

2.3.4 Die Entwicklungszeit

Mit der Entwicklungszeit ist die kalendermäßig verstrichene Zeit gemeint. Diese Zeit ist zum einen für den Kunden bedeutsam, da durch die Entwicklungszeit bestimmt wird, wie lange er auf das bestellte Produkt warten muss. Meistens wird die Entwicklungszeit mittels sogenannter *Meilensteine* gegliedert. Meilensteine kennzeichnen den Beginn und das Ende eines Projekts, den Abschluss einer Projektphase und manchmal auch den Abschluss einer Gruppe von Vorgängen innerhalb einer Phase (vgl. [Bal98], S. 31). Oft werden durch Meilensteine Zeitpunkte definiert, an denen aufgrund des Erreichten eine Entscheidung über den Projektfortgang getroffen werden muss (vgl. [Hen02], S. 64).

Meilensteine

2.3.4.1 Termintreue

Für einen übersichtlichen Vergleich der tatsächlichen Entwicklungszeiten mit den ursprünglich geplanten und evtl. mit dem Kunden vereinbarten Entwicklungszeiten kann sowohl für das Gesamtprojekt als auch für Teilprojekte, Projektphasen oder Meilensteine die *Termintreue* bestimmt werden. Dafür muss zunächst für die einzelnen Teilprojekte der *Terminverzug* T_Δ berechnet werden.

Terminverzug

$$T_\Delta = T_{\mathrm{V'Ist}} - T_{\mathrm{Plan}}$$

Dabei bezeichnet $T_{\mathrm{V'Ist}}$ die voraussichtliche Ist-Dauer und T_{Plan} die geplante Dauer des Vorgangs. Damit kann für jedes Teilprojekt die Termintreue TT_{TP} berechnet werden.

Termintreue

$$TT_{\mathrm{TP}} = \frac{T_{\mathrm{Plan}} - T_\Delta}{T_{\mathrm{Plan}}} \times 100$$

Für die Termintreue des Gesamtprojekts mit n_{TP} Teilprojekten gilt dann:

$$TT_{\mathrm{ges}} = \frac{\sum TT_{\mathrm{TP}}}{n_{\mathrm{TP}}}$$

(vgl. [Bur02], S. 336)

Falls die (voraussichtliche) Ist-Dauer größer als die geplante Dauer ist, so ergibt sich eine Termintreue von unter 100 %, ansonsten ein Wert von über 100 %. Ziel der Projektführung muss daher sein, insbesondere für die Termintreue des Gesamtprojektes den Wert 100 % zu erreichen und wenn möglich zu überschreiten (vgl. [Bur02], S. 337).

Neben der Termintreue, welche statisch das Verhältnis zwischen der (voraussichtlichen) Ist-Dauer und der geplanten Dauer von Vorgängen angibt, kann auch der

[6] Dies wird von Boehm in [Boe81] in ähnlicher Weise dargestellt. So wird beim COCOMO-Verfahren aus dem Produktumfang der Entwicklungsaufwand und daraus die (optimale) Projektdauer berechnet.

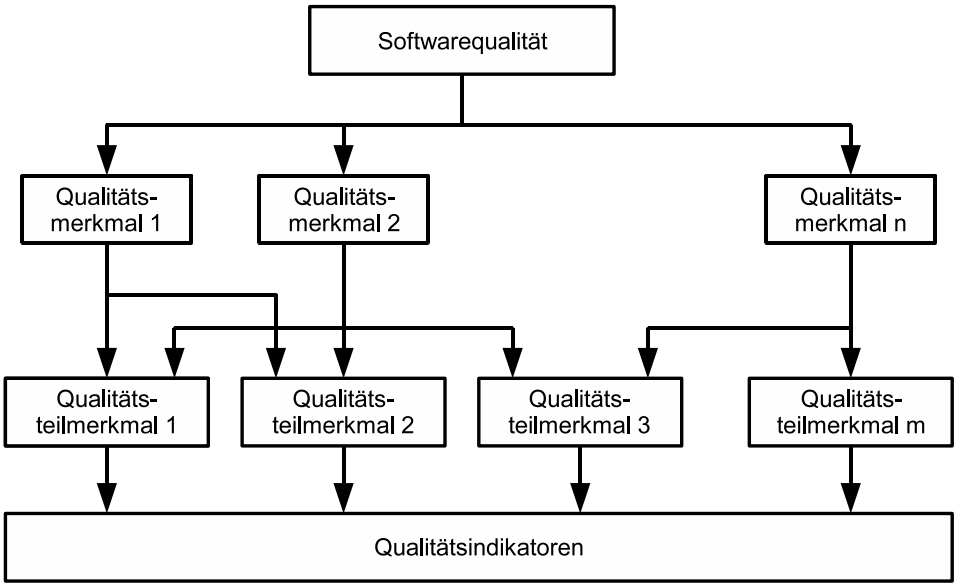


Abbildung 2.5: Aufbau des FCM-Qualitätsmodells (in Anlehnung an [Bal98])

dynamische Verlauf von Terminschätzungen untersucht werden. Geeignete Methoden hierfür sind die *Meilenstein-Trendanalyse* und die *Meilenstein-Signalliste* (siehe dazu [Bur02], S. 337 ff.).

2.3.5 Messung der Qualität

Softwarequalität wird in der ISO-Norm 9126 beschrieben als die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen (vgl. [Int01]). Für die praktische Anwendbarkeit des Qualitätsbegriffs – und um die Qualität eines Produktes überprüfen zu können – wurden *Qualitätsmodelle* entwickelt, mit denen der allgemeine Qualitätsbegriff durch Ableiten von Unterbegriffen operationalisiert werden kann (vgl. [Bal98], S. 257, bzw. auch Abschnitt 2.2.3, S. 29 ff.).

Qualitätsmodelle

Bei diesen Qualitätsmodellen wird die Qualität mittels eines hierarchischen Systems von Qualitätsmerkmalen (engl. *quality factors*), Qualitätsteilmerkmalen (engl. *quality criteria*) und Qualitätsindikatoren (engl. *quality metrics*) beschrieben. Daher wird ein so aufgebautes Modell auch als FCM-Qualitätsmodell (*Final Classification Matrix*) bezeichnet (vgl. Abbildung 2.5).

Dabei spiegeln die Qualitätsmerkmale hauptsächlich benutzerorientierte Charakteristika wider, während die Kriterien eher der softwareorientierten Sichtweise entsprechen. Die Indikatoren sind ausgewiesene Eigenschaften eines Softwareprodukts, die zu den Qualitätsmerkmalen in Beziehung gesetzt werden können. Quantifizierbare Indikatoren können mit Hilfe von Softwaremaßen bestimmt werden (vgl. [Bal98], S. 258).

ISO 9126

Beispielsweise definiert die ISO 9126 die Software-Qualitätsmerkmale

- Funktionalität,
- Zuverlässigkeit,
- Benutzbarkeit,
- Effizienz,
- Wartbarkeit und
- Portierbarkeit.

In einem Anhang der ISO 9126 werden diese Merkmale in Untermerkmale gegliedert. Beispielsweise besteht Effizienz aus den Untermerkmalen Zeitverhalten und Ressourcenverhalten. Eine schematische Darstellung des ISO-9126-Qualitätsmodells findet sich z. B. in [Bal98].

FURPS

Von der Firma *Hewlett-Packard* wurde in den 1980er Jahren ein ähnliches Qualitätsmodell entwickelt, um die Qualität ihrer Produkte zu verbessern. Die zu entwickelnde Software sollte

- die gewünschte Funktionalität besitzen,
- vom Benutzer leicht zu bedienen sein,
- zuverlässig sein,
- performant arbeiten und
- aus der Sicht des Kunden eine schnelle Unterstützung bieten.

Diese Anforderungen führten zu dem *FURPS*-Qualitätsmodell (*Functionality–Usability–Reliability–Performance–Supportability*). Auch hier gibt es entsprechende Untermerkmale oder Kriterien für die einzelnen Merkmale. So wird beispielsweise die Benutzbarkeit anhand der Kriterien Ergonomie, Ästhetik, Konsistenz und Dokumentation beurteilt (vgl. [GC87], S. 159 ff.).

Qualität als Design-Ergebnis

Nach Ansicht von Putnam und Myers sind die meisten der genannten Qualitätsmerkmale eine Frage des Software-Designs und müssten in den frühen Phasen eines Softwareprojektes bei der Konzeption entsprechend berücksichtigt werden. Insbesondere gäbe es keine allgemeingültige Möglichkeit, Qualitätsmerkmale wie die Benutzbarkeit oder die Effizienz zu messen. Allein für die Zuverlässigkeit sei eine generelle Messbarkeit in Form verschiedener Fehlermaße gegeben (vgl. [PM02], [PM03], S. 103 ff.).

Quality Function Deployment

Im Rahmen der Six-Sigma-Vorgehensweise wird das sogenannte *Quality Function Deployment* (QFD) eingesetzt, um aus Qualitätsanforderungen messbare Einflussfaktoren abzuleiten. Dazu werden Ursache-/Wirkungsanalysen durchgeführt, um die einzelnen Einflussfaktoren entsprechend bewerten und gewichten zu können (vgl. [Feh05], S. 85 ff., [MK03]). Unabhängig davon werden im Rahmen des Six-Sigma-Selbstverständnisses als „Null-Fehler-Strategie“ die verschiedensten Fehlermaße untersucht (vgl. [Feh05], S. 20 ff.).

Zuverlässigkeit eines Systems

Die Zuverlässigkeit eines Systems ist definiert als die Wahrscheinlichkeit der fehlerfreien Funktion über eine gewisse Zeitspanne (vgl. [PM03], S. 105 f.). Als Maß für die Zuverlässigkeit eines Systems wird oft die *Mean Time to Failure* (MTTF), d. h. die mittlere Dauer bis zum Fehlerfall angegeben. Dies ist der Erwartungswert für den Zeitraum, in dem das System ohne Fehler benutzt werden kann.

Abbildung 2.6: Abstufungen fehlerhafter Zustände einer Software

Fehlerrate

Ein leichter zu messendes Zuverlässigkeitsmaß ist die Fehlerrate, welche üblicherweise in Fehler pro Woche oder Fehler pro Monat angegeben wird. Falls die Fehlerrate nicht von der Zeit abhängt, d. h. über einen längeren Zeitraum konstant ist, so ist die Fehlerrate der Kehrwert der MTTF. Beispielsweise liegt bei einem System, bei dem 2 Fehler pro Monat auftreten, der Erwartungswert für den fehlerfreien Betrieb bei einem halben Monat. Allerdings wird man in der Praxis versuchen, die Fehlerrate im Laufe der Zeit immer weiter zu senken.

2.3.5.1 Der Begriff des Fehlers

Systematisierung des Begriffes

An dieser Stelle erscheint es angebracht, den Begriff des *Fehlers* genauer zu bestimmen und zu systematisieren. So sind in der englischsprachigen Literatur die Begriffe *defect*, *fault*, *error* und *failure* zu finden, welche teilweise alle mit dem Begriff *Fehler* übersetzt werden. Parhami beschreibt dagegen in [Par97] sieben Stufen der Fehlerhaftigkeit für ein KFZ-Bremssystem, wobei er Abstufungen wie *faulty*, *erroneous* oder *malfunctioning* verwendet.

Im Bereich der Softwaretechnik sollen vergleichbare deutsche Begriffe, wie sie in Abbildung 2.6 aufgelistet sind, an folgendem Beispiel erläutert werden.

Beispiel 4 *Eine in der Programmiersprache **C** programmierte Anwendung zur Verbuchung von Arbeitszeiten habe eine mangelhaft implementierte Funktion zur Speicherallokation. Für jeden zu verbuchenden Datensatz wird zunächst ein Speicherbereich einer bestimmten Größe belegt, der dann allerdings nur teilweise wieder freigegeben wird. Damit weist die Software einen **Mangel** auf, der vorhanden ist, unabhängig davon, ob die Software verwendet wird.*

Mangel

*Im täglichen Betrieb der Software wird somit mit jedem zu verbuchenden Zeiterfassungsdatensatz der verfügbare Arbeitsspeicher des Computersystems, auf dem die Anwendung läuft, verringert. Aufgrund des Mangels kommt es somit zu einer beobachtbaren **Fehlfunktion** der Anwendung. Diese kann beispielsweise mit geeigneten Monitoring-Tools beobachtet werden. Allerdings muss diese Fehlfunktion nicht notwendigerweise negative Auswirkungen haben. Unter der Annahme, dass das Computersystem über hinreichend große Speicherreserven verfügt und dass die mangelhafte Anwendung nachts im Rahmen der Datensicherung regelmäßig neu gestartet und dadurch der belegte Speicher wieder freigegeben wird, kommt es zu keinen Auffälligkeiten. Eine Fehlfunktion ist somit eine Auswirkung eines Mangels, welche für den Endanwender nicht notwendigerweise negativen Folgen hat.*

Fehlfunktion

Erst wenn aus irgendwelchen Gründen die Anzahl der zu verbuchenden Datensätze stark zunimmt, könnte der Fall eintreten, dass der Speicherplatz für die Verbuchung eines weiteren Datensatzes nicht ausreicht. An dieser Stelle würde der Anwender

Fehler

*eine **Störung** oder einen **Fehler** bei der Software erkennen, da bei der Verbuchung eine Fehlermeldung erzeugt wird.*

Ausfall

*Wenn die Software darüber hinaus derart implementiert ist, dass der mangelnde Speicherplatz nicht erkannt wird, so kann es beim Schreiben in den nicht reservierten Speicherbereich zu einem Absturz der Software und somit zu einem **Ausfall** der Anwendung kommen.*

Implementierungs- und Anforderungsmängel

Ein weiteres Kriterium der Fehler- bzw. der Mängelklassifizierung ist die Ursache des Mangels. Eine Software kann Mängel aufgrund fehlerhafter oder unvollständiger Anforderungen und aufgrund nicht erkannter Bedürfnisse des Kunden oder des Benutzers enthalten. Daneben kann die Ursache auch in der fehlerhaften Implementierung der Anforderungen, etwa in der Nicht-Einhaltung von Spezifikationen oder in „gewöhnlichen“ Programmierfehlern (*Bugs*) liegen. Daher kann man auch von Anforderungs- bzw. Implementierungsmängeln sprechen (vgl. [Feh05], S. 20 f.).

Implementierungsmängel erzeugen üblicherweise Mehraufwand und erhöhte Kosten. Teilweise können sie auch Auswirkungen auf die Kundenzufriedenheit haben. Anforderungsmängel können unter Umständen die ganze Anwendung unbrauchbar machen und somit für ein Scheitern des Projektes verantwortlich sein.

2.3.5.2 Fehlerrate

Damit muss ein Ziel der Softwareentwicklung sein, die Anzahl der in der ausgelieferten Software enthaltenen Mängel zu minimieren. Die Mängel können einerseits beim Auftreten eines Fehler erkannt werden. Sie können aber auch bereits während der Entwicklung im Rahmen von Unit-Tests oder Code-Reviews entdeckt werden. Und auch wenn somit zwischen einer Mängel- und einer Fehlerrate unterschieden werden müsste, soll im Weiteren ausschließlich der Begriff der Fehlerrate verwendet werden, da dieser sich in der deutschsprachigen Literatur eingebürgert hat.

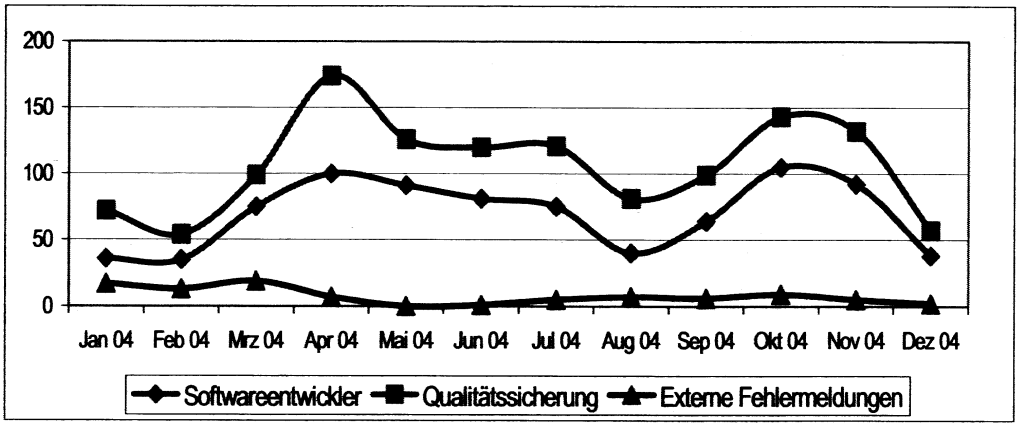
Die Fehlerrate wird, wie bereits erwähnt, in Anzahl der Fehler pro Monat angegeben. Zur genaueren Analyse kann es sinnvoll sein, die Fehler nach dem Entdecker zu klassifizieren. Dies können

- die Softwareentwickler,
- die Qualitätssicherung oder
- der Kunde oder Mitarbeiter im Vertrieb (externe Fehlermeldungen)

sein. Diese verschiedenen Fehlerarten unterscheiden sich insbesondere darin, wie lange sie in der Anwendung enthalten waren.

Fehlerfieberkurve

In Abbildung 2.7 ist die Entwicklung dieser drei Fehlerarten für die Dauer eines Jahres dargestellt. Bei diesen Fehlermeldungen wurde nicht zwischen Anforderungs- und Implementierungsmängeln unterschieden, sondern alle Fehlermeldungen wurden in einem Problem-Management-System erfasst und daraus die Auswertung generiert (vgl. [Feh05], S. 178 f.). Insbesondere kann es zum Zeitpunkt der Fehlererfassung schwierig sein, den Fehler einem Anforderungs- oder Implementierungsmangel korrekt zuzuordnen.



	01-04	02-04	03-04	04-04	05-04	06-04	07-04	08-04	09-04	10-04	11-04	12-04
Softwareentwickler	36	35	75	100	91	81	75	40	64	105	92	38
Qualitätssicherung	72	54	99	174	126	120	121	81	99	143	132	57
Externe Fehlermeldungen	*17*	*13*	*19*	*7*	*0*	*1*	*5*	*7*	*6*	*9*	*5*	*2*
Total	125	102	193	281	217	202	201	128	169	257	229	97

Abbildung 2.7: Darstellung einer Fehlerfieberkurve (in Anlehnung an [Feh05], S. 179

Mit oben genannten Informationen lassen sich nicht nur die einzelnen Fehlerraten, sondern auch der Anteil der „Fehler mit negativer Außenwirkung“ ermitteln. Dazu wird der Quotient aus den externen Fehlermeldungen und der gesamten Fehlerrate pro Monat gebildet. In obigen Beispiel nahm dieser Anteil von Oktober bis Dezember 2004 die Werte 3,50 %, 2,18 % und 2,06 % an und zeigte somit eine fallende Tendenz.

2.3.5.3 Defect Removal Effectiveness

Wie im vorhergehenden Abschnitt erwähnt, muss im Rahmen des Entwicklungsprozesses versucht werden, möglichst viele Mängel vor der Auslieferung des Produkts an den Kunden zu entdecken und zu beseitigen. Ein Maß, welches angibt, wie erfolgreich dies im Rahmen eines Projektes gelungen ist, ist die *defect-removal effectiveness* (DRE, vgl. [Kan03], S. 159 ff.), d. h. die Wirksamkeit der Maßnahmen zur Fehlerbeseitigung.

Für dieses Softwaremaß gibt es mehrere Definitionen, welche z. B. in [Kan03] näher beschrieben werden. Auf Managementebene erscheint die von Jones in [Jon96] beschriebene Berechnungsweise besonders interessant:

$$\text{DRE} = \frac{\text{Während der Entwicklung entdeckte Mängel}}{\text{Gesamtzahl der entdeckten Mängel}}$$

Dabei wird davon ausgegangen, dass nicht nur die während des Entwicklungsprozesses entdeckten Mängel gezählt werden, sondern die Zählung auch nach der Auslieferung an den Kunden weitergeführt wird. Die Gesamtzahl der entdeckten Mängel

enthält also insbesondere auch die Mängel, die aufgrund von Fehlermeldungen des Kunden entdeckt wurden.

Über diese einfache Form hinausgehend werden in [Kan03] noch einige Verfeinerungen dieses Softwaremaßes untersucht. Hierbei wird die DRE in Bezug auf die einzelnen Entwicklungsphasen (z. B. Analyse, Design, Implementierung) betrachtet. Diese Verfeinerungen werden dann auch als *Defect Containment Effectiveness* (Grad der Fehlerbeherrschung) bezeichnet, wobei nicht nur die gefundenen Mängel untersucht werden, sondern es wird versucht, auch die Mängel zu berücksichtigen, welche bei der Mängelbeseitigung evtl. neu entstanden sind. In diesem Sinne kann die DRE beispielsweise auch mit dem Six-Sigma-Ansatz kombiniert werden (vgl. [HM07b]).

Defect Containment Effectiveness

Das wie oben angegeben definierte DRE-Maß erlaubt zunächst, nach Abschluss des Projektes die DRE der im Projekt eingesetzten Prozesse einzuschätzen und mit anderen eigenen Projekten zu vergleichen. Es gibt wenig veröffentlichte Informationen von seiten der IT-Industrie über aktuell erreichte DRE-Raten. Jones berichtet, dass im Jahr 1996 in den USA die durchschnittliche DRE bei 85 % gelegen hätte. Allerdings hätten Firmen wie *AT&T*, *IBM*, *Motorola* oder *Hewlett-Packard* Erfolgsraten bis zu 99 % erreicht. Die sei insbesondere der Erfolg von formalen Design- und Code-Inspektionen im Rahmen der Softwareentwicklung gewesen (vgl. [Jon96]).

Die DRE ermöglicht jedoch auch, einzelne Maßnahmen der Qualitätssicherung (z. B. Peer-Programming, Code-Review, automatisierte Testverfahren) hinsichtlich ihrer Effektivität miteinander zu vergleichen. Dazu müssen, wie in [Kan03] beschrieben, die Anteile der durch die einzelnen Qualitätssicherungsmaßnahmen entdeckten Mängel einzeln erfasst werden.

Jones hat in [Jon96] eine interessante Studie über die Effektivität der Qualitätssicherungsmaßnahmen

Effektivität von QS-Maßnahmen

- Design-Inspektion,
- Code-Inspektion,
- Formales Testverfahren und
- Formales Freigabeverfahren

veröffentlicht. Dabei wurde die Gesamt-DRE für 16 verschiedene Kombinationen der Anwendung bzw. der Nichtanwendung einer oder mehrerer der vier genannten QS-Maßnahmen untersucht. Leider wurde dabei nicht angegeben, wieviele Projekte dabei betrachtet wurden und welche Art der Softwareentwicklung (Großrechner, Client-Server, objekt-orientiert oder nicht) dabei zur Anwendung kam.

Als Ergebniss erhielt Jones, dass bei Entfall jeglicher QS-Maßnahmen eine DRE von durchschnittlich 40 % erreicht wurde. Dabei wurden beispielsweise nur die Mängel beseitigt, welche den Entwicklern im Rahmen ihrer Programmiertätigkeit aufgefallen sind. Beim Einsatz aller vier genannten QS-Maßnahmen wurde dagegen eine durchschnittliche DRE von 99 % erreicht.

Interessant ist auch, dass bei den von Jones untersuchten Fällen bei Entfall der formalen Test- und Freigabeverfahren nur durch die beiden Inspektionsarten immer noch eine durchschnittliche DRE von 85 % erreicht wurde. Umgekehrt erreichte man ohne die Inspektionen nur DRE-Raten von durchschnittlich 65 %.

Unabhängig von Jones konkreten Ergebnissen aus dem vergangenen Jahrhundert

CMM Level	potentielle Fehlerdichte	⌀ DRE	Produktions-Fehlerdichte
1	⌀ 5,0 (3–15)	85 %	⌀ 0,750 (0,150–4,500)
2	⌀ 4,8 (3–12)	87 %	⌀ 0,624 (0,120–4,600)
3	⌀ 4,3 (2,5–9)	89 %	⌀ 0,473 (0,075–2,250)
4	⌀ 3,8 (2,3–6)	94 %	⌀ 0,228 (0,023–1,200)
5	⌀ 3,5 (2–5)	97 %	⌀ 0,105 (0,002–0,500)

Tabelle 2.5: Potentielle und Produktions-Fehlerdichten von Projekten verschiedener CMM-Levels berechnet in *Mängel pro Function-Point*, (vgl. [Jon00], S. 148 f.)

kann dieses Softwaremaß somit eingesetzt werden, um für die eigene Organisation die Effektivität der einzelnen QS-Maßnahmen zu bewerten. Diese Information kann dann verwendet werden, um entweder unterdurchschnittlich effektive Maßnahmen zu überarbeiten oder um sich bei Terminengpässen auf die effektivsten Maßnahmen zu beschränken.

2.3.5.4 Fehlerdichte beim produktiven Produkt

Die Fehlerdichte (oder eigentlich die Mängeldichte) wird in [FP98] als *de facto Standardmaß* für die Softwarequalität beschrieben. Die Mängeldichte gibt die Anzahl der bekannten Mängel in einem Produkt in Bezug auf den Produktumfang an:

$$\text{Fehlerdichte} = \frac{\text{Anzahl der bekannten Mängel}}{\text{Produktumfang}}$$

Der Produktumfang wird dabei in einem Maß wie Function Points, LOC oder NCSS angegeben.

Produktions-Fehlerdichte

Für die Zuverlässigkeit des Produkts und damit die Kundenzufriedenheit ist insbesondere die Fehlerdichte nach der Auslieferung an den Kunden relevant. Dazu ist die Anzahl der innerhalb einer bestimmten Zeitspanne nach der Produktfertigstellung entdeckten Mängel zu bestimmen. Um Vergleiche zu ermöglichen, wird hier üblicherweise ein Zeitraum von einem Jahr (vgl. [Rus02]) angesetzt. Laut Kan werden die meisten Mängel bei Anwendungssoftware innerhalb der ersten zwei Jahre und bei Betriebssystemsoftware innerhalb der ersten vier Jahre entdeckt (vgl. [Kan03], S. 88). Die so ermittelte Fehlerdichte für Software, die bereits beim Kunden im Einsatz ist, soll im Weiteren als *Produktions-Fehlerdichte* bezeichnet werden. Im Gegensatz dazu soll die aus der Gesamtzahl der insgesamt entdeckten Mängel berechnete Fehlerdichte als *potentielle Fehlerdichte* bezeichnet werden.

potentielle Fehlerdichte

Leider gibt es kaum aussagekräftige Erfahrungsberichte für durch einzelne Unternehmen tatsächlich erreichte Produktions-Fehlerdichten. Eine entsprechende Studie findet sich in [Jon00]. Diese von Jones durchgeführte Studie bezieht sich auf Daten des von ihm gegründeten Beratungsunternehmens *Software Productivity Research*.

Die Ergebnisse von Jones Studie sind in Tabelle 2.5 zusammengefasst, wobei die Fehlerdichten in *Mängel pro Function-Point* angegeben sind. Man kann erkennen,

dass die Fehlerdichte bei der ausgelieferten Software mit steigendem Reifegrad deutlich sinkt. Insbesondere sinkt sie mit zunehmendem Reifegrad mehr als die potentielle Fehlerdichte. Die Qualitätsergebnisse der Projekte überlappen sich auch über die Reifegrade der Unternehmen hinweg. So sind die besten Ergebnisse von Projekten auf CMM-Reifegrad 1 besser als die durchschnittlichen Ergebnisse auf Reifegrad 4.

Auch bei diesem Maß besteht der Hauptnutzen darin, die Qualität der eigenen Prozessergebnisse einzuschätzen und die Wirksamkeit von Prozessverbesserungsmaßnahmen beurteilen zu können.

2.4 Zusammenfassung

In diesem Kapitel wurden einige grundlegende Begriffe, Konzepte und Verfahren der Softwaremessung eingeführt. Insbesondere wurde dargestellt, warum innerhalb dieser Arbeit der Begriff des *Softwaremaßes* im Gegensatz zu dem sehr häufig in der Softwaretechnik-Literatur anzutreffenden Begriff der *Software-Metrik* verwendet wird.

Softwaremaße wurden als grundlegende Informationsquelle darstellt, mit deren Hilfe Prozesse, Produkte und Ressourcen im Rahmen des Softwareentwicklungsprozesses charakterisiert und beurteilt werden können. Weiter dienen die mittels der Softwaremetrie gewonnenen Daten sowohl der Prognose künftiger Softwareprojekte als auch der Planung und Kontrolle von Verbesserungsmaßnahmen.

Insbesondere das Ziel der Verbesserung von Softwareprozessen und den dabei hergestellten Produkten wurde in Verfahren wie dem CMMI oder der Statistischen Prozessregelung institutionalisiert. Bei diesen Verfahren nimmt die Softwaremetrie eine tragende Rolle ein.

Grundsätzlich erscheinen Softwaremessungsvorhaben, welche ohne klare und präzise Anwendungs- und Zielvorstellungen angegangen werden, mit hoher Wahrscheinlichkeit zum Scheitern verurteilt (vgl. [HF97]). Daher wurden mehrere Verfahren entwickelt, um eine zielgerichtete Auswahl geeigneter Softwaremaße zu treffen.

Mit der Goal-Question-Metrik-Methode haben Basili und seine Kollegen ein Verfahren vorgeschlagen, mit dem die Zielorientiertheit der Softwaremessung sichergestellt werden kann. Dabei werden die Management-Ziele durch schrittweise Verfeinerung soweit zerlegt, dass Softwaremaße angegeben werden können, deren Erhebung Informationen über den Grad der Zielerreichung liefern kann.

Aber auch bei der MQG-Methode werden nicht einfach Softwaremaße ohne eine Zielvorstellung erhoben, auch wenn dies zunächst so erscheinen könnte. Die MQG-Methode geht von dem impliziten Ziel aus, zunächst den aktuellen Zustand der Softwareentwicklungsprozesse zu ermitteln, um von dieser Basis ausgehend sinnvolle Ziele definieren zu können.

Beim bidirektionalen Ansatz schließlich werden die beiden Stoßrichtungen der Softwaremessung kombiniert. Einserseits werden Management-Ziele durch Verfeinerung in Eigenschaften zerlegt und andererseits werden Merkmale des Softwareentwicklungsprozesses mittels Softwaremaße messbar gemacht. Beide Stoßrichtungen

werden dann mittels der Qualitätsindikatoren zusammengeführt, wodurch sich eine Traceability zwischen den IT-Strategien und den erhobenen Softwaremaßen ergibt.

Daneben hat sich in der industriellen Praxis der Softwaremessung gezeigt, dass die Größen Umfang, Aufwand, Zeitdauer, Qualität und Produktivität zu den Standardkennzahlen gehören, die regelmäßig bei Softwaremessungsvorhaben erfasst werden. Insbesondere haben diese Maße im Rahmen der Vertragsgestaltung und des Projektmanagements eine zentrale Bedeutung.

Die Hauptintention der Softwaremetrie besteht letztendlich darin, Softwaremaße zu erheben, um damit die Erreichung bestimmter Ziele zu beurteilen. Dabei ist es wünschenswert, dass diese Softwaremaße möglichst automatisiert erhoben werden können. Zusätzlich ist es insbesondere beim MQG-Verfahren und bei der bidirektionalen Vorgehensweise hilfreich, ein entsprechendes Arsenal mit Softwaremaßen vorrätig zu haben, welche einfach konfiguriert und automatisiert erhoben werden können. Innerhalb der vorliegenden Arbeit wurde daher ein entsprechendes Framework zur Softwaremessung entwickelt, welches dem Anwender erlaubt, sich sogenannte Messwertgeber für Softwaremaße zu definieren, welche dann für die automatisierte Erhebung der gewünschten Softwaremaße konfiguriert werden können.

3 Messwerkzeuge

> Man must shape his tools lest they shape him.
>
> *(Arthur R. Miller)*

Dieses Kapitel gibt einen Überblick über verschiedene Ansätze zur werkzeugunterstützten Softwaremessung. Nach der Definition einiger grundlegender Begriffe wird ein Charakterisierungsschema dargestellt, anhand dessen verschiedene Messwerkzeuge hinsichtlich ihrer Konfigurierbarkeit beurteilt werden können.

Der Schwerpunkt dieses Kapitels bildet eine Übersicht über den Stand der Technik. Dabei werden verschiedene Messwerkzeuge beispielhaft vorgestellt, um deren Funktionsumfang, aber auch evtl. Einschränkungen aufzuzeigen.

Die Reihenfolge orientiert sich dabei am Grad der Konfigurierbarkeit der Werkzeuge und insbesondere an den jeweiligen Möglichkeiten, die zu messenden Softwaremaße bereits vorab festzulegen, so dass die Softwaremessung zur Projektlaufzeit weitestgehend automatisiert erfolgen kann.

Prozessmanagement- und Workflow-Systeme

Nicht näher betrachtet werden dabei Ansätze zur Softwaremessung, welche zwingend in Prozessmanagement- bzw. Workflow-Systeme eingebunden sind. Derartige Ansätze werden beispielsweise in [Lot96] oder [DEA98] beschrieben. Dabei geben die Workflow-Systeme den Anwendern die Aktivitäten vor, welche diese als nächstes auszuführen haben, und stoßen die Erhebung der mit der Ausführung der Tätigkeiten verbundenen Softwaremaße an. Allerdings müssen bei den beiden erwähnten Ansätzen die meisten Daten manuell erfasst werden. Durch diese Vorgehensweise ist sichergestellt, dass die Softwaremetrie-Anwendung erkennen kann, welche Projektphasen gerade durchlaufen und welche Aktivitäten dabei durchgeführt werden. Dadurch können die erhobenen Softwaremaße entsprechend zugeordnet werden. Jedoch scheint dieser Ansatz die Softwareentwickler bei ihrer Entscheidungsfreiheit sehr stark einzuengen, so dass die Entwicklerteams nur noch eingeschränkt auf Änderungen des geplanten Projektverlaufs reagieren können. Insbesondere widerspricht ein derartiger Ansatz der in dieser Arbeit angestrebten „Leichtgewichtigkeit“ (vgl. S. 4) des Softwaremessverfahrens.

3.1 Begriffe

CAME-Tools

In der Literatur finden sich, neben der Bezeichnung *Messwerkzeuge* häufig die Anglizismen *Softwaremesstools* und *CAME-Tools*. Dabei wird nach Dumke ein Softwaremesstool als Softwarewerkzeug definiert, „welches Komponenten eines Software-Produktes oder der Softwareentwicklung in ihrer Quellform oder transformierten

Form (z. B. als spezielles Modell) einliest und nach vorgegebenen Verarbeitungsvorschriften numerisch oder symbolisch auswertet“ (siehe [Dum96], S. 39). Dagegen seien *Tools für die Softwaremessung* sowohl Messtools als auch Softwarewerkzeuge, die der Ausprägung der jeweiligen Mess-Strategie, der Aufbereitung der Messobjekte oder der (statistischen) Auswertung bzw. Darstellung der Messergebnisse dienten (vgl. [Dum96], ebd.). Derartige Werkzeuge werden in der Literatur oft auch als *CAME-Tools* (*Computer Assisted Software Measurement and Evaluation*, vgl. [EDBSt05], S. 49 ff.) bezeichnet.

Messwerkzeug

Innerhalb dieser Arbeit wird mit dem Begriff *Messwerkzeug* eine Zusammenstellung softwarebasierender Werkzeugkomponenten bezeichnet, welche im Rahmen der Softwareentwicklung

- zur Softwaremessung eingesetzt werden können

und dabei evtl. noch zusätzliche Funktionalitäten

- zur Bewertung der Messergebnisse,
- zur statistischen Analyse oder
- zur Darstellung der Ergebnisse

aufweisen.

Messwertgeber

Diejenigen Teilkomponenten eines Messwerkzeuges, welche für die eigentliche Erfassung, Zählung oder Messung der zu erhebenden Messgröße verantwortlich sind, sollen als *Messwertgeber* bezeichnet werden. Dies kann im einfachsten Fall z. B. ein Betriebssystemkommando sein, welches die Anzahl der Zeilen einer Textdatei bestimmt (z. B. das Unix-Kommando `wc -l <`*filename*`>`). Üblicherweise wird man jedoch einen Messwertgeber als API-Funktion im Rahmen eines größeren Softwaremessungs-Frameworks implementieren.

Es gibt inzwischen zahlreiche Publikationen, in denen Messwerkzeuge klassifiziert und hinsichtlich ihrer Eignung für bestimmte Sprachen und Softwaremaße und ihrer Integrierbarkeit untersucht werden (vgl. [DW97], [Dum99], [RW05]). Auch gibt es Leitfäden, wie einzelne Aktivitäten der Softwaremessung durch eine Kombination von speziellen Messwerkzeugen und in der Regel bereits vorhandenen Office-Anwendungen werkzeugmäßig unterstützt werden können (vgl. [KRSZ00]).

3.2 Ein Charakterisierungsschema für Messwerkzeuge

An dieser Stelle soll ein Charakterisierungsschema dargestellt werden, welches es erlaubt, Messwerkzeuge in Bezug auf ihre Konfigurierbarkeit und individuelle Nutzbarkeit einzuordnen. Die Darstellung erfolgt dabei in Anlehnung an [MH04].

Ein Messwerkzeug stellt gemäß der Definition im vorhergehenden Abschnitt Funktionen zur

- Datensammlung,
- Verarbeitung und
- Präsentation (Visualisierung)

von Softwaremetrie-Daten zur Verfügung. Die Variabilität und damit die Konfigurierbarkeit und kontextabhängige Benutzbarkeit eines Messwerkzeuges kann nun in

Bezug auf diese drei Dimensionen charakterisiert werden.

Die erste Dimension charakterisiert die Art und Weise der Datensammlung selbst, d. h. die Erfassung, Zählung oder Messung der zu erfassenden Größe. Dies erfolgt durch die Messwertgeber, welche folgendermaßen kategorisiert werden können:

Variabilität des Messwertgebers

- *Vordefinierte Messwertgeber*: Hierbei beinhaltet das Messwerkzeug eine vordefinierte Menge von Messwertgebern, mit denen jeweils eine bestimmte Messgröße erfasst werden kann. Diese verrichten den Messvorgang in einer vorgegebenen Art und Weise und es ist nicht oder nur schwer möglich, weitere Messwertgeber in das Messwerkzeug zu integrieren. Ein Beispiel hierfür ist das im Abschnitt 3.3.1 erwähnte Kommandozeilen-Tool *CMTJava*, welches bei jedem Aufruf grundsätzlich vier vorgegebene Softwaremaße erfasst und das Ergebnis als Messprotokoll ausgibt.

- *Konfigurierbare Messwertgeber*: Hierbei enthält das Messwerkzeug ebenfalls eine vorgegebene Menge von Messwertgebern, deren Verhalten jedoch durch den Benutzer angepasst werden kann. Beispielsweise könnte ein LOC-Messwertgeber Konfigurationsmöglichkeiten bzgl. der Zählvorschriften für Leer- oder Kommentarzeilen aufweisen.

- *Variable Messwertgeber*: Bei diesen Messwerkzeugen können die vorhandenen Messwertgeber konfiguriert und zusätzliche Messwertgeber hinzugefügt werden. Üblicherweise werden dazu vom Hersteller des Messwerkzeugs die Schnittstellen, welche ein Messwertgeber implementieren muss, offen gelegt.

Variabilität der Datenverarbeitung

Die zweite Dimension charakterisiert die Methoden und Funktionen zur Verarbeitung der gemessenen Daten. Diese können folgendermaßen kategorisiert werden:

- *Vordefinierte Funktionen*: Das Messwerkzeug enthält eine vordefinierte Menge von Funktionen zur Verarbeitung, Bewertung und Verknüpfung der Messwerte. Dadurch wird eine bestimmte Art und Weise der Verarbeitung festgelegt. Beispielsweise könnte auf diese Weise eine Bewertungsfunktionalität für die Wartungsanfälligkeit von Sourcecode-Artefakten implementiert sein. Diese berechnet mittels eines bestimmten Algorithmus anhand dreier Softwaremaße (z. B. LOC, zyklomatische Komplexität, Anzahl der Parameter), ob für den untersuchten Sourcecode mit einem erhöhten Wartungsaufwand zu rechnen ist.

- *Konfigurierbare Funktionen*: Hierbei sind die Funktionen zwar ebenfalls vorgegeben, jedoch können die Verarbeitungsschritte konfiguriert und parametrisiert werden. In obigem Fall wäre beispielsweise denkbar, dass sich die Gewichtung der drei Faktoren parametrisieren lässt.

- *Variable Funktionen*: Bei Messwerkzeugen mit variablen Funktionen können die vorhandenen Funktionen nicht nur konfiguriert und parametriesiert werden, sondern es können auch neue Funktionen hinzugefügt bzw. die vorhandenen ersetzt werden. So könnte in obigem Beispiel bei der Berechnung der

Wartungsanfälligkeit evtl. noch die Anzahl der Revisionen des Sourcecode-Artefakts im Rahmen der Versionsverwaltung mit einfließen.

- *Dynamische Funktionen*: Als Messwerkzeuge mit dynamischer Funktionalität sollen solche Implementierungen bezeichnet werden, bei denen neue Funktionen durch den Endanwender dynamisch zur Laufzeit des Messwerkzeuges konfiguriert werden können. Denkbar wäre hier, dass die Verarbeitungsfunktionen mittels graphischer Symbole in einer Benutzeroberfläche dargestellt werden. Diese könnten dann zur Laufzeit der Anwendung mittels entsprechender graphischer Modellierungsfunktionalitäten zueinander in Beziehung gesetzt werden, so dass dadurch neue Verarbeitungsfunktionalitäten entstünden. Beispielsweise könnte auf diese Weise die Größenmessung und die Revisionszählung für Software-Artefakte zueinander in Beziehung gesetzt werden, so dass sich als kombiniertes Maß die Größenänderung pro Commit-Vorgang im Versionsverwaltungssystem ergibt.

Variabilität der Präsentation

Die dritte Dimension betrifft die Präsentation der Messergebnisse. In [MH04] wird dabei zwischen Präsentation und Visualisierung unterschieden. Mit *Präsentation* sei die Zusammenstellung der verarbeiteten Daten und mit *Visualisierung* die eigentliche visuelle Darstellung gemeint.[1] Da bei dieser Differenzierung die Abgrenzung zur vorhergehenden Datenverarbeitung schwierig erscheint, soll hier nur der Begriff der Präsentation verwendet werden. Die Präsentation von Ergebnissen kann dabei sowohl textuell als auch graphisch oder mittels entsprechender Mischformen erfolgen. Auch bezüglich der Präsentation lassen sich wieder drei Kategorien unterscheiden.

- *Statische Ergebnispräsentation*: Die Art und Weise der Ergebnisdarstellung ist fest vorgegeben und kann vom Anwender nicht verändert werden.
- *Konfigurierbare Ergebnispräsentation*: Hier kann die Ergebnisdarstellung vor der Anwendung des Messwerkzeuges konfiguriert werden. Beispielsweise kann der Anwender bestimmen, ob das Ergebnis in Tabellenform oder als Graphik dargestellt werden soll.
- *Dynamische Ergebnispräsentation*: Bei dieser Form kann der Anwender zur Laufzeit der Anwendung entscheiden, auf welche Weise die Ergebnisse dargestellt werden sollen.

Das hier dargestellte Klassifizierungsschema beschreibt in erster Linie den Grad der Konfigurierbarkeit und der Variabilität der Softwaremesswerkzeuge. Dieses Klassifizierungsschema wurde gewählt, um im nächsten Kapitel verschiedene Grundtypen von Messwerkzeugen voneinander abgrenzen zu können.

Funktionale Aspekte

Daneben zeichnen sich die verschiedenen Messwerkzeuge natürlich auch durch unterschiedlich ausgeprägte funktionale Aspekte aus. Dazu gehört beispielsweise die

[1] Beispielsweise könnten im Rahmen der Präsentation die darzustellenden Daten in einer XML-Datei abgelegt werden. Bei der Visualisierung wird dann die XML-Datei wahlweise im HTML- oder PDF-Format ausgegeben.

Möglichkeit, bei der Datenerfassung auf verschiedene Quellen zugreifen zu können, oder die Fähigkeit zur Echtzeitverarbeitung. Bzgl. des dargestellten Klassifizierungsschemas werden diese beiden Aspekte von den Dimensionen *Datensammlung* und *Verarbeitung* abgedeckt.

Darüber hinaus werden derartige funktionale Aspekte im folgenden Abschnitt bei der exemplarischen Beschreibung verschiedener Messwerkzeuge mit dargestellt. Die Gliederung orientiert sich dabei am Grad der Konfigurierbarkeit der Messwerkzeuge.

3.3 Messwerkzeuge: Stand der Technik

3.3.1 In Softwareentwicklungsumgebungen integrierte Messwerkzeuge

Neben kommandozeilen-orientierten Messwerkzeugen, wie *CMTjava*[2], welche einfache Komplexitäts- und Größenmaße für einzelne Sourcecode-Dateien berechnen können, gibt es schon seit einigen Jahren Messwerkzeuge, die in Softwareentwicklungsumgebungen integriert sind.

Beispielhaft hierfür ist die Entwicklungsumgebung *Borland Together 2006*[3] zu nennen. Diese Entwicklungsumgebung ist eine Weiterentwicklung der ursprünglich von der Firma *TogetherSoft* entwickelten IDE *Together ControlCenter* und basiert inzwischen auf der *Eclipse-Plattform*[4]. Eine sehr ähnliche Funktionalität wie Borland Together 2006 bietet auch das *Eclipse Metrics Plugin*[5], welches als Open-Source-Software verfügbar ist.

Together Metrics & Audits

Die Together-IDE enthält sowohl für den Sourcecode als auch für die mit Together modellierten UML-Diagramme eine große Menge vordefinierter Softwaremaße. Im Together-Sprachgebrauch wird dabei zwischen *Metrics* und *Audits* unterschieden. Eine *Metric* ist dabei ein wie in dieser Arbeit definiertes Softwaremaß. Beispielsweise gibt es eine Code-Metric, welche die Anzahl der im aktuellen Softwareprojekt vorkommenden Java-Klassen ermittelt. Ein *Audit* ist dagegen ein Softwaremaß, für das ein Schwellwert definiert wurde. Beispielsweise kann für die maximale Zyklomatische Komplexität einer Methode eine Obergrenze definiert werden. Durch das entsprechende Audit würden dann die Komplexitätswerte aller Methoden berechnet und für diejenigen Methoden, bei denen der Schwellwert überschritten wurde, eine Refaktorisierung vorgeschlagen werden.

Die Audits gehen teilweise über den Anwendungsbereich von Softwaremaßen hinaus und dienen auch dazu, Codierungs- oder Modellierungsfehler aufzudecken, indem sie die Semantik des Sourcecodes oder des UML-Modells untersuchen. Beispielsweise gibt es ein Audit, welches innerhalb von UML-Zustandsdiagrammen Zustände erkennt, die zwar erreicht, aber nicht mehr verlassen werden können und auch keinen Endzustand darstellen.

[2] CMTJava ist ein Produkt der Firma *Testwell* mit Sitz in Tampere (Finnland) (`http://www.testwell.fi/cmtjdesc.html`)

[3] `http://www.borland.com/de/products/together`

[4] `http://www.eclipse.org`

[5] `http://metrics.sourceforge.net`

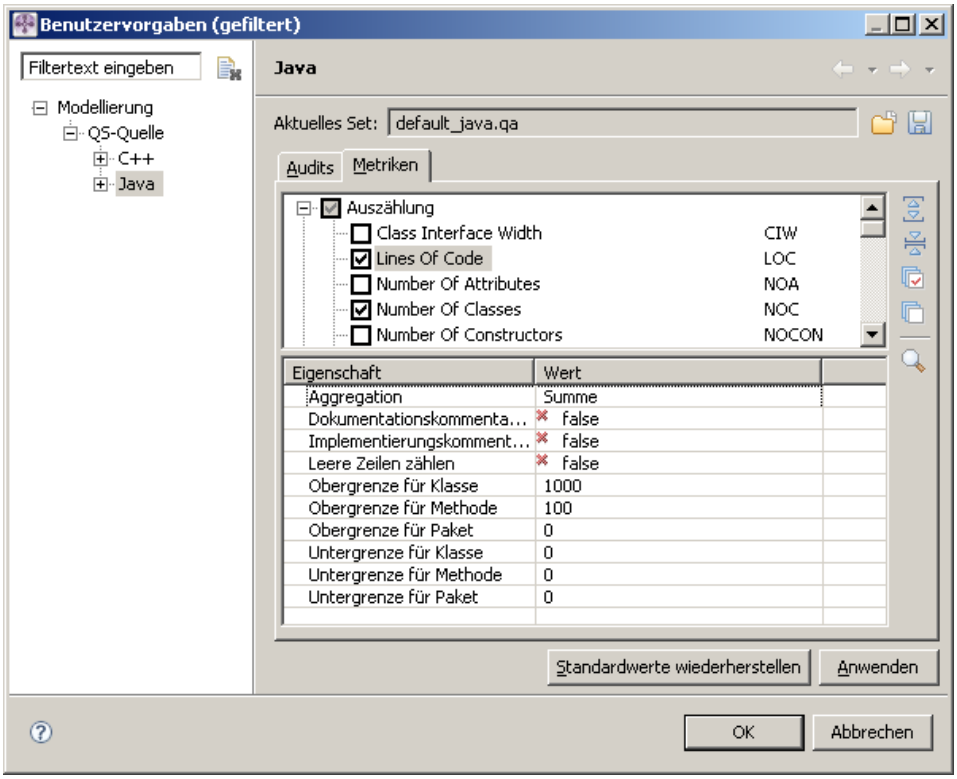


Abbildung 3.1: Konfiguration von Softwaremaßen in Borland Together

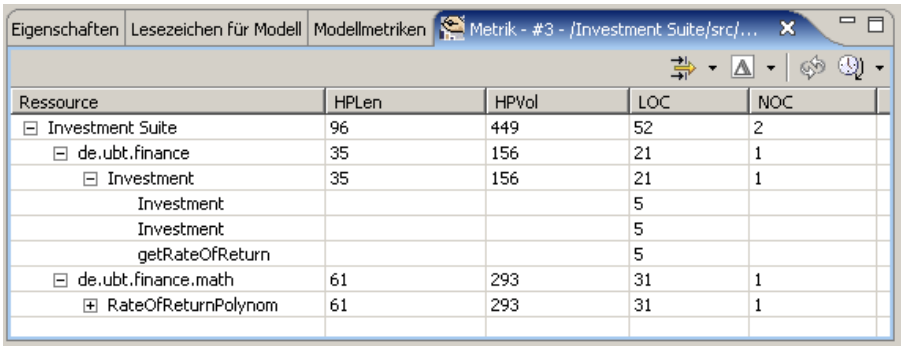
Verwendung der Together Softwaremaße

Um die in Together enthaltenen Softwaremaße verwenden zu können, müssen der Sourcecode bzw. die UML-Diagramme bereits existieren. Anschließend können die zu ermittelnden Softwaremaße ausgewählt und in engen Grenzen konfiguriert werden (vgl. Abbildung 3.1). Beispielsweise kann angegeben werden, ob das Softwaremaß für das gesamte Projekt oder nur auf ein Teilpaket angewendet werden soll und wie die Aggregation eines einzelnen Maßes (z. B. LOC) über das gesamte Paket hinweg erfolgen soll. Angeboten werden hier üblicherweise die Summation, die Mittelwertbildung oder die Ermittlung von Maximalwerten.

Eine beispielhafte Darstellung der Messergebnisse ist in Abbildung 3.2 zu sehen. Hier wurden für ein Java-Projekt die Softwaremaße

- Halstead-Länge,
- Halstead-Volumen,
- Lines of Code und
- Number of Classes

auf Methoden-, Klassen-, Paket- und Projekt-Ebene berechnet. Da die Halstead-Maße auf Methodenebene nicht definiert sind, ist bei diesen auf Klassen-Ebene die kleinste Granularität angegeben. Die Softwaremaße wurden nur für den produktiven Sourcecode und nicht für den ähnlich umfangreichen Test-Code ermittelt.



Ressource	HPLen	HPVol	LOC	NOC
Investment Suite	96	449	52	2
de.ubt.finance	35	156	21	1
Investment	35	156	21	1
Investment			5	
Investment			5	
getRateOfReturn			5	
de.ubt.finance.math	61	293	31	1
RateOfReturnPolynom	61	293	31	1

Abbildung 3.2: Ergebnisdarstellung für ausgewählte Softwaremaße in Borland Together 2006

Programmierer als Zielgruppe

Die Fähigkeiten von in Entwicklungsumgebungen integrierten Messwerkzeugen sind somit auf die Bedürfnisse des einzelnen Programmierers zugeschnitten. Dieser kann damit Code-Bestandteile erkennen, die evtl. zu kompliziert aufgebaut, zu lang oder nicht hinreichend dokumentiert sind. Auch können mit diesen Werkzeugen teilweise potentielle semantische Fehler im Sourcecode oder im Design-Modell entdeckt werden.

Eine Verknüpfung der Messergebnisse mit Softwaremaßen, welche außerhalb der IDE ermittelt werden, ist erst einmal nicht möglich, wodurch viele Softwaremaße außen vor bleiben. Es ist mit diesen Tools beispielsweise nicht möglich, den Aufwand zu bestimmen, der für die Implementierung eines Java-Paketes nötig war, und diesen zum Halstead-Volumen dieses Pakets in Beziehung zu setzen.

Auch ist es nicht möglich, inhaltliche Selektionskriterien für die Berechnung der Softwaremaße anzugeben. Beispielsweise können die Messwerkzeuge nicht erkennen, ob in bestimmten Paketen des Projektes die Anwendungslogik, die Persistenz- oder die Visualisierungs-Komponenten der Anwendung implementiert werden. Somit ist auch ein automatisierter Größenvergleich dieser Komponenten nicht möglich.

Erweiterbarkeit des Together Quality Assurance Moduls

Allerdings lassen sich die Softwaremetrie-Fähigkeiten von Together mittels seines Modul-Konzepts erweitern. Die Audit- und Metric-Funktionalitäten von Together sind im sogenannten *Quality Assurance* (QA) Modul enthalten. Mittels derartiger Module ist es grundsätzlich möglich, die Funktionalität von Together zu erweitern, da die API von Together offen und dokumentiert ist. Im Prinzip muss für ein eigenes Metric-Plugin eine Java-Klasse erstellt werden, welche die entsprechenden Interfaces aus dem QA-Modul implementiert, und in einem Unterverzeichnis der Together-Modul-Verzeichnisstruktur abgelegt werden (vgl. [CH02], S. 140 ff.).

Eine Anwendung dieser Erweiterungsmöglichkeiten wurde in [Dau05] in Form des *Metrics Builder* Plugins vorgestellt. Diese als Prototyp verfügbare Anwendung erlaubt die Kombination der in Together serienmäßig implementierten Mess-Plugins mit selbstentwickelten Plugins und damit die Definition neuer Softwaremaße. Der Messbereich (einzelne Klasse, Paket, ganzes Projekt) dieser Softwaremaße und auch die Kombination dieser Softwaremaße in Form einfacher mathematischer Terme kann

mittels einer GUI definiert werden. Beispielsweise wird in [Dau05] auf diese Weise die Berechnung der LOC-Produktivität dargestellt, indem das serienmäßige LOC-Metric-Plugin mit einem eigenentwickelten Plugin zur Aufwandsbestimmung kombiniert wird.

3.3.2 Dashboards

Die englische Bezeichnung *Dashboard* lässt sich wohl am Besten mit *Anzeigetafel* ins Deutsche übersetzen. Damit wird auch deutlich, dass Softwaremetrie-Dashboards in der Regel weniger flexibel als die im nächsten Abschnitt beschriebenen Projektleitstände sind, bei denen sich im Unterschied zu traditionellen Dashboards individuelle Softwaremaße erfassen und nutzerspezifisch im Kontext der Unternehmensziele interpretieren lassen (vgl. [MHS+06]). Im Rahmen dieser Arbeit sollen dagegen als Dashboards solche Softwaremetrie-Visualisierungssysteme bezeichnet werden, welche die Ergebnisse vordefinierter Softwaremaße mittels Zahlendarstellungen und graphischer Darstellungen visualisieren können. Diese werden zum Sammeln, zur Analyse und zur Darstellung von Daten eingesetzt, um dadurch Entscheidungsträgern eine Möglichkeit zu geben, Projektfortschritte einzuschätzen, Alternativen zu vergleichen, Risiken zu bewerten und Ergebnisse vorauszuberechnen (vgl. [Sel05]).

Eigenschaften idealtypischer Dashboards

Folgende Eigenschaften gelten typischerweise für Dashbords (vgl. [Sel05]):

- Unterstützung verschiedener Softwaremaße zur Befriedigung unterschiedlicher Informationsbedürfnisse bzw. Unterstützung unterschiedlicher Messziele
- Verwendung verschiedener Darstellungs- und Messtechniken für unterschiedliche Arten von Messdaten
- Fähigkeit zur aggregierten Darstellung auf Projekt- bzw. auf Organisationsebene
- Darstellung verantwortlicher Ansprechpartner für Feedback-Meldungen oder die Einleitung von Maßnahmen
- Darstellung von Trends und Gültigkeitszeiträumen für die Messdaten
- Darstellung von unteren und oberen Kontrollschranken im Rahmen der statistischen Prozessregelung und Zoom-Funktionalität zur Untersuchung von Messausreißern
- Umschaltmöglichkeit zu einer tabellarischen Darstellung der angezeigten Daten
- Kontext-abhängige Hilfefunktionalität
- Zoom-Funktionalität zur Datendarstellung auf unterschiedlichen Aggregationsebenen (*Drill-Down-Funktionen*)
- Ampel-Darstellung zur farblichen Kennzeichnung des Gesamtstatus

Beispiel eines Dashboards

Ein Beispiel für ein derartiges Dashboard ist in Abbildung 3.3 dargestellt. Die Abbildung 3.4 ist eine Ausschnittvergrößerung daraus. Darin werden einige der oben genannten Eigenschaften von Dashboards wie beispielsweise die Umschaltmöglichkeit zwischen tabellarischer oder graphischer Ansicht oder die Zoom-Funktionalität illustriert.

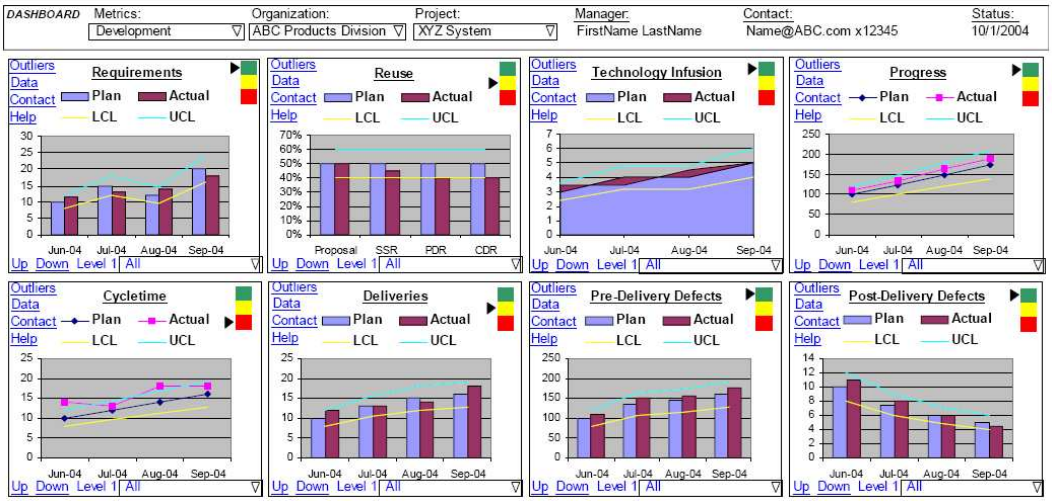


Abbildung 3.3: Beispiel einer Dashboard-Anwendung (aus [Sel05])

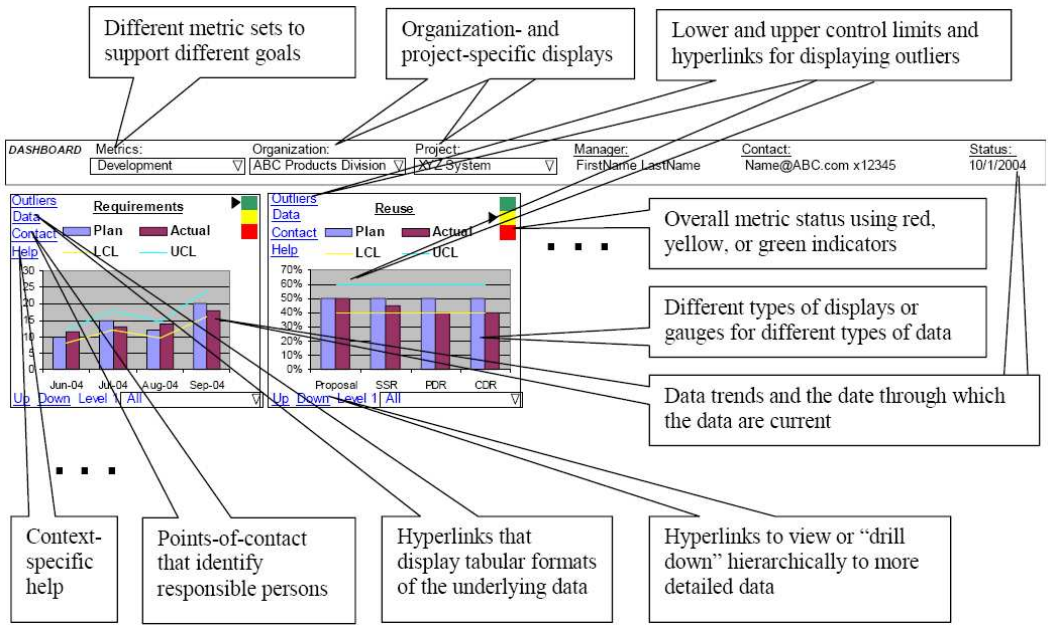


Abbildung 3.4: Teildarstellung eines Dashboards mit Illustration von Einzelfunktionalitäten (aus [Sel05])

Interaktive Dashboards

Ein Dashboard ist grundsätzlich mehr als nur eine Visualisierungsmöglichkeit für einzelne Softwaremaße, auch wenn der optische Eindruck (vgl. Abbildung 3.3) dies vielleicht nahelegen würde. Dashboards sind interaktive Werkzeuge, die z. B. teilweise die Auswahl verschiedener Aggregationsebenen ermöglichen und es damit dem Benutzer erlauben, in den Messdatenbestand „hineinzuzoomen“. Auch werden in den Anzeigebereichen nicht notwendigerweise nur einfache Messergebnisse dargestellt, sondern die generierten Darstellungen können auch auf komplizierten Verknüpfungen verschiedener Softwaremessungen basieren.

Dashboards oft ohne Messwertgeber

Jedoch wird in der Literatur bei der Beschreibung von Softwaremetrie-Dashboards nicht unbedingt gefordert, dass diese die Softwaremaße entweder selbst oder über angeschlossene Werkzeuge aktiv erheben können (vgl. [Sel05], [AGB03]).

Beispielsweise wird in [AGB03] eine Untersuchung aktueller, kommerzieller Softwaremesswerkzeuge mit Dashboard-Funktionalitäten dargestellt. Dabei waren nur zwei der fünf untersuchten Produkte in der Lage, Messwerte automatisiert zu erfassen. Diese beiden Produkte (*Metric Center*[6] und *ProjectConsole*[7]) erfassen hauptsächlich Softwaremaße, welche das gesamte Projekt betreffen (Anzahl der Anforderungen, Fehlerrate, etc.), ohne jedoch den Kontext der Messergebnisse mit zu erfassen.

Reine Visualisierung von Projektdaten

Die große Mehrheit der angebotenen Dashboard-Lösungen dient der komfortablen Visualisierung von Projektdaten, welche bereits in entsprechenden Datenbanken abgelegt sein müssen (vgl. [AGB03]). Teilweise wird unter „Automatisierter Softwaremessung“ auch nur verstanden, dass die im Softwaremesswerkzeug hinterlegten direkten und indirekten Softwaremaße automatisiert neu berechnet werden, wie dies beispielsweise bei dem Werkzeug *MetriFlame*[8] der Fall ist (vgl. [VTT99], [KPR01]). Eine Messung im eigentlichen Sinn, d. h. die Ermittlung von Messwerten mittels Messwertgeber, erfolgt dabei jedoch nicht.

3.3.2.1 Das *Amadeus*-Dashboard

Innerhalb des *Arcadia*-Projekts[9] wurden Anfang der 1990er Jahre u. a. Konzepte für derartige Softwaremetrie-Dashboards entwickelt, welche damals noch als „*measurement-driven analysis and feedback systems*“ bezeichnet wurden. Dabei wurde mit der Anwendung *Amadeus* [SPSB91] ein Prototyp eines Softwaremetrie-Dashboards entwickelt.

Empirically guided development and maintenance process

Im Vordergrund bei der Entwicklung dieses Dashboards stand insbesondere die Unterstützung eines als „*empirically guided development and maintenance process*“ bezeichneten Prozessmodells. Danach sollen Messwerkzeuge eine Art Erfahrungsdatenbank besitzen und selbständig mittels geeigneter Algorithmen erkennen können, ob beispielsweise bestimmte Entwicklungsartefakte besonders mängelanfällig sind. In diesem Fall werden automatisch entsprechende Analyse- und Testwerkzeuge auf

[6] `http://www.distributive.com`
[7] `http://www.ibm.com/developerworks/rational/products/projectconsole`
[8] `http://virtual.vtt.fi/metriflame`
[9] `http://www.ics.uci.edu/~arcadia`

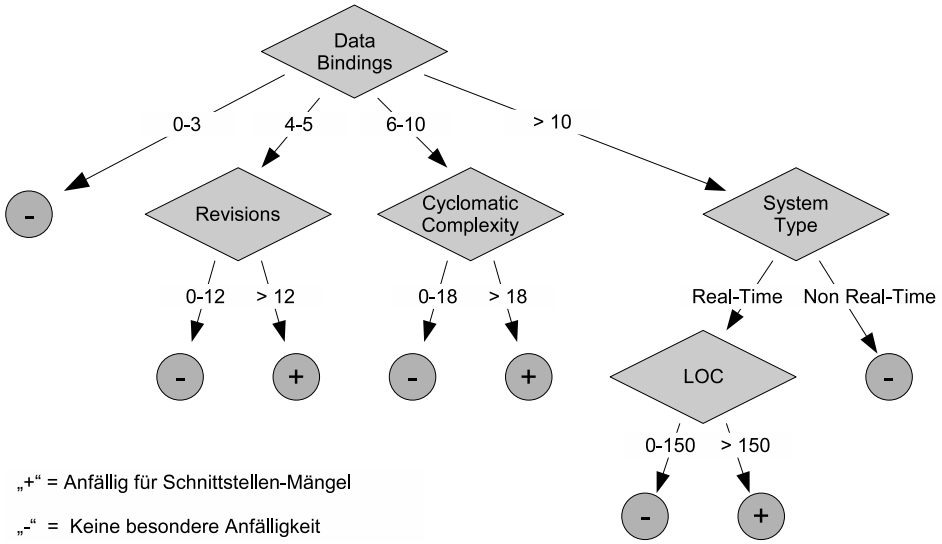


Abbildung 3.5: Auf Softwaremaßen basierender Klassifikationsbaum (in Anlehnung an [Sel05])

diese Artefakte angewendet (vgl. [SPSB91]).

Das Erkennen von mängelanfälligen Artefakten erfolgt beispielsweise mittels Klassifikationsbäumen, die auf Softwaremaßen basieren, und den zugehörigen Messwerkzeugen. Ein Beispiel für einen derartigen Klassifikationsbaum ist in Abbildung 3.5 dargestellt.

Im angegebenen Beispiel wird geprüft, ob ein Code-Objekt (z. B. ein C-Modul) eine erhöhte Anfälligkeit für Mängel bzgl. der Schnittstelle aufweist. Jede Raute stellt dabei die Anwendung eines Softwaremaßes dar und jede Verzweigung korrespondiert mit einem Wertebereich dieses Maßes. Die Blätter zeigen jeweils an, ob das Code-Objekt eine bestimmte Eigenschaft (z. B. Mängelanfälligkeit) aufweist oder nicht.

Derartige Algorithmen werden im Amadeus-System mit einer eigenen Skriptsprache implementiert. Diese stellt darüber hinaus Mechanismen zur statischen und dynamischen Interpretation von

- Prozess-Ereignissen,
- Objekt-Zustandsänderungen und
- Kalender-Ereignissen

zur Verfügung. Somit ist es beispielsweise möglich, skriptgesteuert auf Ereignisse wie

„*C-Modul* `xyz` *wurde heute mehr als dreimal modifiziert*“

zu reagieren.

Events und Agents in Amadeus

Dazu können sowohl sogenannte *Events* als auch *Agents* mittels der Skriptsprache spezifiziert werden. Ein Event ist dabei ein Monitor-Skript, mit dem Aktivitäten der Prozessdurchführung, Objektzustände und Kalender-Ereignisse beobachtet werden und in dem Bedingungen definiert werden, wann ein bestimmtes Ereignis signalisiert

werden soll. Ein Agent dagegen ist eine Prozedur, welche Daten sammeln, analysieren, zusammenfassen und visualisieren kann. Damit ist eine flexible Zuordnung zwischen Events und Agents möglich. So kann ein Event mehrere Agents anstoßen, aber auch ein Agent von verschiedenen Events angestoßen werden. Es ist auch möglich, bestimmten Events keine Agents zuzuordnen. Dann wird das Auftreten entsprechender Ereignisse zur Laufzeit nur signalisiert und der Anwender kann geeignete Agents manuell starten (vgl. [Sel05]).

Das Arcadia-Projekt

Das Amadeus-System ist keine eigenständige Anwendung, sondern wurde als Teil der Arcadia-Architektur entwickelt. Das Arcadia-Projekt endete im Jahr 1997 und die Veröffentlichungen, in denen die Amadeus-Anwendung als Prototyp beschrieben wird, stammen aus den frühen 1990er Jahren (vgl. [SPSB91], [Sel92], [Lot94]), und auch in [Sel05] wird keine neuere Veröffentlichung zitiert.

Es wird als Framework dargestellt, welches Funktionen zum Sammeln und Auswerten von Softwaremetrie-Daten zur Verfügung stellt. Um dieses Framework anwenden zu können, muss es in eine entsprechende Entwicklungsumgebung oder auch in ein Prozess-Steuerungssystem integriert werden (vgl. [Lot94]).

Implementierung der Amadeus-Konzepte

Wie weit die in [Sel05] beschriebenen Konzepte tatsächlich vollständig implementiert wurden, lässt sich über zehn Jahre nach dem Ende des Arcadia-Projekts nur schwer feststellen, da es keine neueren bzw. ausführlicheren Veröffentlichungen gibt.

Die meisten Veröffentlichungen zum Amadeus-System konzentrieren sich auf dessen Event-gesteuerte Funktionalitäten, auf Grund derer es in ein Prozess-Steuerungssystem integriert werden kann und somit die Entwickler bei der Durchführung von Projekten anleiten und lenken kann. Beispielsweise kann das Prozess-Steuerungssystem mittels der Amadeus-Funktionalitäten feststellen, welche Software-Elemente des zu entwickelnden Systems besonders mängel- oder risiko-behaftet erscheinen. Somit können Test- und Review-Aktivitäten auf derartige Projektbereiche konzentriert werden, wo der größte Nutzen für diese Aufwände zu erwarten ist.

Jedoch gibt es keine Angaben, inwiefern das Amadeus-System die im Abschnitt 2.3 dargestellten management-relevanten Softwaremaße unterstützt.

3.3.2.2 IBM Rational ProjectConsole

Als Beispiel für ein aktuelles Softwaremetrie-Dashboard soll die Software *IBM Rational ProjectConsole*[10] betrachtet werden. Diese Anwendung dient als Teil der IBM Rational Suite zur Generierung von Projektstatusberichten, die auf einer Projektwebsite in einer graphischen Statusanzeige auf der Basis der erfassten oder gemessenen Daten dynamisch erstellt und präsentiert werden.

IBM Rational Suite

Die ProjectConsole ist ein Bestandteil der *IBM Rational Suite*, welche noch zusätzlich die Anwendungen

- RequisitePro (Anforderungsverwaltung)
- ClearQuest (Aktivitäts- und Fehlerverfolgung)
- Rose (Modellierung und Programmierung)

[10] `http://www.ibm.com/developerworks/rational/products/projectconsole`

- TestManager (Testverwaltung)
- ClearCase (Konfigurationsverwaltung)

enthält.

Die Anwendung *RequisitePro* dient innerhalb der IBM Rational Suite zum Verwaltung von Anforderungen. Diese werden von RequisitePro grundsätzlich in einer relationalen Datenbank abgelegt. Zusätzlich können einzelne Anforderungen mit Textpassagen in *Microsoft Word* Dokumenten verknüpft werden. Ein solches Dokument könnte z. B. eine textuelle Beschreibung eines *Use Case* enthalten. In diesem Fall könnte dann der Anforderungsdatensatz in RequisitePro noch eine Verknüpfung mit dem graphischen UML-Modell des Use Case enthalten (vgl. [Rat02a]). *RequisitePro*

ClearQuest dient dagegen zur Verwaltung von Fehlermeldungen und Entwicklungsvorschlägen (inklusive Change-Requests). Ein Entwicklungsvorschlag oder ein sog. *Enhancement Request* kann von jedem Projektbeteiligten eingebracht werden und unterliegt keinen Formvorschriften. Er beschreibt eine neue Eigenschaft oder Funktionalität der zu erstellenden Software (vgl. [IBM04]). Im Gegensatz dazu muss eine Anforderung (*Requirement*) vollständig, eindeutig, konsistent, verifizierbar und korrekt formuliert sein. Üblicherweise werden Entwicklungsvorschläge zu Anforderungen konsolidiert, die dann auch vertraglich relevant sind. *ClearQuest Issue-Tracking*

Die Anwendung *Rose* war ursprünglich ein reines Modellierungstool für UML-Diagramme und wurde im Laufe der Zeit um Komponenten zur Code-Generierung, zur Code-Bearbeitung und zum Round-Trip-Engineering erweitert. *Rational Rose*

Zur Verwaltung von Testfällen enthält die Rational Suite den *TestManager* (vgl. [Rat02b]). Dabei lassen sich Testfälle unmittelbar aus *TestManager*

- Anforderungen aus RequisitePro,
- Rose UML-Modellen und
- Microsoft Excel Tabellen

generieren.

Alle innerhalb der IBM Rational Suite erzeugten Artefakte (Dokumente, Modelle, Sourcecode, etc.) können mit *ClearCase* versioniert werden.

Weiter soll durch die Verwendung der verschiedenen Komponenten der Rational Suite die *Traceability* von den Anforderungen, über die UML-Modelle hin zu den Code-Komponenten der zu erstellenden Anwendung und den Testfällen hergestellt werden. Dazu können die UML-Modelle mit den Anforderungen verknüpft werden, wodurch auch die aus den UML-Modellen generierten Code-Komponenten transitiv mit den Anforderungen verbunden sind. Ebenfalls können die Testfälle entweder direkt oder indirekt über UML-Diagramme mit den Anforderungen in Verbindung gesetzt werden. *Traceability*

Dadurch würden sich auch entsprechend aufschlussreiche Möglichkeiten bzgl. der Softwaremessung ergeben. So könnte man messen, welchen Umfang Code-Komponenten haben, durch die bestimmte Anforderungen implementiert werden. Auch ließe sich, bei einer entsprechend disziplinierten Pflege der Traceability-Verbindungen, anhand der Menge der erfolgreich ausführbaren Testfälle automatisiert auf den Implementierungsgrad der zugrunde liegenden Anforderungen schließen.

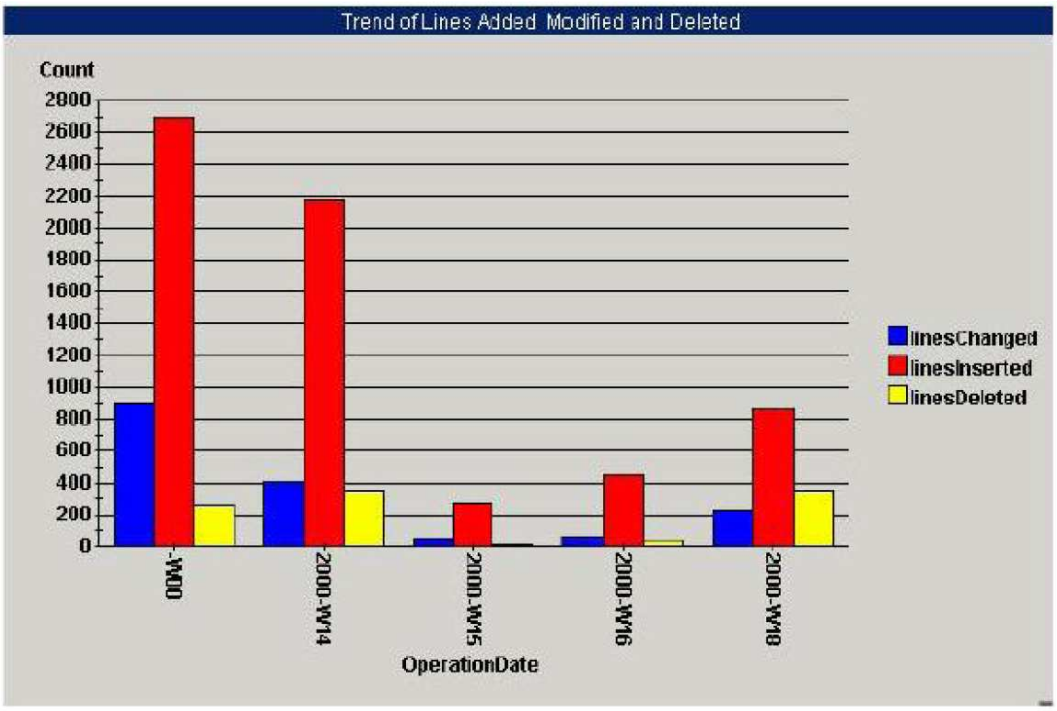


Abbildung 3.6: IBM Rational ProjectConsole – Beispielsoftwaremaß: Code-Zeilen-Vergleich zu verschiedenen Meilensteinen (aus [GWI04])

Serienmäßig nur einfache Softwaremaße implementiert

Beim Studium der entsprechenden Herstellerinformationen zur ProjectConsole (vgl. [IJ03], [GWI04], [Ish04]) fällt jedoch auf, dass sich die dort beschriebenen Softwaremaße auf das reine Abzählen von Artefakten beschränken. So wird die Anzahl der Anforderungen, die Veränderung der Anzahl von Anforderungen, die Anzahl der Anwendungsfälle (*Use Cases*) mit einem bestimmten Status und Ähnliches beschrieben. Auch können die Gesamtzahl der Programmcodezeilen (LOC) und deren Veränderung (siehe Abbildung 3.6) oder die Anzahl der auftretenden Fehler gemessen werden. Von den innerhalb der IBM Rational Suite speicherbaren Traceability-Informationen scheint man herstellerseitig bei der Verwendung der ProjectConsole keinen Gebrauch zu machen.

Softwaremessungen in Bezug auf Ressourcenmaße wie Mitarbeitereinsatz, Entwicklungszeiten und Personalaufwände sind nur in Verbindung mit *MS Project* möglich (vgl. [Ish04]). Allerdings handelt es sich hier, wie auf Seite 62 dargestellt, nicht um Softwaremessung im eigentlichen Sinn, da dabei nur Projektinformationen, welche manuell innerhalb von MS Project gepflegt werden müssen, visualisiert werden.

Kombination von ProjectConsole und MS Project

Dazu muss zunächst innerhalb von MS Project ein *Projektstrukturplan* erstellt und gepflegt werden. Für jedes Arbeitspaket müssen Informationen wie Dauer, Ressourcen, Vorgänger, Zeitaufwand und Fertigstellungsgrad in dem MS Project Projektplan manuell fortgeschrieben werden, d. h. auch hier findet keine automatisierte Messung

Task ID	Milestone	Percent Complete	Task Name	Duration	Work Hours	Start Date	Finish Date
1	True	55%	Inception Phase	68.2 days	1985.92 hrs	5/3/1999 8:00:00 AM	8/5/1999 9:36:00 AM
4	True	100%	Identify and Assess Risks	2 days	16 hrs	5/3/1999 8:00:00 AM	5/4/1999 5:00:00 PM
5	True	100%	Develop Business Case	1 day	8 hrs	5/4/1999 8:00:00 AM	5/4/1999 5:00:00 PM
6	True	100%	Initiate Project	1 day	8 hrs	5/5/1999 8:00:00 AM	5/5/1999 5:00:00 PM
7	True	100%	Project Approval Review	1 day	8 hrs	5/5/1999 8:00:00 AM	5/5/1999 5:00:00 PM

Abbildung 3.7: IBM Rational ProjectConsole – Beispielsoftwaremaß: Zeitaufwände und Fertigstellungsgrade von Projekt-Meilensteinen (aus [GWI04])

statt. Zusätzlich muss ein sogenanntes *Microsoft Project Source Template* mit Hilfe der ProjectConsole generiert werden, in dem die zu analysierenden Projekt-Felder definiert werden. Schließlich muss noch ein Mapping von den MS-Project-Feldern zu den ProjectConsole Daten-Tabellen in Bezug auf die Typisierung der Daten angegeben werden.

Anschließend können die in MS Project gepflegten Daten mittels der ProjectConsole in Form einer Dashboard-Darstellung visualisiert werden. Beispielsweise werden in Abbildung 3.7 die Zeitaufwände und die Fertigstellungsgrade von Projekt-Meilensteinen dargestellt.

Auch gibt es eine Darstellung über die Einhaltung von Kosten- und Zeit-Budgets mittels einer plakativen Tachometer-Darstellung (siehe Abbildung 3.8). Der *Cost Performance Index* (CPI) gibt an, ob durch die Projektaktivitäten aktuell mehr oder weniger als die budgetierten Kosten aufgewendet werden. Analog gibt der *Schedule Performance Index* (SPI) an, ob die Projektaktivitäten dem Zeitplan vorauseilen oder diesem hinterherhinken. Beide Indikatoren sollten ungefähr den Wert Eins annehmen.

Erweiterbarkeit der IBM Rational Suite

Da die von der ProjectConsole verwendeten Softwaremaße in Form von Perl-Skripten implementiert sind, könnten diese grundsätzlich beliebig um neue Maße bzw. Kombinationen von Softwaremaßen erweitert werden. Auch die Werkzeuge ClearCase und ClearQuest können mittels Perl-Skripte erweitert werden. Somit könnte auch auf Informationen, die der ProjectConsole standardmäßig nicht zur Verfügung stehen, zugegriffen werden, so dass diese bei der Definition entsprechender Softwaremaße verwendet werden können.

Auch wäre es grundsätzlich denkbar, die Fähigkeiten der ProjectConsole mittels geeigneter Perl-Skripte dahingehend zu erweitern, dass auch Prozessdaten wie Aufwände und Ressourcen mit einbezogen werden können. Dies wird insbesondere durch die einheitliche Perl-Programmierschnittstelle der Bestandteile der IBM Rational Suite begünstigt. Standardmäßig sind derartige Möglichkeiten derzeit jedoch nicht implementiert.

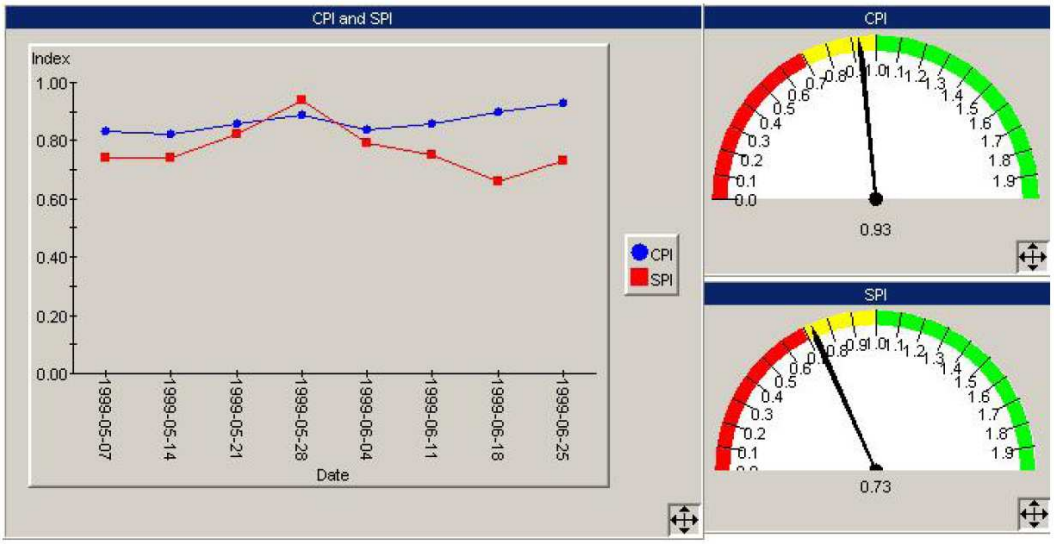


Abbildung 3.8: IBM Rational ProjectConsole – Beispielsoftwaremaß: Einhaltung von Kosten- und Zeit-Budgets (aus [Ish04])

3.3.3 Projektleitstände

Das Hauptunterscheidungsmerkmal von Projektleitständen im Vergleich zu klassischen Dashboards ist nach [MHS+06], dass sich Projektleitstände individueller konfigurieren und bedienen lassen, so dass die auf diese Weise gewonnenen Softwaremaße leichter nutzerspezifisch interpretiert werden können. Auf der anderen Seite werden die Begriffe Projektleitstand, Softwareleitstand, *Software Project Control Center* (SPCC) und Dashboard auch innerhalb derselben Publikation synonym verwendet (vgl. [MHS+06]).

Software Project Control Center

Der Unterschied von Projektleitständen zu Dashboards ergibt sich im Wesentlichen durch den Grad der Implementierung

- variabler und dynamischer Messverarbeitungsfunktionen und
- konfigurierbarer und dynamischer Ergebnispräsentationsmöglichkeiten.

Da einige Ziele der Softwaremessung (Prognose, Verbesserung, Schwachstellenanalyse) erst mit einem entsprechenden Erfahrungsdatenbank angegangen werden können, haben Projektleitstände oftmals die Möglichkeit, Messergebnisse in einem Datenbanksystem abzulegen und wiederzuverwenden.

Insgesamt ist die Grenze zwischen Dashboards und Projektleitständen eher fließend. Beispielsweise wird die IBM Rational ProjectConsole innerhalb dieser Arbeit als Dashboard eingestuft, da sie weder über eine dynamische Funktionsimplementierung noch über dynamische Ergebnispräsentationsmöglichkeiten verfügt.

3.3.3.1 Idealtypisches Architekturmodell für Projektleitstände

Münch und Heidrich haben in [MH04] ein Modell für die Architektur von Projektleitständen beschrieben, welches man wohl auf Grund der Anforderungen an Flexibilität und Konfigurierbarkeit als idealtypisch bezeichnen könnte. Um die Architekturprinzipien tatsächlich implementierter Projektleitstände beurteilen zu können, soll dieses idealtypische Architekturmodell von Münch und Heidrich in diesem Abschnitt dargestellt werden.

Das Architekturmodell basiert auf drei Schichten (siehe Abbildung 3.9):

- In der *Informations-Schicht* sind alle zum Betrieb des Projektleitstands benötigten Informationen gespeichert. Neben den Messdaten des aktuellen Projekts und Erfahrungsdaten aus früheren Projekten sind auch funktionale Komponenten zur Datenverarbeitung und zur Darstellung der Ergebnisse in entsprechenden Pools innerhalb dieser Schicht abgelegt.
- Die eigentliche Datenverarbeitung findet in der *Funktions-Schicht* statt. Hier erfolgt die Konfiguration der Verarbeitungsfunktionen, die eigentliche Verarbeitung der Messdaten und die Aufbereitung der Ergebnisse zur graphischen Darstellung.
- Die *Anwender-Schicht* schließlich ist für die Interaktion des Benutzers mit dem Projektleitstand zuständig.

Erfahrungsdatenbanken

Die eigentliche Softwaremessung nimmt dabei nur einen kleinen Anteil innerhalb der skizzierten Architektur ein. Innerhalb der Informations-Schicht gibt es dazu je eine projektspezifische und eine organisationsweite Erfahrungsdatenbank. Diese müssen die dort bereitgestellten Daten nicht notwendigerweise selbst enthalten, sondern stellen vielmehr Mechanismen zum Zugriff auf verteilte Datenbestände bereit.

Die projektspezifische Erfahrungsdatenbank bietet somit Zugriffsmöglichkeiten auf Projektpläne, Projektziele und -eigenschaften sowie auf projektspezifische Softwaremaße, während die organisationsweite Erfahrungsdatenbank Informationen wie Qualitätsmodelle oder qualitative Erfahrungswerte bereit stellt.

Bausteinkonzept

Die oben angesprochene Flexibilität und Konfigurierbarkeit dieses Architekturkonzepts rührt hauptsächlich daher, dass die Funktionalität des Systems in Form von Bausteinen in drei Pools innerhalb der Informations-Schicht hinterlegt ist, jedoch erst zur Laufzeit gemäß der aktuellen Anforderungen konfiguriert wird. Diese Bausteine implementieren die Verarbeitungsfunktionen des Projektleitstandes, die Sichten auf die Datenbestände und die Schnittstellen zur Ausgabe der Ergebnisse:

- *Funktionsbausteine*: Im sogenannten *Pool of Functions* sind Funktionen wie Monitoring, Vorhersage oder Anleitung hinterlegt.
- *Sichten*: Um die Ergebnisse der Verarbeitungsfunktionen aus verschiedenen Perspektiven darstellen zu können, stellt das System unterschiedliche Sichten beispielsweise für den Projektleiter oder den Qualitäts-Manager zur Verfügung.
- *Ausgabeschnittstellen*: Da die Sichten schließlich mit einem geeigneten Ausgabesystem (z. B. Web-Browser, Gnuplot, MS Excel) visualisiert werden müssen, ist auch ein Pool mit entsprechenden Schnittstellenfunktionen vorgesehen.

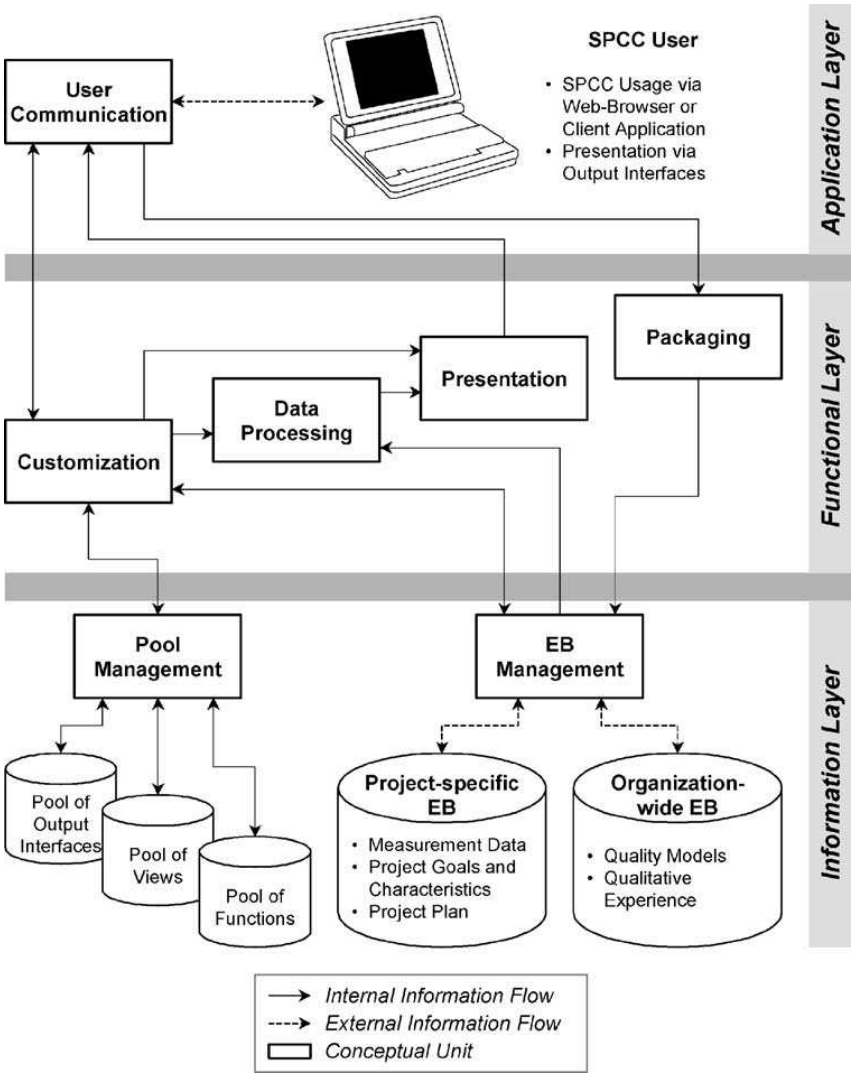


Abbildung 3.9: Idealtypisches Architekturmodell für Projektleitstände (aus [MH04])

Konfiguration

Da die in der Informationsschicht hinterlegte Funktionalität der Softwareleitstands-Architektur nach Münch und Heidrich so variabel konzipiert ist, resultieren daraus auch entsprechend komplexe Konfigurationsmöglichkeiten. Dies soll in der *Customizing*-Einheit innerhalb der Funktionsschicht erfolgen.

Zunächst müssen die Erfahrungsdatenbanken initialisiert werden, indem alle Datenquellen definiert werden. Dann muss die eigentliche Funktionalität mittels der Funktionsbausteine zusammengestellt werden. Dabei müssen auch Sichten definiert und Ausgabenschnittstellen zugeordnet werden.

Datenverarbeitung

Die *Data-Processing*-Einheit soll nun über die ausgewählten und konfigurierten Funktionsbausteine von der Customizing-Einheit informiert werden und diese hinsichtlich benötigter Ein- und Ausgabeinformationen und Abhängigkeiten zu anderen Funktionsbausteinen analysieren. Während der Ausführung der Funktionsbausteine ruft die Data-Processing-Einheit die entsprechenden Eingabe-Informationen aus den Erfahrungsdatenbanken ab und stößt dabei gegebenenfalls entsprechende Softwaremetrie-Prozesse an. Die Ergebnisse werden an die Präsentations-Einheit übermittelt werden.

Präsentation

Da dort je nach ausgewählter Sicht entsprechende Darstellungen generiert werden sollen, müssen die Sichten zunächst analysiert werden, um zu ermitteln, welche Ergebnisse der Funktionsbausteine in die Ergebnisdarstellungen integriert werden müssen. Die Ergebnisdarstellungen werden dann zur Visualisierung an die Benutzerschicht weitergeleitet.

Erfahrungssammlung

Parallel dazu können in der *Packaging*-Einheit Erfahrungen, welche bei der Benutzung des Projektleitstandes durch den Anwender gewonnen wurden, subsumiert und generalisiert werden, um sie für zukünftige Projekte in den Erfahrungsdatenbanken abzuspeichern. Dabei sollen in der Regel nicht nur einfach die Messwerte persistent abgelegt werden, sondern die Erfahrungen sollen einzeln validiert und zu *Erfahrungspaketen* gebündelt in einer *Experience Base* gespeichert werden (vgl. S. 165).

Münch und Heidrich haben mehrere Software-Messwerkzeuge hinsichtlich ihrer Konformität mit obigen Empfehlungen für Projektleitstände untersucht und für sieben Werkzeuge die gewonnenen Ergebnisse in [MH04] beschrieben. Von den untersuchten Werkzeugen weisen dabei nur vier entsprechende Möglichkeiten auf, sowohl die Funktionen als auch die Ergebnispräsentationen selbst zu konfigurieren:

- ARIS Process Performance Manager[11] (IDS Scheer AG)
- Amadeus (Arcadia Project)
- SEL-NASA Software Management Environment (SME)
- WebME

Nachdem das System *Amadeus* bereits im Abschnitt 3.3.2.1 besprochen und als klassisches Dashboard eingestuft wurde, sollen die anderen drei genannten Systeme und weitere Ansätze für Projektleitstände im Folgenden näher untersucht werden.

[11] http://www.ids-scheer.com/germany/products/aris_controlling_platform/49532

3.3.3.2 ARIS Process Performance Manager

Es überrascht ein wenig, dass der *ARIS Process Performance Manager* (ARIS PPM), welcher von der IDS Scheer AG in Saarbrücken entwickelt und vertrieben wird, in [MH04] als *Software Project Control Center* eingestuft wurde. Laut Herstellerinformationen (vgl. [PPM07]) handelt es sich dabei um ein Werkzeug zur Analyse, zur Bewertung und zum Monitoring von Unternehmensprozessen, wie beispielsweise

Werkzeug zur Untersuchung von Unternehmensprozessen

- Auftragsbearbeitung,
- Beschaffung,
- Kreditbearbeitung und Wertpapiergeschäft im Finanzbereich sowie
- zur generellen Überwachung von Service Level Agreements.

ARIS PPM wird üblicherweise an ERP-Systeme wie SAP ERP[12] angeschlossen, um die Leistung von Geschäftsprozessen anhand von Laufzeitdaten zu messen. Dabei werden Prozesskennzahlen wie

- Durchlaufzeiten,
- Termintreue oder
- Prozesskosten

berechnet. Diese Kennzahlen können dann mittels eines sogenannten *Performance-Cockpit* visualisiert werden (vgl. Abbildung 3.10).

Damit ist ARIS PPM ein Werkzeug, um die Performance von betriebswirtschaftlichen Geschäftsprozessen zu messen und zu analysieren. Grundsätzlich ist dieses System dahingehend erweiterbar, dass Daten aus anderen IT-Systemen extrahiert und verdichtet werden können. Allerdings scheint das System auf die Analyse standardisierter Prozesse, wie beispielsweise die Abwicklung einer Bestellung, spezialisiert zu sein (vgl. [PPM07]). Im Software-Engineering hat man es dagegen in der Regel mit variablen Prozessen zu tun. Insbesondere werden die durchzuführenden Projektaktivitäten für jedes Softwarevorhaben projektspezifisch angepasst (*Tailoring*), so dass das ARIS-PPM-System keine definierten Anknüpfungspunkte finden dürfte. Auch wird von Seiten des Herstellers ein derartiges Anwendungs-Szenario nicht beschrieben.

Jedoch weist die Dashboard-Darstellung viele der in Abschnitt 3.3.2 beschriebenen Eigenschaften, wie Trend-Darstellung, Drill-Down-Funktionen und Ampel-Darstellungen auf. Somit könnte ARIS PPM durchaus als Dashboard-Frontend für entsprechende Softwaremetrie-Werkzeuge in Betracht kommen. Im Auslieferungszustand ist die Software jedoch nicht als Leitstand für Softwareprozesse verwendbar.

ARIS PPM als Dashboard-Frontend

3.3.3.3 Die SEL-NASA Software Management Environment

Auch bei der *Software Management Environment* (SME, vgl. [HKVD92], [HKV94]), welche am *Software Engineering Laboratory* des *NASA Goddard Space Flight Centers* entwickelt wurde, fällt es schwer, diese als Software-Leitstand zu bezeichnen.

Vielmehr handelt es sich um ein Frontend zur Auswertung von Erfahrungsdaten über Softwareprojekte, welche in einer Datenbank abgelegt sind. Ein herausragendes

[12] `http://www.sap.com/solutions/business-suite/erp`

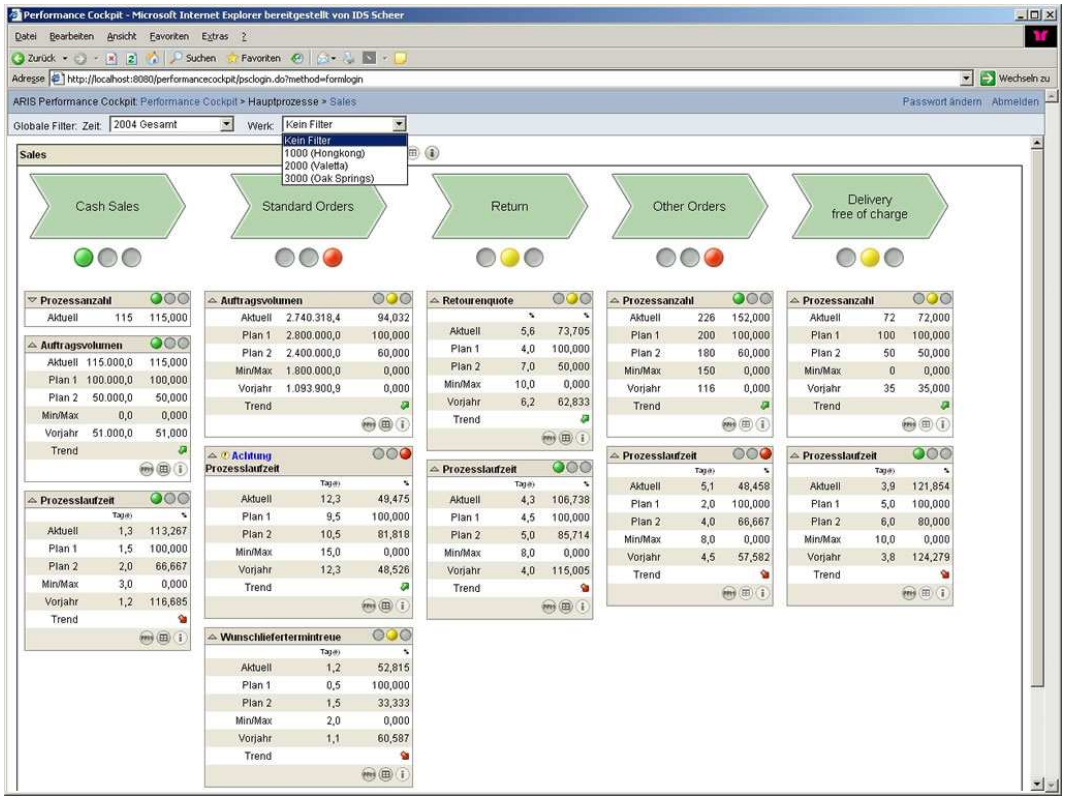


Abbildung 3.10: Performance Dashboard des ARIS Process Performance Managers

Merkmal dieses Systems ist jedoch, dass es eine Art Expertensystem integriert hat, welches dem Anwender auf Basis von sowohl aktuellen Projektwerten als auch von Erfahrungswerten konkrete Empfehlungen für die weitere Vorgehensweise geben soll.

Expertensystem

Die Daten vergangener und aktueller Projekte sind in der sogenannten *SEL-Datenbank* abgelegt, welche sich am genannten Software-Engineering-Labor im Laufe mehrerer Jahre entwickelt hat. Zu den gespeicherten Informationen gehören

- Aufwände,
- Rechenzeitnutzung,
- Softwareänderungen,
- Softwarefehler sowie
- Code-Umfangsdaten in LOC bzw. Modulanzahlen.

Letztendlich kennt das SME-System genau acht Softwaremaße dieser Art, wobei es selbst keine Softwaremessungen durchführen kann. Stattdessen müssen die Daten sowohl vergangener als auch aktueller Softwareprojekte manuell in die Datenbank eingepflegt werden.

Manuelle Datenerfassung

Auch Informationen über das Anwendungsgebiet der Softwareprojekte, verwendete Programmiersprachen, Werkzeuge und Planungsdaten (Zeit- und Aufwandsschätzungen) werden in der Datenbank hinterlegt.

Daneben wurden auf Basis in früheren Projekten gemachter Erfahrungen Modelle entwickelt, welche Beziehungen zwischen bestimmten Entitäten des Softwareentwicklungsprozesses beschreiben. Diese Modelle sind ebenfalls innerhalb des SME-Systems abrufbar. Damit soll es möglich sein, ein aktuelles Projekt im Vergleich zu einem typischen Projekt zu beurteilen und die künftige Entwicklung dieses Projektes vorherzusagen bzw. abzuschätzen.

Erfahrungsmodelle

Als Beispiele solcher Erfahrungsmodelle werden in [HKVD92] Funktionen genannt, welche den Personalbedarf innerhalb eines Projektes als eine Funktion über die Entwicklungszeit darstellen oder eine Gleichung, welche eine Beziehung zwischen dem Entwicklungsaufwand und der Anzahl der Code-Zeilen herstellt.

Schließlich enthält das SME-System noch eine Reihe von Entwicklungsregeln und Problemlösungsvorschlägen in Form eines Expertensystems (siehe folgendes Beispiel).

Beispielregel eines Expertensystems

Beispiel 5 (Expertensystemregel) *Eine Fehlerrate, die niedriger als gewöhnlich ausfällt, kann folgende Ursachen haben:*

- *Ungenügende Testaktivitäten*
- *Erfahrenes Entwicklungsteam*
- *Das Problem ist weniger kompliziert als erwartet*

Das SME-System bietet nun Möglichkeiten, die aktuellen Projektmaße wie Fehlerrate, Personalstunden oder Code-Umfang (in LOC) zu beobachten[13] und sowohl mit ausgewählten vergangen Projekten als auch mit generierten Erwartungswerten zu vergleichen, welche auf obigen Modellen basieren. Ebenfalls ist es möglich, auf Basis der aus vergangenen Projekten gewonnenen Erfahrungsmodelle, die aktuellen Messwerte zu extrapolieren, um so den weiteren Projektverlauf abzuschätzen.

Dabei geht das SME-System von einem streng wasserfallartigen Vorgehensmodell aus, welches aus den Phasen

- Design
- Codierung und Unit-Tests
- Integrationstests
- Akzeptanztests

besteht.

Aufgrund dieser Vorhersagemodelle kann das SME-System auch Abweichungen gegenüber dem geplanten Verlauf bestimmter Projektkennzahlen erkennen. Mittels des integrierten Expertensystems versucht das System in diesem Fall, dem Anwender Vorschläge für die weitere Vorgehensweise zu unterbreiten. Die Abbildung 3.11 zeigt beispielsweise eine Abweichung der kumulierten Fehlermenge vom erwarteten Verlauf. Dabei wird der erwartete Verlauf inklusive einer möglichen Bandbreite dargestellt. Zusätzlich werden mögliche Empfehlungen gegeben.

Vorschläge für die Projektleitung

Die Konfigurations- und Funktionsmöglichkeiten des SME-Systems sind somit sehr beschränkt. Das System kennt acht Softwaremaße und besitzt eine feste Anzahl Funktionen zur Beobachtung, Analyse und zum Vergleich dieser Maße. Das System

[13] Dazu müssen diese Projektmaße vorher manuell erfasst worden sein.

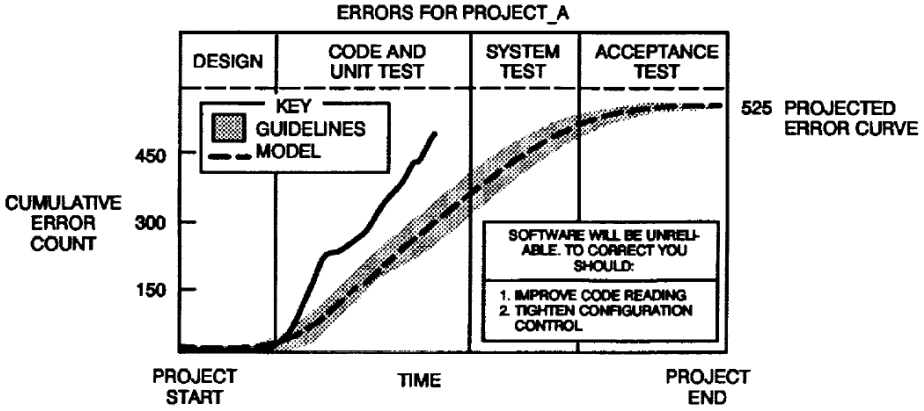


Abbildung 3.11: Empfehlungen des SME-Systems aufgrund von Abweichungen gegenüber dem geplanten Projektverlauf (aus [HKVD92])

kann die Softwaremaße nicht selbständig ermitteln, sondern ist auf eine manuelle Pflege der Daten durch den Anwender angewiesen.

Herauszuheben ist jedoch das integrierte Expertensystem, welches versucht, aufgrund hinterlegter Erfahrungswerte und Regeln, Abweichungen in den Projektkennzahlen zu erkennen und zu begründen und entsprechende Vorschläge zu unterbreiten.

3.3.3.4 WebME

Einen anderen Schwerpunkt legt das von Tesoriero und Zelkowitz entwickelte Softwaremesswerkzeug *WebME* (*Web Measurement Environment*, vgl. [TZ97]). Während bei SME der Expertensystemansatz und die Möglichkeiten der Datenvisualisierung im Vordergrund stehen, zeichnet sich das WebME-System durch dynamische Definitionen von Softwaremaßen in einer verteilten Entwicklungsumgebung aus.

Dynamische Definition von Softwaremaßen

Zwar wird in [MH04] erklärt, dass das WebME-System auf dem SME-Ansatz basieren würde, und auch die Namensgebung deutet auf einen derartigen Sachverhalt hin. Tatsächlich verwendet WebME jedoch keine Komponenten des SME-Systems und basiert auf einer völlig anderen Architektur (vgl. [TZ97]), welche derjenigen des in Abbildung 3.9 (S. 70) dargestellten idealtypischen Projektleitstandes ähnelt. Dabei kann die Informationsschicht mehrere verteilte Datenquellen enthalten, auf die mittels sogenannter *Data-Wrappers* zugegriffen wird. Diese implementieren eine einheitliche Schnittstelle zwischen den Datenquellen und der Verarbeitungslogik.

Beim WebME-System werden mittels einer Skriptsprache die verarbeitbaren Softwaremaße definiert, welche als *Attribute* bezeichnet werden. Dabei wird zwischen direkten und indirekten Attributen unterschieden. Direkte Attribute werden durch sogenannte *Instruments* erhoben. Das sind mittels Data-Wrapper gekapselte Messwertgeber. Indirekte Softwaremaße werden als *Equation* bezeichnet und ergeben sich durch Kombination von direkten Attributen mittels der vier Grundrechenarten.

Beispiel 6 *Die Skript-Anweisungen*

> ***create host*** *herkules.irgendwo.de* ***port=****8001;*
> ***create instrument*** *getSize* ***host=****herkules.irgendwo.de,*
> ***path=****/opt/webme/bin/getsize;*

sorgen dafür, dass der auf dem Server herkules *im angegebenen Verzeichnis installierte Messwertgeber über den Instrumenten-Namen* getSize *angesprochen werden kann.*

Anschließend kann mittels der Anweisung

> ***create attribute direct*** *CodeSize* ***with units*** *LOC,* ***interval*** *week,*
> ***instrument*** *getSize;*

das Softwaremaß CodeSize *definiert werden. Dieses Softwaremaß wird dabei mit der Einheit* LOC *und einem Bezugszeitraum (*interval*) von einer Woche versehen.*

Analog könnte man ein zweites Attribut Effort *definieren, welches evtl. von einem anderen Server die Personalstunden pro Woche ermittelt. Damit ließe sich dann die wöchentliche Produktivität als indirektes Softwaremaß berechnen:*

> ***create attribute indirect*** *Productivity* ***using*** *CodeSize / Effort;*

Zu beachten ist, dass beim WebME-System durch die Data-Wrappers keine einzelnen Messwerte zurückgegeben werden, sondern Vektoren mit allen verfügbaren Messwerten. Zum Beispiel würden für das oben definierte Attribut *CodeSize* die Code-Umfangsänderungen für den vollen Projektzeitraum von beispielsweise n Wochen und analog für das Attribut *Productivity* die Produktivitäten für diese n Wochen berechnet werden.

Die Einheiten und Bezugszeiträume dienen zusätzlich zur Kompatibilitätsprüfung bei den arithmetischen Berechnungen. Beispielsweise können LOC-Maße und Aufwandsmaße nicht addiert werden.

Ungeklärte Details der WebME-Funktionalität

Bei der Darstellung des Systems in [TZ97] bleibt jedoch offen, ob die Einheiten- und Zeitraum-Angaben nur innerhalb des WebME-Kerns zur Konsistenzprüfung verwendet werden, oder ob dadurch auch die Data-Wrapper bzw. die Messwertgeber selbst gesteuert werden können. Beispielsweise ist fraglich, ob das Instrument *getSize* grundsätzlich nur die wochenweise Änderung des Code-Umfangs ermitteln kann, oder ob es dies auch auf täglicher Basis könnte, wenn man dies beim Attribut *CodeSize* entsprechend angeben würde. Da die Definition von *Instrument* und *Attribute* unabhängig voneinander erfolgen, bleibt außerdem unklar, wie sichergestellt wird, dass das *Instrument* die bei der *Attribute*-Definition geforderten Intervall-Anforderungen auch beherrscht. So ist eher davon auszugehen, dass sich diese bei der *Attribute*-Definition gemachten Angaben nur innerhalb des WebME-Kerns auswirken. Dies gilt insbesondere für die Einheiten-Angaben, da nur schwer vorstellbar ist, dass die Messwertgeber diesbezüglich konfiguriert werden können.

3.3.3.5 Das Soft-Pit-Projekt

Das Ziel des *Soft-Pit*-Projektes[14], welches seit Oktober 2005 am Fraunhofer *Institut für Experimentelles Software Engineering* (IESE) zusammen mit mehreren Industriepartnern durchgeführt wird und bis zum Jahr 2008 dauern soll, „ist die Bereitstellung von *ganzheitlichen* Projektleitständen, mit denen alle für eine ingenieurmäßige Software-Projektdurchführung relevanten Aspekte analysiert, Fehlentwicklungen zeitnah erkannt und effektive Gegenmaßnahmen abgeleitet werden können“ (siehe [MHS+06]).

Ganzheitliche Projektleitstände

Unter „ganzheitlich“ wird bei diesem Ansatz verstanden, dass in einem sogenannten Software-Cockpit, kurz Soft-Pit, ausgehend von den Geschäftszielen mittels des GQM-Ansatzes (vgl. Abschnitt 2.2.1.1) systematisch Projektziele und schließlich sogenannte *Key Performance Indicators* (KPIs) für die (verteilte) Entwicklung von Software abgeleitet werden (vgl. [MHS+06]). Die KPIs entsprechen dabei den im Abschnitt 2.2.3 dargestellten *Qualitätsindikatoren* bei der bidirektionalen Strategie zur Auswahl von Softwaremaßen. Darüber hinaus sollen durch das Soft-Pit-Projekt Zusammenhänge zwischen verschiedenen, eventuell verteilten, Daten analysierbar und interpretierbar dargestellt werden und Insellösungen bezüglich der Messdatenerfassung und -verarbeitung durch einen ganzheitlichen Management-Ansatz abgelöst werden.

Key Performance Indicator

Die Forschungsschwerpunkte des Soft-Pit-Projektes sollen dabei insbesondere in folgenden Bereichen liegen (vgl. [MHS+06], [ZM07]):

- Der Soft-Pit-Leitstand soll insbesondere in kleinen und mittelgroßen Unternehmen (KMUs) eingesetzt werden. Für diese sollen zunächst Erfolgsfaktoren bzw. Geschäftsziele identifiziert werden, um zielgerichtet die korrespondierenden KPIs auswählen zu können. Dabei sollen Faktoren auf der Ebene des durchgeführten Entwicklungsprozesses (z. B. Stabilität), auf der Ebene der Projektdurchführung (z. B. Ressourcenverbrauch) und auf der Ebene der Gesamtorganisation (z. B. Reifegrad) berücksichtigt werden.

  Zielbestimmung und KPI-Auswahl

- Die ermittelten KPIs sollen im Rahmen des Projekt-Controllings eingesetzt werden und müssen entsprechend überwacht und interpretiert werden. Hierzu sind geeignete Überprüfungstechniken wie Arbeitswertverfahren, statistische Prozesskontrolle und *Tolerance Range Checking* auszuwählen und für die Verwendung in KMUs anzupassen.

  Controlling

- Beim Auftreten von Planabweichungen auf Basis der eingesetzten Projektkontrolltechniken müssen die Ursachen der Abweichungen festgestellt und geeignete Gegenmaßnahmen eingeleitet werden können. Daher müssen Techniken zur Verfolgung von Planabweichungen bereitgestellt, Ursache-Wirkungs-Relationen erstellt, sowie Möglichkeiten zur Historisierung aller ermittelten Werte und Artefakte für kontinuierliche Trendanalysen aufgezeigt werden.

  Ursache-Wirkungs-Relationen

[14] `http://www.soft-pit.de`

- Schließlich sollen geeignete Visualisierungsmechanismen entwickelt werden, um die durch den Leitstand generierten Informationen schnell erfassbar und interpretierbar zu machen und lediglich rollenbezogen relevante Informationen anzuzeigen. Eine stufenweise Erhöhung des Detailgrades der Darstellung soll dabei unterstützt werden. In die Überlegungen zu den Visualisierungsmechanismen sollen dabei auch neuere technische Möglichkeiten wie dreidimensionale VRML-Bilder[15] mit einbezogen werden.

Visualisierung

In [HM07a] wird berichtet, dass im Rahmen des Soft-Pit-Projekts ein erster Prototyp einer Leitstandsimplementierung namens *Specula* entwickelt wurde, welche in vier Unternehmen evaluiert wurde (vgl. [CHM+07]). Die Anwendung selbst ist als Web-Applikation implementiert. Die Definition der Softwaremaße und deren Verarbeitung und Interpretation erfolgt mittels sogenannter *Visualisation Catenae* (Visualisierungsketten, vgl. [HMW06]). Diese beschreiben die eigentliche Ablauflogik bei der Softwaremessung und müssen derzeit für jedes Projekt noch manuell implementiert werden (vgl. [HM07a]). In [CHM+07] wird erwähnt, dass auch ein großer Teil der Messdatenerfassung manuell erfolge. Trotz dieser Einschränkungen, kann diese Anwendung derzeit als Referenzimplementierung eines Projektleitstandes angesehen werden.

3.3.4 Werkzeuge zur Architekturüberprüfung

Neben den gerade beschriebenen und explizit für die Softwaremessung entwickelten Werkzeugen, gibt es auch Systeme, welche auch die Untersuchung und Überprüfung von Softwarearchitekturen erlauben und somit eher für Software-Architekten oder die Qualitätssicherung gedacht sind. Die Fähigkeiten dieser Systeme sollen am Beispiel der Software-Analyseumgebung *Sotograph*[16] dargestellt werden. Diese Systeme werden deshalb hier erwähnt, weil beispielsweise die Abweichung der implementierten Anwendung von einer Soll-Architektur ebenfalls bewertet und als Softwaremaß dargestellt werden kann.

Architektur-Abweichungen als Messobjekt

3.3.4.1 Der Sotograph

Dieses von der Firma *Software-Tomography* entwickelte und vertriebene Analysewerkzeug kann *Java*, *C#* und *C/C++* Software analysieren. Dazu lädt der Sotograph die Strukturinformationen über das zu untersuchende Softwaresystem in eine relationale Datenbank. Dazu gehören die im Quelltext oder im Bytecode definierten Artefakte (Methoden, Attribute, Klassen, Dateien, Pakete) und deren Beziehungen (Methodenaufruf, Attributzugriff, Vererbung, Enthaltensein, Verwendung von Ausnahmen, Typnutzung). Diese Daten dienen dann als Basis für umfangreiche Analyse- und Visualisierungsmöglichkeiten, wie

Software-Tomographie

- Architektur- und Zyklenanalysen,

[15] Die *Virtual Reality Modeling Language* ist eine Beschreibungssprache für dreidimensionale Computergraphiken.

[16] `http://www.softwaretomography.com`

- Struktur- und Qualitätsanalysen,
- Abhängigkeitsanalysen auf verschiedenen Abstraktionsebenen sowie
- *What-If*- und *Impact*-Analysen,

welche im Folgenden näher dargestellt werden. Einige der Beispiele stammen dabei aus [BBS03], wo die Analyse der freien Softwareentwicklungsumgebung *Eclipse*[17] als Fallstudie über die Analysemöglichkeiten des Sotographen beschrieben ist.

Da die oben genannten Artefakte zu fein-granular sind, um für die Architekturanalyse verwendet werden zu können, kann mittels einer zugehörigen Sprache beschrieben werden, wie der Quelltext zu architekturellen Bausteinen, sogenannten Subsystemen, zusammengefasst werden soll. Dabei werden nicht nur die Artefakte angegeben, aus denen ein Subsystem (z. B. GUI, Persistenzschicht, etc.) besteht, sondern auch die öffentlichen Schnittstellen der einzelnen Subsysteme definiert, so dass deren Einhaltung geprüft werden kann (vgl. [BKL04]).

Architektur- und Strukturanalyse

Auf Basis dieser Subsysteme kann dann wiederum die Architektur des zu untersuchenden Softwaresystem mittels einer Anzahl von Schichtenmodellen beschrieben werden. Zusätzlich können durch das Beschreiben der Schnittstellen der Subsysteme architekturelle Einschränkungen definiert werden. Basierend auf diesen Beschreibungen kann dann geprüft werden, ob es im Quelltext Referenzen gibt, die diese Schichtenmodelle durchbrechen, oder ob an den öffentlichen Schnittstellen vorbei auf Funktionen von Subsystemen zugegriffen wird. Gründe dafür sind in der Regel, dass von den Programmieren beim Zugriff auf ein Subsystem mehr Funktionalität benötigt wird, als durch die Schnittstellen zur Verfügung gestellt wird, oder dass einzelne Artefakte wiederverwendet werden, ohne dass der Entwickler dies ursprünglich vorsah. Für Software-Architekten können derartige Schnittstellenverletzungen deshalb interessant sein, da sie darauf hindeuten, dass die Zerlegung des Systems in öffentliche Schnittstellen und private Implementierungen noch verbessert werden sollte, da ansonsten die Wartbarkeit des Gesamtsystems durch die Verwendung dieser „undokumentierten Features“ leidet (vgl. [BBS03]).

Zusätzlich zu den Schnittstellenverletzungen können auch zyklische Abhängigkeiten zwischen Softwarekomponenten untersucht und entdeckt werden. Diese bewirken, dass sich Systeme nicht mehr in unabhängig handhabbare Teile zerlegen lassen, sondern als monolithische Anwendung wahrgenommen werden. Dadurch erhöhen sich Test- und Wartungsaufwände. Der Sotograph kann diese zyklischen Beziehungen auf Klassen-, Datei-, Paket- und Subsystem-Ebene erkennen (vgl. [BBS03], [BKL04]).

Zyklische Abhängigkeiten

Weiter kann die Anwendung von Design-Patterns in dem zu untersuchenden Softwaresystem mittels des Sotographen erkannt werden. Dazu verwendet der Sotograph, wie auch zur Zyklen-Analyse, das inzwischen eingebettete Werkzeug *CrocoPat* (vgl. [BNL03]).

Erkennen von Design-Patterns

Im Rahmen der Qualitätsanalyse können mit Hilfe des Sotographen unter anderem

Qualitätsanalyse

- abstrahierbare Methoden und Attribute,
- unbenutzte Artefakte,

[17] `http://www.eclipse.org`

- Code-Duplikate und
- Verstöße gegen Programmierrichtlinien

entdeckt werden (vgl. [BBS03]). Diese Code-Defizite deuten auf ein mangelhaftes oder nicht korrekt umgesetztes objekt-orientiertes Design der Anwendung bzw. auch auf mangelhafte Disziplin der Programmierer hin.

What-If- und Impact-Analyse

Der Sotograph unterstützt auch die Simulation und Evaluierung geplanter Änderungen. Refactoring-Schritte können dabei im Sotographen simuliert und somit deren Auswirkungen auf die Gesamtarchitektur geprüft werden. Die Änderungen werden dazu nur innerhalb der Datenbank durchgeführt. Damit kann geprüft werden, ob die ursprünglich erkannten Architektur-Defizite dadurch verbessert werden würden.

Erweiterbarkeit

Auch der Sotograph ist erweiterbar, da die Datenbankstruktur des Sotographen offen gelegt ist und im Sotographen mit sogenannten *Queries* abgefragt werden kann. Diese Queries werden in *TQL* (*Tomography Query Language*), einer Erweiterung von SQL, geschrieben (vgl. [BPKL06]). Dies kann ausgenutzt werden, um mit dem Sotographen noch weitergehende Architekturüberprüfungen durchzuführen als standardmäßig vorgesehen sind.

In [BPKL06] wird eine Fallstudie beschrieben, bei der mittels des Sotographen die Einhaltung der sehr komplexen WAM-Architektur [Zül04] überprüft wurde. Das Akronym *WAM* steht für Werkzeug, Automat und Material und beschreibt ein Architekturmodell zum Erstellen großer Anwendungssysteme. Dieses Architekturmodell basiert auf Architekturregeln, die sich laut [BPKL06] nicht eindeutig auf Schichten und eine festgelegte Menge an Subsystemen und deren Schnittstellen abbilden lassen. Es wird jedoch in dem genannten Beitrag gezeigt, dass sich auch in diesem komplizierteren Architekturmodell Regelverletzungen mittels geeigneter Queries finden lassen.

Software-Architekten als Zielgruppe

Somit ist der Sotograph ein Werkzeug für den Software-Architekten, der Architektur-Defizite aufdecken und die Einhaltung von Architektur- und Programmierrichtlinien überprüfen möchte. Gleichzeitig ist er auch ein Messwerkzeug, wenn man z. B. die Anzahl von Architekturdefiziten als Softwaremaß betrachtet. Auf Managementebene relevante Softwaremaße lassen sich mit dem Sotographen dagegen nicht bestimmen. Jedoch könnte er im Rahmen der Softwaremessung als Werkzeug zur Identifikation von Software-Komponenten eingesetzt werden.

3.3.4.2 Identifikation von Software-Komponenten

Wie im Kapitel 4 dargestellt wird, ist eine Anforderung an das im Rahmen dieser Arbeit entwickelte Softwaremessungs-Framework, einzelne Komponenten einer Software-Anwendung maschinell identifizieren zu können, um beispielsweise den für die Entwicklung dieser Komponenten investierten Aufwand bestimmen zu können.

Identifikation auf Basis eines Architekturmodells

Für diese Identifikationsaufgaben, bei denen jeweils festgestellt werden muss, welche Code-Artefakte zu welcher Software-Komponente gehören, ließe sich der Sotograph eventuell auch einsetzen. Dies setzt jedoch voraus, dass bei der Entwicklung ein bestimmtes Architekturmodell (z. B. *WAM* [Zül04], *Quasar* [Sie04], *CORBA*

[MR99], etc.) zum Einsatz kommt, an das sich von Seiten der Software-Entwickler auch gehalten wird.

In [BPKL06] wird diese Anwendung des Sotographen zur Identifikation von Software-Komponenten beschrieben. Allerdings war die Absicht dieser Arbeit, festzustellen, ob die gewünschte Architektur von den Entwicklern überhaupt verwendet wird. Dabei werden die folgenden Möglichkeiten, einzelne Architektur-Komponenten zu identifizieren, eingesetzt:

Namenskonventionen: Für die einzelnen Elemente einer Software-Modell-Architektur gibt es oft Namenskonventionen, so dass hierüber einzelne Komponenten und die zugehörigen Code-Artefakte identifiziert werden können. Allerdings werden Namenskonventionen durch die Entwickler nicht immer konsequent eingehalten.

Typisierung: Viele Komponenten einer Software-Architektur sind über Klassen oder Interfaces definiert, welche die implementierenden Software-Elemente erweitern bzw. implementieren müssen. Diese Typisierung kann zur Identifizierung genutzt werden, indem nach den Software-Elementen gesucht wird, welche ein bestimmtes Interface implementieren.

Annotationen: In Programmiersprachen wie Java (ab der Sprachversion Java 5) können Annotationen verwendet werden, um bestimmte Metainformationen in den Software-Artefakten abzulegen. Auch diese Informationen können zur Identifikation der Software-Elemente verwendet werden.

Wie auch in [BPKL06] beschrieben, wird in der Regel eine Kombination dieser Identifikationsmöglichkeiten zur Anwendung kommen, um die Software-Artefakte eindeutig den einzelnen Architekturkomponenten zuordnen zu könnnen.

3.4 Zusammenfassung und Beitrag dieser Arbeit

3.4.1 Beurteilung der dargestellten Ansätze

Die dargestellten Implementierungen von Umgebungen zur Softwaremessung haben gemeinsam, dass sie nur Teilaspekte der Ziele der Softwaremessung abdecken können. Wie im Abschnitt 2.1.2 ausgeführt, soll Softwaremessung die Projektbeteiligten in die Lage versetzen, die Prozesse, Produkte und Ressourcen innerhalb des Software-Engineerings zu charakterisieren, zu beurteilen, zu prognostizieren und zu verbessern. Dabei zeigen sich bei den existierenden Implementierungen von Projektleitständen jeweils unterschiedlich stark ausgeprägt Mängel in folgenden Bereichen.

Mängel existierender Projektleitstands-Implementierungen

- Um universell einsetzbar zu sein, müssen Messwerkzeuge beliebig erweiterbar sein. Diese Forderung nach Erweiterbarkeit bezieht sich dabei sowohl auf die verfügbaren Messwertgeber als auch auf die Funktionen, um die gewonnenen (direkten) Softwaremaße zu verarbeiten, zu kombinieren und zu analysieren. Sowohl neue Messwertgeber als auch Funktionsbausteine zur Verarbeitung der

Erweiterbarkeit

Daten sollten Plugin-artig dem Werkzeug hinzugefügt werden können, um auch alle Entitäten des Softwareentwicklungsprozesses vermessen zu können.

Automatisierung

- Die Softwaremessung muss so weit wie möglich automatisiert werden. Nur durch automatisierte Softwaremessung kann sichergestellt werden, dass die Messungen überhaupt durchgeführt werden und dass die Messungen in konsistenter Weise erfolgen, um eine Vergleichbarkeit der Messergebnisse zu gewährleisten. Einige der dargestellten Werkzeuge sind dagegen nur als Visualisierungswerkzeuge für bereits in entsprechenden Datenbanken vorhandene Softwaremaße zu betrachten.

Berücksichtigung des Kontexts

- Gesammelte Daten werden oftmals zusammenhanglos angeboten, was die Ermittlung und Verfolgung von Abweichungen von den Erwartungswerten und deren Ursachen unmöglich macht. Informationen, welche sich nur aus dem Zusammenspiel unterschiedlicher Datenquellen ergeben, bleiben meist völlig unerkannt (vgl. [MHS+06]).

  Viele Softwaremaße erhalten ihre Bedeutung nur in dem Kontext, in dem sie erfasst werden. Sie können nur beurteilt werden, wenn weitere Rahmenbedingungen oder das Umfeld, innerhalb derer die Softwaremessung durchgeführt wurde, bekannt sind.

  Beispielsweise sagt die Anzahl der gefundenen Fehler nichts über die Qualität des Entwicklungsprozesses aus, wenn nicht auch der Projektumfang bzw. der Umfang des zu entwickelnden Softwaresystems mit angegeben wird. Somit muss bei der Softwaremessung sichergestellt werden, dass Informationen über den Kontext des Messobjekts – im angegebenen Fall also Informationen über die Projektgröße – ebenfalls bei der Beurteilung des Messwertes zur Verfügung stehen.

Filtermöglichkeiten

- Bei der Präsentation der Messergebnisse werden die Daten oftmals ungefiltert und nicht rollenbezogen angeboten. Dadurch müssen die Projektbeteiligten eine unnötig große Menge an Daten überblicken, die eigentlich nur für jeweils eine Teilgruppe von Bedeutung sind, wodurch letztendlich die Effizienz bei der Arbeit mit diesen Daten sinkt (vgl. [MHS+06]).

Controlling-gerechte Softwaremessung

- Ein wichtiger Aspekt der Softwaremessung ist das Projekt-Controlling, bei dem insbesondere die Prozesse und die Ressourcen des Softwareprojektes beurteilt werden, mit dem Ziel, die Effizienz der Prozesse zu steigern. Dazu müssen auch die im Rahmen der Softwareentwicklung durchgeführten Aktivitäten bei der Softwaremessung identifiziert und mit erfasst werden können, um diese Aktivitäten auch auf Effektivität und Effizienz hin beurteilen zu können. Viele Messwerkzeuge sind auf die Produkte des Softwareentwicklungsprozesses fokussiert und können daher hauptsächlich im Rahmen des Prozesses erzeugte Artefakte vermessen, jedoch keine Prozesseigenschaften erfassen.

  Es bedarf jedoch auch der Möglichkeit festzustellen, welcher Personalaufwand beispielsweise mit bestimmten Aktivitäten verbunden ist oder wie effizient

bestimmte Aktivitäten (z. B. Code-Reviews) sind. Dazu müssen diese Aktivitäten im Rahmen der Softwaremessung automatisiert identifiziert werden können. Dieser Aspekt wird in den oben aufgeführten Ansätzen jedoch gar nicht berücksichtigt, und ist daher ein Kernaspekt des im Rahmen dieser Arbeit entwickelten Frameworks zur Softwaremessung. Zwar wird dieser Aspekt teilweise bereits von der Forderung nach der Kontext-Berücksichtigung abgedeckt, jedoch soll er zur einfacheren Referenzierung im Rahmen dieser Arbeit als *Controlling-gerechte Softwaremessung* (vgl. Abschnitt 4.1, S. 85 ff.) bezeichnet werden.

Kernanforderungen an Projektleitstände

Wenn man die Erfüllung der genannten Kriterien

- Erweiterbarkeit,
- Automatisierung,
- Berücksichtigung des Kontexts,
- Filtermöglichkeiten,
- Controlling-gerechte Softwaremessung

zusammen mit leistungsfähigen

- Visualisierungsmöglichkeiten

als Anforderungen an Projektleitstände betrachtet, so muss man feststellen, dass es derzeit keine existierenden Systeme gibt, welche allen diesen Forderungen gerecht werden. Dies wird auch an den in [HL06] aufgeführten Publikationen zum Thema *Softwareleitstände* deutlich (vgl. [HL06], S. 83–126). Die dort aufgeführten Softwareleitstände existieren entweder nur in Form eines Konzeptes oder sind auf einen Teilaspekt der Softwareentwicklung, wie beispielsweise die Softwarequalität, eingeschränkt.

Als Beleg dafür, dass im Bereich der Projektleitstände noch Entwicklungsbedarf besteht, mag auch das SoftPit-Projekt (vgl. Abschnitt 3.3.3.5, S. 77 ff.) gelten. Dieses Projekt, welches sich naturgemäß stark an dem im Abschnitt 3.3.3.1 (S. 69 ff.) beschriebenen idealtypischen Architekturmodell für Projektleitstände orientiert, scheint grundsätzlich die oben genannten Anforderungen angehen zu wollen. Auch wenn der genaue Funktionsumfang der innerhalb dieses Projektes implementierten Specula-Leitstandsimplementierung nur sehr ungenau beschrieben wird (vgl. [HM07a], [CHM+07]), ergibt sich dennoch an dieser Stelle die Möglichkeit, die Zielsetzungen der vorliegenden Arbeit von denen des Soft-Pit-Projektes abzugrenzen.

3.4.2 Zielsetzung: Ein leichtgewichtiger Projektleitstand

Das Soft-Pit-Projekt geht vom Umfang und der Zielsetzung her weit über die eigentliche Softwaremessung hinaus. Die tatsächliche Implementierung einer Projektleitstandsanwendung erscheint nur als Teilprojekt innerhalb des Soft-Pit-Themenkomplexes. Vielmehr stehen dagegen die Auswahl von Key Performance Indikatoren, die Ausarbeitung von Ursache-Wirkungs-Relationen und die Entwicklung von Projektkontrolltechniken im Vordergrund.

In dieser Arbeit soll dagegen ein eher leichtgewichtiges Framework zur Softwaremessung dargestellt werden, welches zur Umsetzung einer Controlling-gerechten Softwaremessung eingesetzt werden kann. Unter „leichtgewichtig“ ist dabei zu verstehen, dass nicht ausgefeilte Visualisierungsmechanismen und eine komfortable Benutzeroberfläche im Vordergrund stehen sollen. Stattdessen soll das Framework in vorhandene Entwicklungsumgebungen integrierbar sein, und die Anwender sollen durch die Verwendung des Softwaremessungs-Framework nicht gezwungen werden, ihre Prozesse zu ändern. Wie im Kapitel 5 dargestellt wird, ist das Framework skriptgesteuert konzipiert und die Erweiterbarkeit und Konfigurierbarkeit ergibt sich aus der jederzeitigen Anpassbarkeit der entsprechenden Programm-Skripte.

Das Ergebnis dieser Arbeit ist damit weder ein Dashboard, im Sinne eines Visualisierungssystems für Softwaremaße, noch ein elaborierter Projektleitstand, wie er im Soft-Pit-Projekt angestrebt wird. Eine gute Charakterisierung wird dagegen durch den Begriff *Kommandozeilen-Leitstand* gegeben, da die Softwaremessungen selbst mittels in einer Skriptsprache implementierter Messwertgeber erfolgen. Diese Messwertgeber können zur Projektlaufzeit letztendlich auf der Kommandozeile den aktuellen Erfordernissen angepasst werden.

Kommandozeilen-Leitstand

Um dabei eine Controlling-gerechte Softwaremessung zu erreichen, bestehen die wissenschaftliche Herausforderungen darin, Möglichkeiten aufzuzeigen,

- die zu erhebenden Softwaremaße unabhängig von einem konkreten Projekt zu definieren und
- die zu messenden Entitäten durch die verwendeten Messwerkzeuge automatisiert identifizieren zu können.

Nur dadurch ist es möglich, wie in Kapitel 1 gefordert (vgl. S. 3), die Softwaremaße wiederzuverwenden und projektübergreifend einzusetzen. Der entsprechende Ansatz wird im folgenden Kapitel dargestellt. Die Kernidee dabei ist, die Softwaremaße an Elemente des Vorgehensmodells anzuknüpfen (vgl. Abschnitt 4.2). Es wird im angegebenen Kapitel gezeigt werden, dass dadurch sowohl

- die automatische Identifizierung der zu messenden Entitäten als auch
- die Zuordnung der Softwaremaße zu einem Kontext sowie
- die konsistente und vergleichende Softwaremessung über Projektgrenzen hinweg

ermöglicht wird.

4 Prozessintegration

Change is the essential process of all existence.

(Mr. Spock, Star Trek)

4.1 Controlling-gerechte Softwaremessung

Wie im Abschnitt 2.3 dargestellt, sind auf Managementebene und damit auch für das Projekt-Controlling insbesondere die als Kernattribute bezeichneten Messgrößen

- Umfang,
- Aufwand,
- Qualität,
- kalendermäßige Zeitdauer und
- Produktivität

von großer Bedeutung. Diese Kernattribute können im Kontext verschiedener Entitäten untersucht werden. Beispielsweise kann der Aufwand in Bezug auf einzelne Arbeitspakete, Meilensteine, Entwicklungsphasen oder Tätigkeitsarten untersucht werden. Diese kontextabhängige Sichtweise ist für das Projektmanagement und dabei insbesondere für die Aufwandsschätzung und das Projekt-Controlling von Belang. Dies soll durch die folgenden Fragestellungen verdeutlicht werden.

Aufwand für einzelne Softwarekomponenten

Komponentenbezogene Aufwandsverteilung: Welcher Aufwand wurde für die Entwicklung einer bestimmten Softwarekomponente, z. B. des Web-Frontends einer Anwendung, erbracht? Der ermittelte Aufwand kann beispielsweise bei einem nachfolgenden Softwareprojekt in die Bepreisung oder in Entscheidungen bzgl. des Personalbedarfes eingehen.
Eine genaue Berechnung des Herstellungsaufwandes für einzelne Komponenten ist grundsätzlich nur möglich, wenn bei der Zeiterfassung auf Komponentenebene kontiert wird. Da dies in der Praxis nicht immer der Fall ist, sondern meistens nur auf Tätigkeiten kontiert wird, sind heuristische Verfahren (Faustregeln) denkbar, den Herstellungsaufwand für einzelne Komponenten zumindest näherungsweise zu bestimmen. Dies ist beispielsweise möglich, wenn bekannt ist, dass bestimmte Softwareentwickler in der Regel jeweils nur einzelne Komponenten (z. B. Web-Oberflächen) implementieren.

Aufwand je Phase

Phasenbezogene Aufwandsverteilung: Wie verhalten sich die normierten Aufwände für einzelne Entwicklungsphasen zwischen verschiedenen Projekten? Daraus können evtl. Rückschlüsse auf unterschätzte Schwierigkeiten bei der Projektplanung gezogen werden oder es zeichnen sich Trends bzgl. der Aufwandsverteilung in den abgeschlossenen Projekten ab. Da die Projekte in der Regel einen

unterschiedlich großen Umfang haben, müssen die Aufwandsangaben normiert werden, indem sie durch einen Faktor dividiert werden, der die Projektgröße widerspiegelt.

Verteilung des Arbeitsfortschrittes: Softwareentwicklung erfolgt heutzutage häufig iterativ. Dabei wird zuerst ein Prototyp entwickelt, der dann in weiteren Iterationsstufen ausgebaut und verfeinert wird. Schließlich werden im Rahmen von Kundenakzeptanztests weitere Modifikationen vorgenommen bzw. noch zusätzliche Funktionen implementiert. Bei dieser Konstellation kann von Interesse sein, welcher Anteil der fertigen Anwendung bereits in der Prototyp-Phase entstanden ist, und welcher Programmumfang später hinzugefügt wurde. Auch ist interessant, welchen Umfang die Programmänderungen während der Testphasen haben, da daraus Rückschlüsse auf die Genauigkeit bzw. die Stabilität der ursprünglichen Anforderungen gezogen werden können. Falls dieser in den einzelnen Phasen erreichte Arbeitsfortschritt bzw. Fertigstellungsgrad monetär bewertet wird, so erhält man den in diesen Phasen erzeugten *Earned Value* (vgl. [Fie05], S. 157 f., [Bur02], S. 371 ff.).

Aufwand für Prototyp vs. Gesamtaufwand

Earned Value Analyse

Fehlerverteilung: Wie im Abschnitt 2.3.5.3 dargestellt, kann unter bestimmten Bedingungen durch Einführung formaler Design- und Code-Reviews die *Defect-Removal-Effectiveness* (DRE) deutlich erhöht werden. Die DRE kann zwar erst nach Projektende bestimmt werden, jedoch sollte sich die Effektivität der Reviews auch bereits zur Projektlaufzeit durch eine Verschiebung der Fehlerrate hin zu den früheren Projektphasen zeigen.

Effektivitätskontrolle

Der entscheidende Punkt an obigen Beispielen, welche noch weiter fortgesetzt werden könnten, ist, dass es sich hier nicht um einfache Produktmaße, sondern um Prozessmaße handelt, welche von den im Kapitel 3 dargestellten Messwerkzeugen nur unzureichend ermittelt werden können.

Es reicht für diese Softwaremaße nicht, den Aufwand, den Code-Umfang oder die Anzahl der Fehler auf Projektebene bestimmen zu können, sondern diese Maße müssen im Kontext einzelner Projektphasen und Aktivitäten oder auch in Bezug auf einzelne Komponenten des zu erstellenden Produkts bestimmt werden können.

Natürlich könnten obige Maße aus den im Versionsverwaltungssystem und im Zeiterfassungssystem enthaltenen Informationen manuell und mit einigem Aufwand errechnet werden. Aber für die Integration derartiger Softwaremaße in einen Projekt-Leitstand müssen Aufwände zu einzelnen Projektaktivitäten oder Produktkomponenten automatisiert zuordenbar sein. So sollte das System erkennen können, welche Code-Bestandteile z. B. zur Datenbankschicht der zu entwickelnden Anwendung gehören, damit diese bzgl. ihres Umfanges oder der Fehlerdichte vermessen werden können. Genauso sollte das System den Aufwand, der für eine bestimmte Projektphase erbracht wurde, selbständig ermitteln können.

Da die Bezeichnungen für die zu entwickelnden Komponenten in der Regel nicht und die der Projektaktivitäten nicht notwendigerweise von vornherein feststehen, müssen generische Bezeichnungen für diese Entitäten definiert werden, welche dann

zur Definition der entsprechenden Softwaremaße verwendet werden. Gesucht ist damit eine Grundstruktur für die zu vermessenden Softwareprojekte, welche mit diesen generischen Bezeichnungen beschrieben wird.

Allgemeine Grundstruktur von Softwareprojekten

Im einfachsten Fall könnte man ein Softwareentwicklungsprojekt mittels der Struktur

Analyse — Design — Implementierung — Test — Verwaltung

beschreiben. Die Elemente dieser Grundstruktur dienen dann als Anknüpfungspunkte für die zu ermittelnden Softwaremaße. Dabei wird nicht notwendigerweise vorausgesetzt, dass sich das so beschriebene Projekt eines streng wasserfallartigen Prozessmodells bedingen würde. Die angegebene Struktur beschreibt in diesem Fall nur die Tätigkeitsarten, welche in dem Projekt vorkommen, und es können damit Softwaremaße wie

Anknüpfungspunkte für Softwaremaße

- *Design-Aufwand*,
- *aufgetretene Fehler während der Implementierung* oder
- *während der Design-Aktivitäten produzierter Code-Umfang* ($\widehat{=}$ *Umfang der Prototypen*)

definiert werden.

Um diese Softwaremaße automatisiert bestimmen zu können, müssen die zu messenden Entitäten (z. B. Produkte, Arbeitszeiten, Fehler) entsprechend mit den (abgekürzten) Bezeichnern dieser Grundstruktur, d. h. mit eindeutigen IDs, gekennzeichnet werden. Dies bedeutet im obigen Beispiel,

Kennzeichnung von Entitäten

- dass im Rahmen der Zeiterfassung Arbeitszeiten, welche mit Design-Aktivitäten verbunden sind, als solche zu kennzeichnen sind,
- dass während der Implementierung aufgetretene Fehler im Bug-Tracking-System entsprechend gekennzeichnet werden und
- dass beim *Check-in* im Versionsverwaltungssystem die Commit-Informationen mit den IDs der verwendeten Projektstruktur versehen werden.

Die Anknüpfung von Softwaremaßen an einer derartigen Grundstruktur mit generischen, d. h. projektunabhängigen Bezeichnern (IDs) für die Elemente dieser Grundstruktur, ermöglicht es, die zu ermittelnden Softwaremaße *ex ante* und unabhängig von einem konkreten Projekt zu definieren. Auch kann somit festgelegt werden, welche Softwaremaße grundsätzlich bei jedem durchzuführenden Softwareentwicklungsprojekt zur Anwendung kommen sollen, um Vergleiche zwischen den einzelnen Projekten durchzuführen.

Projektunabhängigkeit und Vergleichbarkeit

In den nächsten Abschnitten werden daher mögliche Anknüpfungspunkte für Softwaremaße dargestellt und auf ihre Eignung für eine automatisierte Softwaremessung untersucht. Es werden somit Möglichkeiten dargestellt, eine Grundstruktur für Softwareentwicklungsprojekte anzugeben, die unabhängig von einem konkreten Projekt ist, und deren Elemente als Anknüpfungspunkte für Softwaremaße verwendet werden können.

4.2 Anknüpfungspunkte für Softwaremaße

Ein Softwareentwicklungsprozess lässt sich grundsätzlich als Verbund von

- **Projekt-Rollen** (Funktionsträger, *process roles*) beschreiben, die bestimmte als
- **Aktivitäten** bezeichnete Operationen durchführen, welche auf
- **Prozess-Produkte** (*Artefakte*) angewendet werden

(vgl. [Obj05]). Neben den genannten drei Prozesselementen, werden bei der Beschreibung konkreter Vorgehensmodelle typischerweise noch

- **Phasen** und **Iterationen**, welche den Prozess gliedern,
- verwendete **Werkzeuge** und
- **Regeln** (Methoden, Anweisungen), gemäß derer die Aktivitäten durchgeführt werden,

genannt (vgl. Abbildung 4.1).

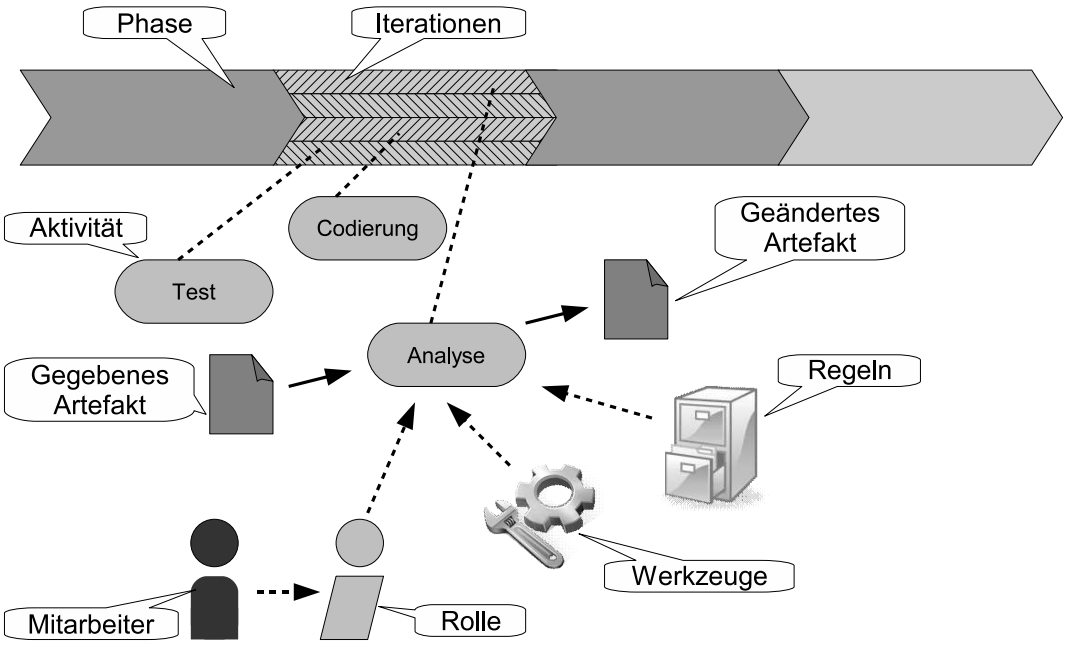


Abbildung 4.1: Grundelemente eines Softwareentwicklungsprozesses

Ausklammerung von Prozessregeln und Werkzeugen

Die Prozessregeln und die eingesetzten Werkzeuge sollen innerhalb dieser Arbeit nicht als Anknüpfungspunkte für Softwaremaße verwendet werden, da sie selbst nicht vermessen werden können. Es lassen sich lediglich die Auswirkungen ihrer Anwendung und insbesondere deren Effektivität und Effizienz beurteilen. Dazu sind jedoch die Auswirkungen dieser beiden Elemente auf die Aufwände für bestimmte Aktivitäten oder die Qualität der erstellten Artefakte zu untersuchen.

Dabei muss zwar tatsächlich festgehalten werden, nach welchen Regeln die Aktivitäten durchgeführt wurden oder welche Werkzeuge bei der Erstellung der Artefakte eingesetzt wurden. Trotzdem erscheint es nicht grundsätzlich sinnvoll, die zu messen-

den Entitäten mit IDs für die angewendeten Regeln bzw. die verwendeten Werkzeuge zu kennzeichnen. Falls derartige Untersuchungen bzgl. der Effektivität und Effizienz von Methoden und Werkzeugen durchgeführt werden sollen, so würde man eher deren Auswirkungen bei verschiedenen Projekten oder Teilprojekten betrachten. Da in diesem Fall bekannt ist, welche Aktivitäten oder Artefakte zu welchem Projekt gehören, ist eine einzelne Kennzeichnung hier nicht sinnvoll.

Grundsätzlich sind Softwaremessungen bzgl. bestimmter Projekt-Rollen denkbar, wenn untersucht werden soll, welche Kosten für bestimmte Projekt-Rollen anfallen. Dazu können jedoch viel unmittelbarer die Kosten bzw. die Aufwände für die erbrachten Aktivitäten ermittelt werden.

Keine Softwaremessung an Mitarbeitern

Die softwaremetrische Untersuchung von Projekt-Rollen führt letztendlich auch zur Betrachtung der Projektmitarbeiter, welche diese Rollen innehaben. Hier besteht die große Gefahr, dass man von Seiten der Organisationsleitung versucht ist, die Beurteilung von Mitarbeitern z. B. auf Basis der ermittelten Produktivitätsmaße durchzuführen. Da von einer solchen Vorgehensweise abzuraten ist, da sie in der Regel kein ganzheitliches Bild von der Leistung der einzelnen Mitarbeiter liefern wird, soll im Folgenden die Anknüpfung von Softwaremaßen an Projekt-Rollen bzw. an Projekt-Mitarbeitern nicht weiter betrachtet werden.

Als Anknüpfungspunkte für Softwaremaße werden innerhalb dieser Arbeit somit

- Aktivitäten,
- Artefakte und
- Projektphasen (bzw. Iterationen)

verwendet. Diese drei Möglichkeiten der Projektstrukturierung werden in den nachfolgenden Unterabschnitten 4.2.1–4.2.3 noch genauer dargestellt.

Zusätzlich wird im Abschnitt 4.2.4 mit den sogenannten *Features* eine weitere Möglichkeit dargestellt, Projekte zu strukturieren.

Die Elemente dieser derart definierten Projektstruktur liefern eine Möglichkeit, die im Rahmen des Softwareentwicklungsprozesses auftretenden Entitäten zu kennzeichnen, um sie dann im Rahmen der Softwaremessung wiedererkennen zu können.

Kontext-informationen

Da diese Angaben (Aktivität, Artefakt, Projektphase, Feature) Aufschluss darüber geben, innerhalb welchen Projektkontexts die damit gekennzeichneten Entitäten bearbeitet wurden, sollen sie im Weiteren als *Kontextinformationen* bezeichnet werden.

Elemente auf Prozess- bzw. Instanzebene

Dabei ist zu unterscheiden, ob die jeweiligen Entitäten nur auf Instanzebene oder auch auf Prozessebene existieren. Eine Projektentität auf Prozessebene ist beispielsweise die Aktivität „*Entwicklung des Datenbank-Designs*“. Diese Aktivität wird in jedem Softwareentwicklungsprozess vorkommen, bei dem eine neu zu implementierende Datenbankschicht benötigt wird. Dagegen gehört die Aktivität „*Entwicklung des Relationenschemas für die Kundendaten der Fa. XYZ*“ zu genau einem Softwareprojekt und damit zu einer bestimmten Prozessinstanz.

Softwaremaße auf Prozessebene

Softwaremaße auf Prozessebene können zwischen verschiedenen Projekten verglichen werden. Beispielsweise kann auf diese Weise festgestellt werden, ob in einem be-

stimmten Projekt der Aufwand für das Datenbank-Design außerordentlich hoch ist. Und nur derartige Softwaremaße auf Prozessebene können, wie es innerhalb dieser Arbeit vorgesehen ist, *ex ante* definiert werden, um sie dann im laufenden Projekt automatisiert erheben zu können. Da umgekehrt vor dem konkreten Projektstart noch gar nicht bekannt ist, dass für die *Kundendaten der Fa. XYZ* ein eigenes Relationenschema entwickelt werden muss, können einerseits derartige Softwaremaße auf Instanzebene nicht von vornherein festgelegt werden und andererseits ist auch keine projektübergreifende Vergleichbarkeit möglich. Es mag im Einzelfall tatsächlich sinnvoll sein, derartige Instanz-Softwaremaße zu bestimmen. Jedoch muss dies dann manuell und projektspezifisch erfolgen.

Betrachtung der projektübergreifenden Vergleichbarkeit

Grundsätzlich lässt sich sagen, dass Softwaremaße auf Prozessebene eher für strategische Ziele und Softwaremaße auf Instanzebene eher für taktische und operative Ziele verwendet werden können. Wenn sich z. B. im aktuellen Projekt herausstellen sollte, dass der Design-Aufwand für das angegebene Relationenschema besonders hoch ist, so ist zu erwarten, dass auch die Implementierungs- und Test-Aufwände für diesen Teil der Datenbank entsprechend umfangreich werden und dass dies bei der Personalplanung berücksichtigt werden muss. Dies sind operative Überlegungen, welche im Rahmen des aktuellen Projekts durchgeführt werden müssen.

Strategische und taktische Ziele der Softwaremessung

Umgekehrt können Softwaremaße auf Prozessebene strategisch eingesetzt werden, um Tendenzen im Verlauf mehrerer Softwareprojekte zu erkennen oder die eigene Prozessqualität bzw. -stabilität zu beurteilen. Da zu diesem Zweck die Maße automatisiert erhoben werden sollten, müssen die zu messenden Entitäten auch bedienerlos identifiziert werden können.

In den folgenden Abschnitten soll daher gezeigt werden, wie auf Basis von

- Aktivitäten,
- Artefakt-Funktionalitäten und
- Projektphasen

eine projektübergreifende Grundstruktur für Softwareentwicklungsumgebungen definiert werden kann, deren Elemente auch auf der Prozessebene existieren.

Im Abschnitt 4.2.4 werden die sogenannten *Features* als weitere Kontextinformation dargestellt. Diese können für die Fortschrittsmessung im Rahmen eines konkreten Projekts eingesetzt werden, existieren jedoch nicht auf Prozessebene.

4.2.1 Projektaktivitäten als Anknüpfungspunkte

4.2.1.1 Der Projektstrukturplan

Die Projektaktivitäten werden üblicherweise mittels eines *Projektstrukturplans* (engl. *Work Breakdown Structure*) hierarchisch gegliedert (vgl. [Pro04], S. 112 ff.). Nach DIN 69901 (Projektmanagement, vgl. [DIN87]) beschreibt der Projektstrukturplan (PSP) die Gesamtheit der wesentlichen Beziehungen zwischen den Elementen eines Projektes. Durch ihn wird das Projekt hierarchisch in planbare und kontrollierbare Teilaufgaben und Arbeitspakete strukturiert, wobei sich eine Baumstruktur mit Projektstrukturebenen wie in Abbildung 4.2 ergibt.

Teilaufgaben

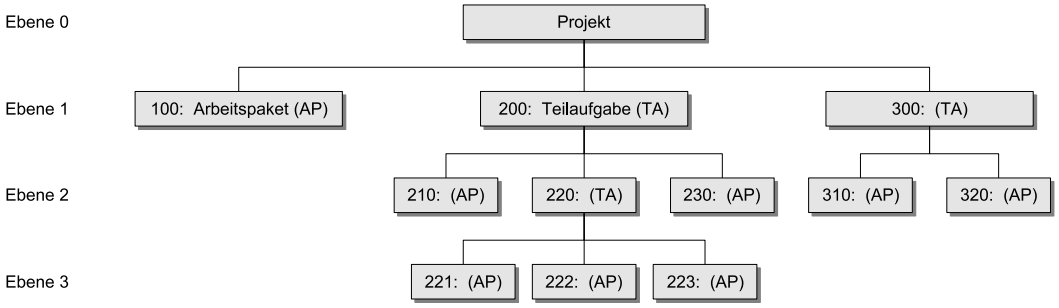


Abbildung 4.2: Projektstrukturplan zur hierarchischen Gliederung eines Projekts

Arbeitspakete

Die Arbeitspakete stellen dabei nicht weiter untergliederte Teilaufgaben auf einer beliebigen Ebene des PSP dar. Dabei sollte jedes Arbeitspaket ein in sich geschlossenes Ergebnis beinhalten. Weiter sollten der Zeit- und Kostenaufwand für ein Arbeitspaket im Vergleich zum Gesamtprojekt so gering sein, dass eine wirkungsvolle Projektsteuerung vorgenommen werden kann (vgl. [SZ05], S. 433 ff.).

Der Projektstrukturplan enthält alle Projektaktivitäten, die in den einzelnen Entwicklungsphasen durchzuführen sind, und bildet nach Burghardt [Bur02] das Fundament für die gesamte Projekt- und Produktplanung, insbesondere in Hinblick auf die Termin-, Kosten- und Einsatzmittelplanung.

Typen von Projektstrukturplänen

Dabei werden drei Arten von Projektstrukturplänen unterschieden. Beim *produktorientierten Projektstrukturplan* richtet sich die Definition der Aufgabenpakete nach der technischen Struktur des zu entwickelnden Systems. In einem *funktionsorientierten Projektstrukturplan* werden die durchzuführenden Arbeitspakete nach den Tätigkeitsfunktionen wie Analyse, Programmierung oder Testen gegliedert. Beim *ablauforientierten Projektstrukturplan* schließlich werden die Arbeitspakete gemäß der Prozessphasen bestimmt und strukturiert. Die oberste Projektstrukturebene (Ebene 1 in Abbildung 4.2) eines derartigen Projektstrukturplans spiegelt damit die Prozessabschnitte des Vorgehensmodells wider (vgl. [Bur02], S. 143 f.).

Abgrenzung des Projektstrukturplans vom Projektablaufplan

Der PSP stellt jedoch nur eine inhaltliche Gliederung der Projekttätigkeiten dar und zeigt nicht die zeitliche Abfolge der Aktivitäten, wie dies beispielsweise in einem *Gantt-Diagramm* der Fall ist (vgl. [SZ05], S. 433 ff.). Er ist dafür in der Regel ausführlicher hierarchisch gegliedert, da er als Basis für das Controlling eingesetzt wird. Diesbezüglich ist der PSP auch viel stabiler als ein Projektablaufplan, da Änderungen in Bezug auf Termine, Ablaufreihenfolgen und Ressourcen keine Auswirkungen auf den PSP haben.

Üblicherweise werden die Projektablaufpläne (Netzpläne, Gantt-Diagramme) auf der Grundlage des Projektstrukturplanes unter Verwendung der *Netzplantechnik* entwickelt. Dabei wird zunächst ermittelt, welche Abhängigkeiten zwischen den Arbeitspaketen bestehen und welche Vorgänge parallel, nacheinander oder unabhängig voneinander ablaufen können. Zusammen mit Informationen über die für die Arbeitspakete benötigten Ressourcen und evtl. vorgegebene Fristen und Termine wird aus diesen Informationen rechnerisch ein entsprechender Netzplan entwickelt

(vgl. [SZ05], S. 433 ff., und [Hen02], S. 302 f.).

In Unternehmen, in denen häufig ähnliche Entwicklungen immer wieder durchgeführt werden, wie es insbesondere in der Softwareentwicklung der Fall ist, bietet es sich an, Standardstrukturen für gleichartige Entwicklungsprojekte zu entwerfen.

Standard-Projektstrukturplan

Man benutzt diese Standard-Projektstrukturpläne als „Checklisten-Struktur“, aus der die zutreffenden Strukturzweige mit den zugehörigen Arbeitspaketen übernommen und nicht zutreffende Teile übergangen werden. Der eigentliche individuelle Projektstrukturplan entsteht aus dieser Standardstruktur als „erweiterte Untermenge“ durch entsprechende Anpassungen. Dabei fehlen einerseits dem angepassten Projektstrukturplan einige Arbeitspakete, andererseits wird er auch um projektspezifische Arbeitspakete ergänzt (vgl. [Bur02], S. 145). Somit kann ein Standard-Projektstrukturplan für Softwareprojekte als generische Grundstruktur zur Anknüpfung von Softwaremaßen verwendet werden.

PSP-Codes

Üblicherweise wird jedes Element eines Projektstrukturplans mit einer eindeutigen alphanumerischen Kennzeichnung, den sogenannten PSP-Codes, versehen. Bei der *dekadischen Codierung* (vgl. Abbildung 4.2) gibt die Anzahl der Ziffern ungleich Null im PSP-Code die Einordnung des PSP-Elements innerhalb der hierarchischen Struktur des Projektstrukturplans an. Diese PSP-Nummern können damit aufgrund ihrer Eindeutigkeit als Anknüpfungspunkte für Softwaremaße fungieren.

Kennzeichnung von Aktivitäten

Beispielsweise empfehlen Patzak und Rattay ([PR98]), die Ebene 1 des Projektstrukturplanes (siehe Abbildung 4.2 auf S. 91) nach Projekt- bzw. Aktivitätsfunktionen zu gliedern. Im Rahmen der Softwareentwicklung ergibt sich daraus eine Gliederung nach den Teilaufgaben *100–Analyse*, *200–Design* usw. Somit würde allen im Rahmen des Projekts anfallenden Design-Aktivitäten ein PSP-Code zwischen 200 und 299 zugeordnet werden. Damit können diese in der Zeiterfassung entsprechend gekennzeichnet werden, um eine spätere Auswertung zu ermöglichen.

Kennzeichnung von Artefakten

Die PSP-Codes können jedoch nicht nur zur Identifikation von Teilaufgaben und Arbeitspaketen des Projektstrukturplans, sondern auch zur Kennzeichnung der innerhalb der jeweiligen Arbeitspakete erstellten oder veränderten Artefakte verwendet werden. Dazu ist es notwendig, dass die Artefakte in einem Versionsverwaltungssystem versioniert werden. Dann kann, immer wenn für eine veränderte Version eines Artefakts ein *Commit* im Versionsverwaltungssystem durchgeführt wird, mit der Commit-Information der PSP-Code des Arbeitspakets gespeichert werden (vgl. [DHW06a], [DHW06c]).

Somit sind über die PSP-Codes in den Commit-Informationen des Versionsverwaltungssystems die Artefakte und auch die Veränderungen von Artefakten mit den entsprechenden Aktivitäten im Projektstrukturplan verknüpft, so dass Softwaremaße, die an Elementen des Standard-Projektstrukturplans verankert wurden, automatisiert erhoben werden können.

Voraussetzungen für den Standard-PSP

Ein derartiger Standard-Projektstrukturplan hängt naturgemäß sehr stark von dem zu Grunde liegenden Vorgehensmodell ab. Nur wenn dieses eine hinreichend detaillierte, hierarchische Tätigkeitsstruktur aufweist, fällt auch der Standard-PSP entsprechend detailliert und umfangreich aus. Dies wiederum ermöglicht auch die Definition feingranularer Softwaremaße. Eine Voraussetzung für die effektive An-

wendbarkeit dieses Ansatzes ist, dass ein möglichst großer Teil des projektspezifischen PSP mit dem Standard-PSP übereinstimmt, da über diesen die Softwaremaße bestimmt werden, welche projektunabhängig verwendbar sein sollen.

Ein Beispiel für ein Vorgehensmodell, welches diese Anforderung bzgl. der Detaillierung der Tätigkeitsstruktur grundsätzlich erfüllt, ist das *V-Modell XT* (siehe Anhang A.1, S. 173 ff.).

4.2.1.2 Der Standard-PSP am Beispiel des V-Modell XT

Das V-Modell XT weist eine sehr detailliert beschriebene und hierarchisch organisierte Tätigkeitsstruktur auf (vgl. A.1.2, S. 173 ff.). Die *Vorgehensbausteine*, welche als grundlegende, strukturierende Einheiten dieses Vorgehensmodells dienen, bieten sich somit unmittelbar zur Gliederung der Ebene 1 des Standard-Projektstrukturplans an, wodurch sich auf dieser Ebene eine funktionsorientierte Gliederung des Projekts ergibt, wie sie auch weiter oben empfohlen wird. Konkret ergibt sich für ein Systementwicklungsprojekt auf Auftragnehmerseite eine Gliederung des PSP gemäß der in Tabelle A.1 (S. 176) dargestellten Vorgehensbausteine.

Vorgehensbausteine auf Ebene 1 des Standard-PSP

Die Aktivitätsgruppen des V-Modell XT dienen eher der „Katalogisierung“ der Aktivitäten als zur Projektgliederung (siehe S. 176). So verteilen sich die Aktivitäten der Aktivitätsgruppe *Systemelemente* auf drei verschiedene Vorgehensbausteine (vgl. Abbildung A.3). Da zudem jede Aktivität nicht nur genau einem Vorgehensbaustein, sondern auch genau einer Aktivitätsgruppe zugeordnet ist, empfiehlt es sich, die zweite Ebene des Standard-Projektstrukturplanes nach den Aktivitäten zu gliedern und von einer Gliederung nach Aktivitätsgruppen abzusehen. Falls dennoch eine Analyse von Projektmaßen auf Ebene der Aktivitätsgruppen nötig sein sollte, kann diese aufgrund der Zuordenbarkeit jederzeit durchgeführt werden.

Aktivitäten auf Ebene 2 des Standard-PSP

Einige der Aktivitäten des V-Modells XT sind noch zusätzlich in Teilaktivitäten gegliedert (vgl. Abbildung A.1, S. 174). Diese Gliederung kann bei Bedarf mit in den Standard-PSP auf der Gliederungsebene 3 übernommen werden, so dass die Arbeitspakete dann entweder auf der Ebene 4 oder auf der Ebene 3 parallel zu evtl. vorhandenen Teilaktivitäten zu liegen kämen. Allerdings hängt eine derartige Detaillierung sicherlich von der Art und der Größenordnung der innerhalb der Organisation typischerweise durchgeführten IT-Projekte ab.

Ebene 3 und Ebene 4 des Standard-PSP

Beispielsweise lassen sich der Aktivität „Datenbankentwurf erstellen“ grundsätzlich immer die Themen „Technisches Datenmodell ableiten“ und „(physikalische) Struktur der Datenbank entwerfen“ als Teilaktivitäten zuordnen. Ob jedoch die Aktivität „Make-or-Buy-Entscheidung durchführen“ für jedes Projekt weiter zerlegt werden muss, erscheint fraglich.

Da die Arbeitspakete selbst grundsätzlich von der Instanz des Softwareentwicklungsprojektes abhängig sind, gehören diese nicht zum Standard-PSP. Auch auf Instanzebene ist es in der Regel nicht möglich, bereits zu Projektbeginn einen bis auf die Ebene der Arbeitspakete ausgearbeiteten Projektstrukturplan zu erstellen (vgl. [SZ05], S. 433 ff.). Stattdessen wird hier gewöhnlich vorgeschlagen, den Projektstrukturplan phasenweise bzw. iterationsweise fortzuschreiben.

Arbeitspakete gehören zum Instanz-PSP

Kritische Beurteilung

Grundsätzlich ist das V-Modell XT aufgrund seiner hierarchischen Struktur gut für die Konstruktion eines Standard-PSP geeignet. Insbesondere wurde bei der Entwicklung des V-Modell XT darauf geachtet, mit diesem Modell möglichst alle Belange einer großen Bandbreite von Projektarten (Softwareentwicklung, Hardwareentwicklung, externe Vergabe von Entwicklungsprojekten) abzudecken. So umfasst das V-Modell XT derzeit 100 verschiedene Aktivitäten.[1] Allerdings entspricht die Gliederung des V-Modells XT nicht notwendigerweise den Erwartungen des typischen Softwareentwicklers.

Praxisfremde Gliederung

So unterscheidet ein Softwareentwickler nicht unbedingt, ob er gerade

- ein *SW-Modul realisiert*, d. h. an einer einzelnen Sourcecode-Datei (z. B. einer Java-Klasse) arbeitet,
- mehrere SW-Module zu einer *SW-Komponente integriert*, d. h. die Beziehungen mehrerer Sourcecode-Dateien organisiert, oder
- mehrere SW-Komponenten zu einer *SW-Einheit integriert*, welche sich eigentlich nur durch die Größe bzw. den Funktionsumfang von einer SW-Komponente unterscheidet (vgl. Abbildung A.2, S. 175).

Ein Softwareentwickler erstellt aus seiner Sicht einfach nur Software und hat wenig Interesse, bzgl. der drei genannten Aktivitäten zu differenzieren und diese bei der Zeiterfassung getrennt zu erfassen.

Auch ist nicht unmittelbar einsichtig, dass sich die Software-Test-Aktivitäten im Vorgehensbaustein *Systemerstellung* (siehe S. 178) verbergen und nicht zu den Vorgehensbausteinen *SW-Entwicklung* oder *Qualitätssicherung* gehören. Innerhalb des Vorgehensbausteins *Qualitätssicherung* werden Dokumente und Prozesse geprüft, QS-Berichte erstellt und das QS-Handbuch gepflegt. Die Aktivitäten eines Teams, dass sich beispielsweise im Großrechnerbereich oder bei Systemen wie SAP R/3 um die Übertragung der aktuellen Programmstände von der Entwicklungsumgebung in die Integrationstestumgebung und schließlich in die Produktionsumgebung kümmert, fehlen völlig.

Geringe Berücksichtigung von Test-Aktivitäten

Softwaretests gibt es nur in sehr abstrakter Form im Vorgehensbaustein *Systemerstellung* mit der Aktivität *Systemelement prüfen*. Die Erstellung von Unit-Tests ist dabei unter dem Begriff „Prüfprozedur Systemelement realisieren“ (vgl. [BMI06], S. 561) verborgen.

Abgesehen von dieser für viele Softwareentwickler vermutlich gewöhnungsbedürftigen Gliederung des V-Modell XT ist es auch nicht einfach, PSP-Codes für einen auf dem V-Modell XT basierenden Projektstrukturplan zu definieren. Da es 21 Vorgehensbausteine gibt und bis zu 18 Aktivitäten in einem Vorgehensbaustein enthalten sind, ist ein dekadisches Nummerierungsschema, wie in Abbildung 4.2 dargestellt, nicht anwendbar. Stattdessen können in diesem Fall die Knoten des Standard-PSP mit entsprechenden Abkürzungen gekennzeichnet werden. Beispielsweise ist für die Aktivität *SW-Modul realisieren* innerhalb der Teilaufgabe *SW-Entwicklung* der PSP-Code *SWE-ModulRealisation* denkbar.

Definition geeigneter PSP-Codes

[1] V-Modell XT, Version 1.2

Diese genannten Einwände ändern jedoch nichts an der grundsätzlichen Eignung des V-Modell XT zur Herleitung eines Standard-PSP.

4.2.1.3 Der Standard-PSP am Beispiel des Rational Unified Process

Ein weiteres Vorgehensmodell, auf dessen Basis sich ein Standard-PSP definieren lässt, ist der *Rational Unified Process* (siehe Abschnitt A.2, S. 179 ff.).

Kein Projektstrukturplan im RUP

Unter den im Rahmen des *Rational Unified Process* (RUP) zu erzeugenden Artefakten ist ein Projektstrukturplan zunächst einmal nicht enthalten (vgl. [KK05], S. 223 f., [Kru04], S. 115 f.). Kroll und Kruchten erklären dies damit, dass Projekte, welche einen sich an den auszuliefernden Produkten orientierenden Projektstrukturplan verwenden, auch automatisch bzgl. der Produktstruktur organisiert werden würden. Diese führe wiederum dazu, dass die Produktstruktur zu früh ausgearbeitet wird, wenn noch wenig über das zu entwickelnde Produkt bekannt ist.

Allerdings geht es im Rahmen dieser Arbeit nicht um einen produktorientierten, sondern um einen funktionsorientierten Projektstrukturplan. Dieser muss tatsächlich nicht im Rahmen des Softwareentwicklungsprojekts erstellt werden, da ein Standard-PSP implizit bereits durch die statische Struktur des RUP mit seinen Core-Worklows, Workflow-Details und Aktivitäten gegeben ist (siehe Abschnitt A.2.2, S. 180).

PMBOK Guide

Gemäß dem *PMBOK Guide* [Pro04] gehört die Verwaltung des PSP zum Wissensgebiet des *Project Scope Managements*. Die Weiterentwicklung des PSP und insbesondere die detaillierte Definition von Arbeitspaketen finden in jeder Projektphase statt. Dieser Sachverhalt wird im Rahmen des RUP dadurch berücksichtigt, dass einerseits ein Standard-PSP implizit vorhanden ist und somit nicht erstellt werden muss. Andererseits wird die iterative Definition von Arbeitspaketen, wodurch sich ein Projekt-PSP letztendlich von einem Standard-PSP unterscheidet, beim RUP durch die Erstellung von Iterationsplänen (siehe S. 182) für die jeweils folgende Iteration ersetzt.

Impliziter Standard-PSP beim RUP

Der Standard-PSP auf Basis des RUP enthält somit als Teilaufgaben auf Ebene 1 die neun Disziplinen des RUP (Core Workflows, siehe Abbildung A.5 auf S. 180 und Beschreibung auf S. 180).[2] Auf Ebene 2 werden dann zur Gliederung die Workflow-Details verwendet (siehe S. 180), da diese den Core-Workflows eindeutig zugeordnet sind, was für die Aktivitäten des RUP nicht gilt.

Grundsätzlich ist es denkbar, die Aktivitäten als Gliederung der Ebene 3 des Standard-PSP zu verwenden, so dass nicht nur die Workflow-Details, sondern auch die einzelnen Aktivitäten selbst als Kontextinformationen zur Kennzeichnung der zu messenden Entitäten zur Verfügung stehen. Allerdings kann eine derartige Kennzeichnung oftmals zu feingranular sein. Die einzelnen Projektbeteiligten sind sicher-

[2] Interessanterweise hat diese Strukturierung große Ähnlichkeit mit den beim V-Modell XT im Vorgehensbaustein *Systemerstellung* enthaltenen Aktivitätsgruppen (vgl. S. 178). Laut [BMI06] definiert dieser Vorgehensbaustein das „Grundgerüst der Systementwicklung, auf dem weitere Vorgehensbausteine wie SW-Entwicklung oder HW-Entwicklung aufbauen“ (siehe [BMI06], S. 107).

lich in der Lage, bei der Zeiterfassung zu vermerken, für welche Disziplinen bzw. Workflow-Details sie entsprechende Arbeitszeiten aufgewendet haben.

Probleme durch zu hohe Granularität

Ein Detaillierungsgrad auf der Ebene der einzelnen Aktivitäten könnte jedoch einen zu großen Erfassungsaufwand darstellen. Insbesondere lassen sich die im RUP beschriebenen Aktivitäten bei der praktischen Durchführung teilweise nur schwer voneinander abgrenzen. Hier bestünde somit die Gefahr, dass die Erfassung der Arbeitszeiten durch den Mitarbeiter bewusst oder unbewusst fehlerhaft erfolgt (vgl. Abschnitt 2.3.2), so dass die gewonnen Daten nicht verwendbar sind.

Auch sinkt bei dieser Granularität die Vergleichbarkeit zwischen den Projekten, da manche Aktivitäten evtl. nur in einzelnen Projekten vorkommen. Hinzu kommt, dass einige Aktivitäten in mehreren Workflow-Details auftreten können. Wollte man also die erfassten Daten auf der Ebene der Workflow-Details auswerten, müsste zusätzlich zur Aktivitäts-Kontextinformation auch ein Verweis auf das zugehörige Workflow-Detail erfasst werden.

4.2.2 Artefakt-Funktionalitäten als Anknüpfungspunkte

Neben den Projektaktivitäten können auch die Funktionalitäten der im Rahmen des Softwareentwicklungsprojektes erzeugten Artefakte als Anknüpfungspunkte für die Softwaremessung verwendet werden. Dabei muss zwischen projektinternen Artefakten und auszuliefernden Artefakten (engl. *Deliverables*) unterschieden werden. Bei den projektinternen Artefakten handelt es sich hauptsächlich um Dokumente, welche im Laufe des Projektes erzeugt werden, wie beispielsweise Projektpläne, Stücklisten, Prüfkonzepte usw. In der Regel sind die zu erstellenden Dokumente durch das eingesetzte Vorgehensmodell vorgegeben. Allerdings kann bei diesen Dokumentarten nur die Größe bzw. der Umfang automatisiert bestimmt werden, was keine besonders aussagekräftigen Informationen liefert.

Projektinterne Artefakte und Deliverables

Im Rahmen des Projekt-Controllings viel interessanter sind dagegen Softwaremaße, welche an den auszuliefernden Artefakten anknüpfen, wie z. B. der Aufwand, der in die Entwicklung der Datenbank-Komponente des zu entwickelnden Systems gesteckt wurde, oder der Umfang der Anwendungslogik.

Da zum Zeitpunkt der Definition der zu erhebenden Softwaremaße die tatsächlich zu entwickelnden und auszuliefernden Artefakte nicht bekannt sind, muss die von ihnen zur Verfügung gestellte Funktionalität als Kontextinformation für die Anknüpfung von Softwaremaßen verwendet werden. Eine typische Kategorisierung der Funktionalität der verschiedenen Systemkomponenten wird durch die Funktionsbeschreibungen

Artefakt-Funktionalitäten als Anknüpfungspunkte

- Anwendungslogik,
- Präsentationsschicht,
- Datenhaltung,
- Transaktionsverwaltung und
- Benutzer- und Berechtigungsverwaltung

gegeben.

Diesen Funktionalitäten können IDs zugeordnet werden, welche zur Kennzeichnung der verschiedenen Entitäten mit Kontextinformationen dienen. So können die Zeiterfassungsdaten um Informationen über die Funktionalität der im angegebenen Zeitraum erstellten Komponenten ergänzt werden. Daneben können auch die Sourcecode-Dateien oder die Commit-Informationen im Versionsverwaltungssystem um diese Kontextinformationen ergänzt werden, um eine Umfangsmessung auf Dateiebene bzw. pro Versionierungs-Commit zu ermöglichen.

Anwendbare Softwaremaße

Dadurch ist es möglich, Maße wie den Aufwand für die Implementierung einzelner Funktionalitäten (z. B. Präsentationsschicht) zu bestimmen. Auch lässt sich prüfen, ob Programmteile, welche bestimmte Funktionalitäten realisieren, überdurchschnittlich häufig im Laufe der Entwicklung modifiziert werden und daher als „sensibel“ eingestuft werden müssen.

4.2.3 Projektphasen als Anknüpfungspunkte

Schließlich können auch die Elemente der dynamischen Prozessstruktur selbst als Anknüpfungspunkte für Softwaremaße verwendet werden, indem Arbeitszeitdatensätze, Artefakte im Versionsverwaltungssystem oder Fehlermeldungen mit diesen entsprechend gekennzeichnet werden.

Die Struktur eines Softwareentwicklungsprozesses kann, wie in Abschnitt A.2.2 (S. 179) dargestellt, hinsichtlich einer statischen Dimension, welche die einzelnen Prozessaktivitäten umfasst, und einer dynamischen Dimension, welche den zeitlichen Ablauf des Entwicklungsprozesses beschreibt, gegliedert werden. Da die statischen Strukturelemente bereits bei der Verwendung von Aktivitäten als Anknüpfungspunkte (Abschnitt 4.2.1) behandelt wurden, sollen hier nun die Elemente der Ablaufgliederung betrachtet werden.

So kennen Vorgehensmodelle wie der RUP (siehe Abschnitt A.2, S. 179) oder das *Feature-Driven Development* (siehe Abschnitt 4.2.4, S. 98) definierte Phasen auf Prozessebene, welche somit in allen Instanzen dieser Prozesse vorhanden sind. Beim RUP sind dies namentlich

Projektphasen beim RUP

- die Konzeptionsphase (*Inception*),
- die Ausarbeitungsphase (*Elaboration*),
- die Konstruktionsphase (*Construction*) und
- die Übergabephase (*Transition*),

welche den zeitlichen Ablauf des Projektes gliedern (siehe Abbildung A.5, S. 180).

Das Ende jeder dieser Phasen ist mit einem Projekt-Meilenstein (siehe S. 42) verbunden, für dessen Erreichen bestimmte Kriterien erfüllt sein müssen (vgl. [Kru04]). Beispielsweise muss am Ende der Inception-Phase Klarheit über die Projektziele und den Projektumfang herrschen (*Lifecycle Objective Milestone*) und es muss zum Ende der Elaboration-Phase die Architektur der Anwendung feststehen (*Lifecycle Architecture Milestone*).

An den Meilensteinen am Ende jeder Phase werden wichtige Entscheidungen bzgl. des weiteren Projektverlaufes getroffen. Insbesondere wird hier jeweils über die Projektfortführung an sich, über Veränderungen des Projektinhalts oder über Anpas-

sungen des Zeitplans entschieden.

Projektphasen als Kontextinformation

Somit ist es auch sinnvoll, den in den einzelnen Projektphasen erbrachten Aufwand auswerten zu können. Hierzu muss die Angabe der jeweiligen Phase als Kontextinformation bei der Zeiterfassung mit aufgenommen werden.

Iterationen als Kontextinformation

Als Kontextinformation nur bedingt geeignet ist jedoch die Iteration, innerhalb derer Tätigkeiten erbracht oder bestimmte Artefakte erzeugt wurden. Die Anzahl und Inhalte der Iterationen werden erst kurzfristig im Laufe der Projektdurchführung geplant. Daher sind Vergleiche bezüglich der Arbeitsaufwände, welche in einzelnen Iterationen erbracht wurden, zwischen zwei Projekten nicht sinnvoll. Dagegen kann innerhalb eines Projekts die Untersuchung der Aufwandsverteilung auf die einzelnen Iterationen durchaus Aufschluss über die Güte der Iterationsplan geben.

Keine vordefinierten Projektphasen beim V-Modell XT

Innerhalb des V-Modell XT werden dagegen auf Prozessebene keine definierten Projektphasen beschrieben.[3] Dies mag an der allgemeineren Ausrichtung dieses Vorgehensmodells liegen, welches nicht nur für die Durchführung von Softwareentwicklungsprojekten, sondern als allgemeines Vorgehensmodell für eine Vielzahl verschiedenartiger Systementwicklungsprojekte (z. B. auch Hardwareentwicklung) konzipiert ist. Daher gibt es auf der Prozessebene auch keine definierten Projektmeilensteine, sondern die projektspezifischen Meilensteine werden erst im Rahmen der Projektdurchführungsplanung bestimmt (vgl. [BMI06], S. 419).

Projektdurchführungsstrategien beim V-Modell XT

Es gibt zwar sogenannte *Projektdurchführungsstrategien*, welche eine Reihenfolge festlegen, in der die für das Projekt relevanten Entscheidungspunkte durchlaufen werden müssen (vgl. [BMI06], S. 603). Diese Entscheidungspunkte regeln jedoch im Wesentlichen nur rechtliche Aspekte, wie die Genehmigung, Ausschreibung, Beauftragung und die Abnahme eines Projektes, und sind nicht zur Kontrolle des Verlaufs der Projektaktivitäten gedacht.

Ungeachtet dieser Ausführungen steht es den einzelnen Organisationen, welche das V-Modell XT verwenden, natürlich frei, selbst obligatorische Projektphasen mit entsprechenden Projektmeilensteinen zu definieren. Dies erscheint insbesondere dann sinnvoll, wenn regelmäßig gleichartige Entwicklungsprojekte durchgeführt werden sollen.

4.2.4 Features als Anknüpfungspunkte

4.2.4.1 Feature-Driven Development

Neben den Artefakt-Funktionalitäten können auch sogenannte *Features* als Kontextinformationen für die Anknüpfung von Softwaremaßen verwendet werden. Diese Möglichkeit und die Abgrenzung der Features von den Artefakt-Funktionalitäten soll anhand des *Feature-Driven Developments* (FDD) dargestellt werden. Dabei handelt es sich um ein iteratives Vorgehensmodell für Softwareentwicklungsprojekte, welches zu den agilen Entwicklungsmethoden gezählt wird. Die Methode wurde zum ersten

[3] Das V-Modell XT definiert vier Systemlebenszyklusausschnitte (Entwicklung, Wartung und Pflege, Weiterentwicklung und Migration). Diese beziehen sich jedoch auf den gesamten Lebenszyklus des zu entwickelnden Systems und nicht auf das Entwicklungsprojekt (vgl. [BMI06], S. 80).

Mal von De Luca in [CLD99] veröffentlicht. Eine ausführlichere Beschreibung findet sich in [PF02].[4] Oestereich und Weiß verwenden in [OW08] zwar ebenfalls *Features* zur Projektgliederung, betonen jedoch, dass ihr Ansatz sich deutlich von demjenigen De Lucas unterscheide.

Feature als kundenwertige Funktionalität

Das FDD basiert auf acht sogenannten *Best Practices*, wobei eine dieser Praktiken als *Developing by Feature* bezeichnet wird. Ein *Feature* ist dabei eine Funktionalität, welche für den Kunden einen unmittelbaren Wert hat und in maximal zwei Wochen implementiert werden kann.

Probleme aufgrund funktionaler Zerlegung

Dabei wird, wie bei vielen anderen Vorgehensmodellen auch, die Gesamtfunktionalität des zu entwickelnden Systems mittels funktionaler Zerlegung in handlichere Teilprobleme zerlegt. In [PF02] wird jedoch als Problem vieler durch funktionale Anforderungen gesteuerter Entwicklungsprozesse dargestellt, dass bei den dort verwendeten funktionalen Bausteinen (z. B. *Use Cases*, *User Stories*, Datenfluss-Diagramme usw.) sehr oft die Geschäftslogik mit Aspekten der Benutzerschnittstelle, der Datenhaltung und der Netzwerkkommunikation vermischt werden würde. Wenn derartige Zusammenstellungen funktionaler Anforderungen zu Arbeitspaketen geschnürt und an die Entwickler übergeben werden, resultiere dies oft darin, dass die Entwickler einen großen Teil der Zeit mit der Implementierung von technischen Funktionalitäten, wie der Implementierung von *Enterprise Java Beans* oder der Definition des O/R-Mappings, verbrächten und somit die eigentlichen Geschäftsanwendungsfunktionalitäten vernachlässigten.

Daher wird beim FDD die Liste der funktionalen Anforderungen auf solche mit einem Wert für den Anwender oder den Kunden beschränkt. Zusätzlich werden die Anforderungen so formuliert, dass sie vom Anwender bzw. vom Kunden verstanden werden können. Derartige funktionale Anforderungen werden beim FDD als *kundenwertige Features* bezeichnet und dienen zur Steuerung und Nachverfolgung des Projektfortschrittes. Dies bedeutet z. B. auch, dass die Entwicklung von Infrastrukturkomponenten (z. B. einer O/R-Mapping-Komponente) nicht mit zum Projektfortschritt zählt, da sie keinen unmittelbaren Beitrag zu den Geschäftsanwendungsfunktionalitäten liefern und somit für den Kunden nicht relevant sind. Gegenüber dem Kunden wird der Projektfortschritt somit nur in Form von implementierten Features dokumentiert, welche dieser verstehen und bewerten kann. Dabei kann der Kunde einzelne Features in Bezug auf ihre Wichtigkeit für die Geschäftsanwendungsfunktionalität auch priorisieren.

Implementierung des technischen Unterbaus zählt nicht zum Projektfortschritt

4.2.4.2 Eigenschaften von Features

Features werden beim FDD grundsätzlich in der Form

<Aktivität> <Arbeitsergebnis> <Objekt>

formuliert, wobei in die jeweiligen Formulierungen noch geeignete Artikel und Präpositionen eingefügt werden, um vollständige Sätze zu erhalten. Hierbei kann das

[4] Da das Feature-Driven Development als Vorgehensmodell relativ unbekannt ist, wird es – im Gegensatz zum V-Modell XT und zum RUP – im Hauptteil dieser Arbeit vorgestellt.

Objekt eine Person, ein Ort oder eine Sache sein.

Beispiel 7 (Features)

- *Berechne die Gesamtsumme des Verkaufs.*
- *Ermittle das Guthaben des Bankkontos.*
- *Autorisiere eine Kreditkartentransaktion des Karteninhabers.*

Features sind feingranular

Features sollen so klein sein, dass sie innerhalb von zwei Wochen implementiert werden können, wobei eine Implementierungsdauer von zwei Wochen als obere Grenze gesehen wird. Funktionalitäten, die nicht innerhalb dieses Zeitraumes implementiert werden können, müssen soweit zerlegt werden, dass die so erhaltenen Teil-Funktionalitäten jeweils klein genug sind, um die Feature-Eigenschaften zu erfüllen.

Es gibt auch Features, welche innerhalb weniger Stunden oder Tage implementiert werden können. Allerdings muss durch ein Feature mehr implementiert werden als z. B. nur die *Accessor*-Methoden einer *Java-Bean*.

Sowohl obige Beispiele als auch die genannte Granularität, welche für Features gelten soll, machen deutlich, dass ein Feature eine Bedeutung innerhalb der Geschäftsprozesse des Kunden und damit einen Wert für den Kunden haben muss. Ein Feature korrespondiert üblicherweise mit einem Ablaufschritt innerhalb eines Geschäftsprozesses und hat somit eine unmittelbare Bedeutung für den Kunden.

Features vs. XP User Stories

Somit haben die Features des FDD eine große Ähnlichkeit zu den beim *eXtreme Programming* (XP, [Bec04]) verwendeten *User Stories*. In [BF01] wird die User Story als „*a chunk of functionality that is of value to the customer*“ (siehe [BF01], S. 45) beschrieben. Auch wird hier die User Story als *Einheit der Funktionalität* bezeichnet, mit der der Fortschritt demonstriert wird, indem getesteter und integrierter Code ausgeliefert wird, der eine User Story implementiert. Der Hauptunterschied zwischen einem FDD Feature und einer XP User Story besteht jedoch darin, dass Features grundsätzlich in der oben angegebenen Form formuliert werden.

Zusätzlich werden Features strenger verwaltet als dies bei den User Stories des XP üblich ist, da sie hierarchisch organisiert werden. Features, welche zueinander in Beziehung stehen, werden zu *Feature-Sets* und diese wiederum zu *Major-Feature-Sets* zusammengefasst (vgl. [CLD99]).

Features vs. Use Cases

Auch mit den *Use Cases* des *Rational Unified Process* haben die FDD Features eine gewisse Ähnlichkeit. Use Cases wurden von Jacobson bereits im Jahr 1992 als „*a description of a set of sequence of actions, including variants, that a system performs that yields an observable result to a particular actor*“ (siehe [Jac92]) beschrieben, wobei auch bei dieser Definition betont wird, dass der Use Case (ähnlich dem Feature) ein signifikantes Ergebnis für den Kunden liefert.

Laut [PF02] unterscheiden sich die Use Cases von Features dadurch, dass es für die Modellierung von Use Cases keine Empfehlungen gebe, mit welcher Granularität Use Cases modelliert werden sollen und in welchem Format und welcher Detailliertheit sie notiert werden sollen. Im Ergebnis würden Use Case Sammlungen oft

inkonsistent und unvollständig sein und die einzelnen Use Cases hätten unterschiedliche Granularitäts- und Detaillierungsgrade. Zusätzlich würden dabei oft Details der Benutzerschnittstelle und der Persistenz-Funktionalität mit der Geschäftslogik vermischt werden. Aber auch im Ablauf des Softwareentwicklungsprozesses gibt es Unterschiede. So ist der RUP als „*use case driven*“ beschrieben, d. h. die Ermittlung der funktionalen Anforderungen in einem auf dem RUP basierenden Softwareentwicklungsprozess erfolgt durch die Modellierung von Use Cases. Aus den Use Cases werden dann Aktivitäts- und Klassendiagramme abgeleitet. Beim FDD dagegen beginnt die Softwareentwicklung mit der Modellierung eines *Domain Object Models* (siehe folgender Abschnitt), mit dessen Hilfe im nächsten Schritt die Feature-Liste erarbeitet wird.

4.2.4.3 Die FDD-Aktivitäten

Beim FDD werden im Rahmen des Softwareentwicklungsprozesses die folgenden fünf Aktivitäten (*FDD Processes*, vgl. Abbildung 4.3) ausgeführt (vgl. [CLD99]):

1. *Develop an Overall Model*
   Innerhalb dieser Aktivität wird ein sogenanntes *Domain Object Model* erstellt. Dabei handelt es sich im Wesentlichen um eine Sammlung von UML-Klassendiagrammen, welche die Beziehungen der Business-Objekte der zu entwickelnden Anwendung beschreiben. Die Klassendiagramme können um bereits identifizierte Attribute und Methoden ergänzt werden. Zusätzlich kann die Funktionalität der Methoden mittels Sequenz-Diagrammen verdeutlicht werden.

   Domain Object Model

2. *Build a Features List*
   Das im ersten Schritt erworbene Wissen über die Anwendung wird nun verwendet, um die Feature-Liste zu erstellen. Dazu wird die Anwendungsdomäne in Sachgebiete (z. B. Kundenverwaltung, Versand, Fakturierung) zerlegt, welche verschiedene Geschäftsaktivitäten umfassen. Die einzelnen Schritte der Geschäftsaktivitäten wiederum liefern die entsprechend kategorisierten Feature-Listen.

3. *Plan by Feature*
   Während dieser Aktivität wird die Reihenfolge, in der die Features implementiert werden sollen, festgelegt. Kriterien für die Planung sind Abhängigkeiten zwischen den Features, Auslastung der einzelnen Entwicklungsteams und die Komplexität der einzelnen Features. Die einzelnen Features werden dabei zu Feature-Sets und diese wiederum zu Major-Feature-Sets zusammengefasst. Dabei werden jeweils Fertigstellungstermine geschätzt.

   Feature-Sets

4. *Design by Feature*
   Diese Aktivität beginnt mit einem sogenannten *Domain Walkthrough*, bei dem innerhalb des *Domain Object Models* diejenigen Klassen identifiziert werden, welche mit den aktuell zu implementierenden Features verknüpft sind. Für die

Überarbeiten der Klassendiagramme und Erzeugen von Sequenz-diagrammen

einzelnen Features oder auch für mehrere zu einem Feature-Set zusammengefasste Features werden detaillierte Sequenz-Diagramme entwickelt. Dabei wird auch das *Domain Object Model* überarbeitet und die betroffenen Klassendiagramme detailliert ausgearbeitet. Anschließend wird eine Inspektion des erarbeiteten Designs durchgeführt. Das Ergebnis dieser Tätigkeiten wird als *Design-Paket* bezeichnet.

5. *Build by Feature*
   Eigentliche Implementierung
   Gemäß des in der vorhergehenden Aktivität entwickelten Designs wird ein kundenwertiges Feature implementiert. Die dabei erstellte Software wird einer Code-Inspektion unterzogen und mittels Unit-Tests überprüft. Abschließend wird der Code im Versionsverwaltungssystem als „freigegeben“ gekennzeichnet (*Promote to Build*).

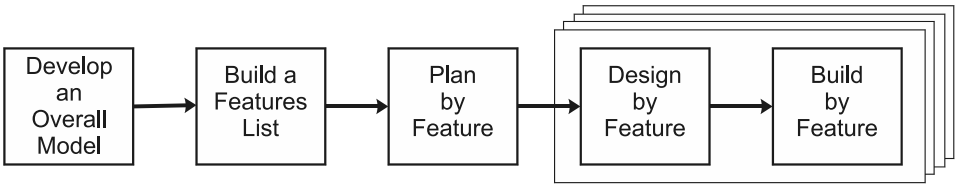


Abbildung 4.3: Aktivitäten beim *Feature-Driven Development*

Iterative Abarbeitung der Feature-Listen

Die ersten drei Aktivitäten werden einmalig nacheinander abgearbeitet und liefern ein Objekt-Modell der Anwendungsdomäne, eine Liste der zu implementierenden Features und einen Implementierungszeitplan mit Verantwortlichkeiten. Die letzten beiden Aktivitäten werden dann iterativ für jedes einzelne Feature durchgeführt (vgl. Abbildung 4.3).

4.2.4.4 Fortschrittsmessung

Meilensteine bei der Feature-Implementierung

Da der Großteil des Projektaufwandes (laut [CLD99] mehr als 75 %) auf die Bearbeitung der Aktivitäten *Design by Feature* und *Build by Feature* entfällt, kann durch die Ermittlung des Abarbeitungsgrades der Featureliste eine Fortschrittsmessung implementiert werden. Dazu wird in [CLD99] und [PF02] vorgeschlagen, für jedes Feature sechs Meilensteine zu definieren, welche bei der seriellen Abarbeitung der Aktivitäten *Design by Feature* und *Build by Feature* für jedes Feature durchlaufen werden (siehe Tabelle 4.1). Die Meilensteine sind durch sechs Aufgaben definiert, welche während der beiden Aktivitäten ausgeführt werden sollen.

Ein Meilenstein soll jeweils dann als abgeschlossen betrachtet werden, wenn die jeweilige Aufgabe vollständig erledigt ist. Zusätzlich können die sechs Meilensteine auch bzgl. des Aufwandes gewichtet werden. In [CLD99] werden dazu als erste Näherung die in Tabelle 4.1 aufgeführen anteiligen Aufwände vorgeschlagen. Damit wäre z. B. ein Feature zu $1\,\% + 40\,\% + 3\,\% = 44\,\%$ fertig, wenn die drei Aufgaben der Aktivität *Design by Feature* vollständig abgeschlossen sind.

Design by Feature			Build by Feature		
Domain Walkthrough	Design	Design Inspection	Code	Code Inspection	Promote to Build
1 %	40 %	3 %	45 %	10 %	1 %

Tabelle 4.1: Die sechs Meilensteine beim FDD mit beispielhaften anteiligen Aufwänden

In [PF02] wird betont, dass nur diejenigen Meilensteine bei der Messung berücksichtigt werden sollen, welche bereits erreicht sind, d. h. bei denen die zugehörige Aufgabe vollständig abgeschlossen ist. In Arbeit befindliche Aufgaben werden nicht mitgezählt, so dass z. B. ein Feature mit einer nur zur Hälfte vollendeten Design-Aufgabe nach wie vor als nur zu 1 % fertig gilt. Dadurch sollen zu optimistisch geschätzte Fertigstellungsgrade vermieden werden. Der Fehler, den man bei dieser Vorgehensweise bei der Berechnung des Gesamtprojektfortschritts begeht, ist nach [PF02] recht klein, da die einzelnen Features von kleiner Granularität sind und maximal einen Zeitraum von zwei Wochen umfassen. Allerdings müssen im Laufe mehrerer Projekte erst Erfahrungswerte bzgl. der anteiligen Gewichtung der Aufwände für die einzelnen Meilensteine gewonnen werden, da die in Tabelle 4.1 genannten Werte nur ein Vorschlag und diese Gewichtungsfaktoren für die einzelnen Meilensteine nach einigen Projekten anzupassen sind.

Vorsichtige Schätzung des Fertigstellungsgrades

Mit der Zerlegung in Meilensteine kann zunächst für jedes Feature aus der Feature-Liste der Fertigstellungsgrad ermittelt werden. Anschließend kann durch Aggregation der Fertigstellungsgrad für die einzelnen Feature-Sets und schließlich für das Gesamtprojekt ermittelt werden. Zusätzlich kann die Anzahl der noch nicht begonnenen, der bereits in Arbeit befindlichen und der fertig gestellten Features ermittelt werden. Diese Feature-Zahlen können wiederum in Relation zu den einzelnen Feature-Sets, und damit zu den zu implementierenden Geschäftsaktivitäten, oder Major-Feature-Sets, welche den einzelnen Sachgebieten der Anwendungsdomäne entsprechen, gestellt werden. In [CLD99] und [PF02] wird dazu eine tabellenartige graphische Darstellung des Fertigstellungsgrades von Feature-Sets vorgeschlagen (vgl. Abbildung 4.4).

Graphische Darstellung des Fertigstellungsgrades

4.2.4.5 Anknüpfungsmöglichkeiten für Softwaremaße

Die erleichterte Fortschrittsmessung war eine Motivation für die Entwicklung des Feature-Driven Development (vgl. [CLD99]). Basierend auf den Features, den Feature-Sets und evtl. den Major-Feature-Sets wurde eine Möglichkeit dargestellt, den aktuellen Projektfortschritt einzuschätzen und diesen Projektstatus auch dem Kunden zu vermitteln und zu begründen. Dazu müssen die Features und Informationen über den Abarbeitungsgrad der einzelnen Features von den Projektbeteiligten gepflegt werden.

Darüber hinaus sind Features auch Kontextinformationen und kommen somit als Anknüpfungspunkte für Softwaremaße in Betracht, da Aktivitäten wie der Domain-Walkthrough oder Design-Tätigkeiten im Kontext eines Features durchgeführt wer-

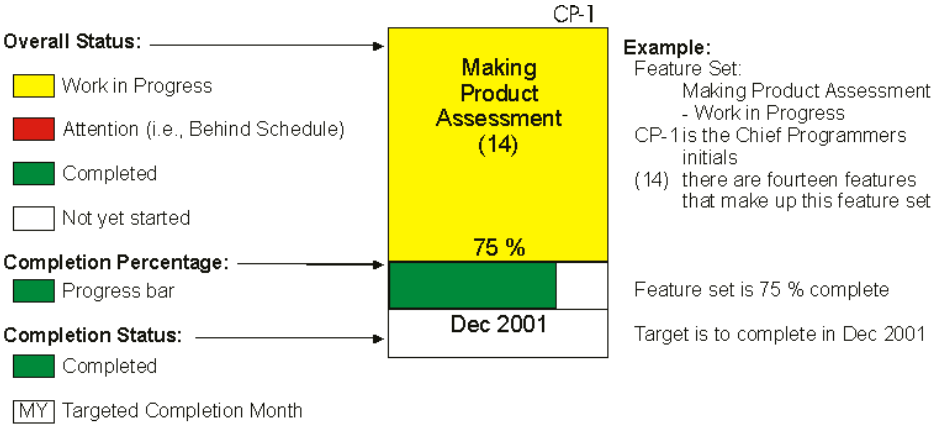


Abbildung 4.4: Graphische Darstellung des Fertigstellungsgrades eines Feature-Sets (vgl. [PF02], S. 85)

Features sind projektspezifisch

den. Allerdings eignen sich die Features weniger gut für die Erhebung projektübergreifender, vergleichender Softwaremaße, da Features projektspezifisch sind.

Features können bezüglich ihrer Eigenschaft als Projekt-Strukturierungselemente mit den Arbeitspaketen eines Projektstrukturplans (vgl. Abschnitt 4.2.1.1) verglichen werden. Ähnlich wie die Arbeitspakete sind die Features von einer bestimmten Projektinstanz abhängig. Aber im Gegensatz zu den Teilaufgaben, welche als Vaterelemente der Arbeitspakete bei einem funktionsorientierten PSP projektübergreifend gültig sein können (Standard-PSP), sind beim FDD auch die Feature-Sets und Major-Feature-Sets vom jeweiligen Projekt abhängig, da diese sich an den betrieblichen Funktionalitätsanforderungen der zu erstellenden Software orientieren.

Phasenbezogene Aufwandsermittlung

Die dynamische oder zeitliche Projektstruktur beim FDD ist, wie in Abbildung 4.3 dargestellt, durch die Abfolge der Phasen

- Develop an Overall Model,
- Build a Features List,
- Plan by Feature und der
- Feature-Implementierungsphase

gegeben. An diesen Phasen lassen sich zumindest Arbeitszeitmaße anknüpfen, welche evtl. noch durch Division mit der Gesamt-Feature-Anzahl normiert werden können. Dadurch ist es beispielsweise möglich, die Aufwände für die einzelnen Phasen projektübergreifend zu vergleichen oder über die Aufwände für die ersten Phasen mittels Hochrechnung den Gesamtprojektaufwand abzuschätzen.

Aktivitätsbezogene Aufwandsermittlung

Als statische Strukturelemente können beim FDD die Aktivitäten der ersten drei Phasen und zusätzlich die in Tabelle 4.1 dargestellten und pro Feature durchzuführenden Aufgaben betrachtet werden. Auch diese sind grundsätzlich als Anknüpfungspunkte für Softwaremaße geeignet. Hier dürfte jedoch weniger im Vordergrund stehen, einzelne Aktivitäten bzw. Aufgaben wie beispielsweise die Designaufwände zwischen zwei Projekten vergleichen zu wollen. Stattdessen kann mittels einer an die-

sen Implementierungs-Aufgaben angeknüpften Aufwandsermittlung die tatsächliche Aufwandsverteilung (vgl. Tabelle 4.1) bei der Feature-Implementierung für die jeweilige Organisation ermittelt werden, welche wiederum bei der Fortschrittsmessung innerhalb zukünftiger Projekte von Bedeutung ist.

4.3 Praktische Anwendung der Anknüpfungspunkte

Im letzten Abschnitt wurden vier Typen von Anknüpfungspunkten für Softwaremaße vorgestellt:

- Aktivitäten
- Artefakt-Funktionalitäten
- Projektphasen
- FDD-Features

Anknüpfungspunkte auf Prozess- bzw. Instanz-Ebene

Die ersten drei Typen von Anknüpfungspunkten unterscheiden sich von den FDD-Features dahingehend, dass sie bereits auf Prozessebene existieren. Softwaremaße, welche sich auf die genannten Kontextinformationen beziehen, können unabhängig von einem konkreten Projekt definiert und damit projektübergreifend verwendet werden.

Dies trifft für die FDD-Features nicht zu, da Features konzeptionsbedingt erst zur Laufzeit eines Projektes festgelegt werden können. Der große Vorteil des Feature-Konzepts liegt jedoch darin, dass durch die Überwachung der bereits implementierten Features eine effektive Messung des Projektfortschrittes erfolgen kann. Mit den ersten drei Typen von Anknüpfungspunkten ist dagegen eine Fortschrittsmessung nicht möglich, da diese konkrete und projektspezifische Kontextinformationen benötigt.

Aktivitäts-orientierte Aufwandsermittlung

Die Aktivitätsanknüpfungspunkte ermöglichen insbesondere eine Berechnung der Aufwände, welche für einzelne Aktivitätsarten eines Softwareentwicklungsprojektes erbracht werden. Dies erleichtert die Beurteilung der Wirksamkeit von Prozessverbesserungsmaßnahmen dahingehend, dass geprüft werden kann, ob sich die (normierten) Aufwände für bestimmte Tätigkeiten tatsächlich ändern, nachdem Modifikationen im Prozess (z. B. die Einführung von Peer-Reviews) durchgeführt wurden.

Funktionalitäts-orientierte Umfangs- und Aufwandsermittlung

Die Verwendung von Artefakt-Funktionalitäten als Anknüpfungspunkte ermöglicht zunächst eine Beurteilung der Umfänge einzelner Software-Komponenten des zu entwickelnden Systems und der dafür erbrachten Aufwände. Dadurch erhält eine Organisation die Möglichkeit, langfristig eine Erfahrungsdatenbank aufzubauen (vgl. [BCR94]), um damit die Aufwände künftiger Projekte besser abschätzen zu können (siehe auch Abschnitt 6.2.2).

Die Berechnung von Softwaremaßen in Bezug auf einzelne Phasen des Projekts ermöglicht eine Hochrechnung von den in den bereits abgeschlossenen Projektphasen erbrachten Aufwänden auf die für das Gesamtprojekt noch zu erbringenden Aufwände.

In Abhängigkeit von den Zielen der Softwaremessung wird man daher eine Kombination der verschiedenen Anknüpfungspunkte einsetzen.

4.3.1 Agile Softwareentwicklung

Im Bereich der *Agilen Softwareentwicklung* gibt es weder hierarchisch strukturierte Aktivitätsbeschreibungen noch sind bestimmte Workflows von vornherein festgelegt (vgl. [Bec04], [Coc01]). Die verschiedenen Vorgehensweisen der Agilen Softwareentwicklung werden stattdessen in Bezug auf anzuwendende *Techniken zur Softwareentwicklung* und den zugrunde liegenden *Prinzipien* beschrieben (vgl. [BT05]).

Agile Aktivitäten

Dennoch lässt sich auch für agile Softwareentwicklungsprojekte ein Standard-Projektstrukturplan gemäß Abschnitt 4.2.1.1 angeben (vgl. [DHW06b]). Dieser besteht nur aus einer Ebene und enthält die Aktivitäten

- Analyse,
- Modellierung,
- Implementierung,
- Test,
- Inbetriebnahme (*Deployment*),
- Konfigurations-Management und
- Projektmanagement,

die normalerweise innerhalb jedes Softwareentwicklungsprojektes durchgeführt werden müssen. Der Einfachheit halber werden die genannten Tätigkeiten an dieser Stelle als *Aktivitäten* bezeichnet, auch wenn diese im Rahmen einzelner Vorgehensmodelle auch als *Aktivitätsgruppen* oder *Disziplinen* bezeichnet werden.

Artefakt-Funktionalitäten

Als zweiter Anknüpfungstyp können die im Abschnitt 4.2.2 dargestellten Artefakt-Funktionalitäten verwendet werden. Somit kann die zweite Dimension der Kontextinformationen durch die Funktionsbeschreibungen

- Anwendungslogik,
- Präsentationsschicht,
- Datenhaltung,
- Transaktionsverwaltung und
- Benutzer- und Berechtigungsverwaltung

definiert werden.

Damit können die zu messenden Entitäten des Softwareentwicklungsprozesses mit zweidimensionalen Kontextinformationen gekennzeichnet werden, wie sie in Tabelle 4.2 als 2-Tupel aufgelistet sind. Um im Rahmen der agilen Softwareentwicklung keinen als unnötig groß empfundenen Mehraufwand zu fordern, kann man sich darauf beschränken, mit diesen Kontextinformationen nur

- die Commit-Informationen im Versionsverwaltungssystem und
- die Zeitbuchungen im Zeiterfassungssystem

zu kennzeichnen. Damit kann nachträglich ermittelt werden, bei welcher Aktivität einzelne Artefakte entstanden sind oder verändert wurden oder für welche Komponenten und während welcher Tätigkeiten die einzelnen Zeitaufwände erbracht wurden (vgl. [DHW06b]).

	Analysis	Modeling	Implementation	Test	Deployment	Config. Mgmt.	Project Mgmt.
Application Logic	$(1,1)$	$(2,1)$	$(3,1)$	$(4,1)$	$(5,1)$	$(6,*)$	$(7,*)$
Presentation	$(1,2)$	$(2,2)$	$(3,2)$	$(4,2)$	$(5,2)$	$(6,*)$	$(7,*)$
Data Management	$(1,3)$	$(2,3)$	$(3,3)$	$(4,3)$	$(5,3)$	$(6,*)$	$(7,*)$
Transaction Mgmt.	$(1,4)$	$(2,4)$	$(3,4)$	$(4,4)$	$(5,4)$	$(6,*)$	$(7,*)$
AuthN/AuthZ	$(1,5)$	$(2,5)$	$(3,5)$	$(4,5)$	$(5,5)$	$(6,*)$	$(7,*)$

Tabelle 4.2: 2-Tupel als Kontextinformationen einer generischen Projektstruktur

Diese Vorgehensweise ist jedoch nicht auf die „klassischen“ agilen Softwareentwicklungsprozesse wie das *eXtreme Programming* [Bec04] beschränkt. Auch auf den RUP, der auch „agil“ durchgeführt werden kann (vgl. [HRS04]), kann dieses Verfahren angewendet werden. Die genannten Aktivitäten entsprechen inhaltlich den Disziplinen des RUP. Bei diesen können jedoch zusätzlich die Disziplinen (Core-Workflows) feiner in Workflow-Details gegliedert werden (siehe Abschnitt 4.2.1.3, S. 95).

Je nach Arbeitsweise und Art der Aufgabenzuordnung an die Projektbeteiligten innerhalb der betrachteten Organisation können somit Softwaremaße an den eher grobgranularen Disziplinen oder an den feingranulareren Workflow-Details angeknüpft werden. Im letzten Fall können die so gewonnenen Softwaremaße immer noch auf die Ebene der Disziplinen aggregiert werden, da jedes Workflow-Detail genau einer Disziplin zugeordnet ist.

4.3.2 Multi-Dimensionale Kontextinformationen

Letztendlich ermöglicht dieser Ansatz, die zu messenden Entitäten mit mehrdimensionalen Kontextinformationen zu versehen. Dabei muss innerhalb der Organisation festgelegt werden, welche Kontextinformationen von den Projektbeteiligten jeweils zu pflegen sind.

Beispiel Zeiterfassung

Die dabei zur Verfügung stehenden Möglichkeiten sollen am Beispiel der Zeiterfassungsdatensätze dargestellt werden.

Wie im vorhergehenden Abschnitt gezeigt wurde, bietet die Ergänzung von Zeiterfassungsdaten mit den Kontextinformationen *Aktivität* und *Artefakt-Funktionalität* eine brauchbare Grundlage für spätere Auswertungen.

Die jeweilige Projektphase braucht bei den einzelnen Zeiterfassungsvorgängen nicht mit angegeben werden, da diese aus dem Datum der Zeiterfassung ermittelt werden kann. Damit ist zusätzlich eine Auswertung der Aufwände nach Projektphasen möglich.

Falls die Methode des Feature-Driven Developments eingesetzt wird, kann entweder an Stelle der Artefakt-Funktionen oder aber auch zusätzlich zu diesen das jeweilige Feature und dessen Zustand mit angegeben werden. Dadurch ist auch eine automatische Ermittlung des Arbeitsfortschrittes bei den einzelnen Features und bei den aggregierten Feature-Sets möglich. Diese Erfassung ist nicht notwendigerweise mit einem Mehraufwand verbunden. So wird durch die Angabe der Informationen

- Aktivität: *Design Inspection*
- Feature: *Kontostandsabfrage*
- Zeitaufwand: *2,5 Stunden*

bei der Zeiterfassung implizit auch die Information zur Verfügung gestellt, dass das Feature *Kontostandsabfrage* den Design-Meilenstein bereits erreicht hat und damit zu 41 % fertiggestellt ist (vgl. Tabelle 4.1, S. 103).

Artefakt-Funktionalität vs. Feature

Für dieses Beispiel würde die Angabe der Artefakt-Funktionalität keinen Sinn ergeben, da bei der Designinspektion eines Features das gesamte Design für dieses Feature, von der Datenhaltung bis zur Darstellung, betrachtet wird. Falls es sich jedoch um die Aktivität *Coding* handelt, könnte zusätzlich als vierte Dimension der Kontextinformationen angegeben werden, dass in den angegebenen zweieinhalb Stunden an der graphischen Benutzeroberfläche der *Kontostandsabfrage* gearbeitet wurde.

Auch die tatsächliche Fertigstellung eines Features könnte man auf diese Weise mittels der Zeiterfassung signalisieren. Dazu müsste man eine Zeitdauer von Null auf die Aktivität *Feature-Abschluss* für das entsprechende Feature in der Zeiterfassung buchen.

Eine analoge Betrachtung ist bezüglich der Auswahl derjenigen Kontextinformationen möglich, um die die Commit-Informationen im Versionsverwaltungssystem ergänzt werden sollen.

Insgesamt stellt dieser Ansatz somit die unterschiedlichsten Möglichkeiten der Auswertung der Prozessaktivitäten zur Verfügung, je nachdem für welche Auswahl an Kontextinformationen und deren verbindlichen Pflege sich die jeweilige Organisation entschieden hat.

Beispielhaft seien die folgenden Softwaremaße und die dazugehörigen Erkenntnisse genannt, welche sich unter Verwendung geeigneter Kontextinformationen gewinnen lassen:

Umfang von Prototypen: Wie bereits im Abschnitt 4.1 beschrieben, ist insbesondere im Rahmen der iterativen Softwareentwicklung interessant, welcher Anteil der fertigen Anwendung bereits in der Prototyp-Phase entstanden ist und welcher Programmumfang später hinzugefügt wurde.

Analyse- vs. Implementierungs-Tätigkeiten

Dies lässt sich messen, wenn die Commit-Informationen im Versionsverwaltungssystem einen Verweis auf die Workflow-Details (*Architectural Synthesis* bzw. *Implement Components*) oder die Disziplinen (*Analysis & Design* bzw. *Implementation*) enthalten, innerhalb derer an den betroffenen Artefakten gearbeitet wurde, so dass nachträglich der bei den jeweiligen Aktivitäten erzeugte Code-Umfang bestimmt

werden kann.

Ergänzend kann auch der zugehörige Arbeitsaufwand bestimmt werden, wenn die Datensätze im Zeiterfassungssystem mit Angaben über die genannten *Workflow-Details* als Kontextinformationen versehen werden.

Requirements Creep: Werden im Laufe des Projektes durch den Auftraggeber immer wieder neue oder veränderte Anforderungen an die Funktionalität bzw. das Verhalten der zu entwickelnden Software gestellt, so wird dies als *Requirements Creep* (Schleichender Anforderungszuwachs) bezeichnet. Bei iterativ und insbesondere bei agil durchgeführten Projekten werden Änderungen an den Anforderungen als normal empfunden. Es ist ein Wesensmerkmal der agilen Softwareentwicklung, dem Auftraggeber durch kurze Implementierungszyklen eine Möglichkeit zu geben, sich über seine Anforderungen im Klaren zu werden und diese auch entsprechend zu modifizieren (vgl. [Bec04]). Auch der RUP geht von vornherein davon aus, dass sich die Anforderungen im Laufe des Projektes ändern werden (vgl. [Kru04], S. 55 ff. und Abbildung A.5, S. 180). Dennoch wird der Umfang dieser Änderungswünsche in Abhängigkeit vom konkreten Projekt und vom jeweiligen Kunden sehr unterschiedlich ausfallen. Um die Aufwände für die Erarbeitung der Anforderungen im Projektverlauf untersuchen zu können, müssen die Zeiterfassungsdatensätze die Kontextinformation *Disziplin* enthalten. Damit kann die Verteilung der Aufwände für die Anforderungsanalyse auf die einzelnen Phasen untersucht werden, was Rückschlüsse auf die Veränderungen der Anforderungen im Laufe des Projekts zulässt.

Verteilung der Testaufwände: Im Gegensatz zu den Aufwänden für die Anforderungsanalyse, welche, obwohl sie während des gesamten Projektverlaufs auftreten, ihren Schwerpunkt in den ersten beiden Projektphasen haben sollten, wird von den Testaufwänden erwartet, dass sie in allen Projektphasen mehr oder weniger gleichmäßig auftreten (vgl. Abbildung A.5, S. 180). Auch dies kann verifiziert werden, wenn die Zeiterfassungsdaten analog zur Untersuchung der Aufwände für die Anforderungsanalyse die Kontextinformationen *Disziplin* enthalten.

4.3.3 Projektfortschritt und Vertragsgestaltung

Projektumfang bei Vertragsabschluss unsicher

Insbesondere bei der Beschreibung des RUP und der Agilen Softwareentwicklung wurde betont, dass bei diesen Vorgehensmodellen die Anforderungen an das zu entwickelnde System iterativ weiterentwickelt oder sogar im Laufe des Projektes überhaupt erst definiert werden. Daher kann zu Projektbeginn kein entsprechendes Verfahren zur Abschätzung des Projektumfangs und der Projektdauer angewendet werden. Die meisten Kunden, die ein IT-Projekt in Auftrag geben, werden jedoch Aussagen über den Projektumfang und die damit verbundenen Projektkosten als Vertragsbestandteil fordern (vgl. [DH03]).

Optional Scope Contracts

Beck und Cleal [BC99] beschreiben in diesem Zusammenhang sogenannte *Optional Scope Contracts* (Verträge mit wahlfreiem Umfang) als mögliche Alternative. Die

Grundidee dabei ist, dass vertraglich nur die Leistungserbringer, der Tätigkeitszeitraum und das Honorar festgelegt werden. Der genaue Umfang der zu erbringenden Leistung wird nicht definiert. Allerdings wird von Seiten des Dienstleisters versprochen, nach seinem Vermögen die bestmögliche Leistung zu erbringen.

Iteratives Festpreismodell

Eine derartige Vorgehensweise wird von Müller in [Mül03] als *Iteratives Festpreismodell* bezeichnet, da die Beteiligten jeweils für die nächste Stufe des Projektablaufes einen Festpreis vereinbaren und für die weiteren Schritte Vorhersagen abgeben. Jede vergangene Iteration fungiert dabei als Basis für den Abgleich von Erwartungen und geleisteter Arbeit und beeinflusst die zukünftige Kalkulation.

Staged Contracts

Lott schränkt in [Lot97] die Ausgestaltungsmöglichkeiten derartiger *Staged Contracts* (Phasenorientierte Verträge) dahingehend ein, dass sich der jeweilige Vertragsumfang an signifikanten Meilensteinen des Entwicklungsprozesses orientieren soll. So könnte ein erster Vertrag zur Projektfindung abgeschlossen werden, in der eine Ist-Analyse durchgeführt und die Anforderungen des Kunden ermittelt werden. Ein zweiter Vertrag würde dann den Analyse- und Design-Aktivitäten gewidmet sein, während ein dritter Vertrag die Konstruktion der Software und deren Inbetriebnahme beschreibt.

Der Kerngedanke dieser Vertragsausgestaltungen ist, dass der Dienstleister zu Vertragsbeginn nicht gezwungen ist, Angaben über Umfang und Dauer des Projektes zu machen, für deren Einschätzung er zu diesem Zeitpunkt noch nicht die notwendigen Informationen hat. Zusätzlich behalten beide Vertragspartner sich das Recht vor, nach dem aktuellen Vertrag keinen Folgevertrag abzuschließen, weil z. B. der Auftraggeber mit der erbrachten Leistung nicht zufrieden war oder aber der Dienstleister evtl. festgestellt hat, dass eine Fortführung des Projektes seine personellen Möglichkeiten übersteigt.

Allerdings muss vertraglich sichergestellt sein, dass der Auftraggeber nach dem Abschluss einer Phase ein Produkt erhält, welches zwar noch nicht perfekt und auch noch nicht fertig sein muss, aber als Grundlage für eine Weiterentwicklung dienen kann.

Phasenorientierte Softwaremessung

An dieser Stelle kann der in diesem Kapitel beschriebene Ansatz zur Softwaremessung zum Tragen kommen. Mit der Anknüpfung von Softwaremaßen an den Kontextinformationen des Entwicklungsprojektes ist es möglich, die Kernattribute Aufwand und Zeitdauer im Kontext einzelner Aktivitäten und Phasen zu bestimmen. Diese Informationen können dann zum Abschätzen des noch verbleibenden Aufwandes im aktuellen Projekt bzw. des Umfangs zukünftiger Projekte verwendet werden.

4.4 Ein Qualitätsmodell für die Prozessintegration der Softwaremessung

4.4.1 Der Live-Ansatz

In [Riz06] wird der *Live-Ansatz* als ein Modell dargestellt, welches beschreiben soll, wie innerhalb von Entwicklungsumgebungen Projektinformationen verwaltet und verfügbar gemacht werden. Dieses Modell enthält, ähnlich dem CMMI Reifegradmodell (vgl. Abschnitt 2.1.4.1), ein Charakterisierungsschema, mit dem der Grad der Auswertbarkeit von Projektinformationen und die dafür vorhandene Werkzeugunterstützung beurteilt werden können.

Der Grund für die Entwicklung des Live-Ansatzes war laut [Riz06], dass klassische Entwicklungsumgebungen die Anforderungen agiler Softwareentwicklungstechniken nicht hinreichend unterstützen würden. Insbesondere wären viele Entwicklungsumgebungen jeweils nur auf ein bestimmtes Vorgehensmodell zugeschnitten. Bei Entwicklungsumgebungen, welche aus mehreren Einzelwerkzeugen bestehen (sogenannte Application Lifecycle Management Suiten), würden verschiedene Rollen mit unterschiedlichen Werkzeugen und Methoden unterstützt werden. Dies verhindere zum einen den flexiblen Einsatz von Teammitarbeitern. Zum anderen bestehe dabei die Gefahr, dass sich durch die einzelnen Werkzeuge sogenannte Insellösungen ausbilden, innerhalb derer Informationen separat vorgehalten werden, so dass sie nur schwer im Rahmen des Projekt-Controllings ausgewertet werden können.

Einschränkungen vieler klassischer Entwicklungsumgebungen

Der Live-Ansatz besteht aus einem Satz von Richtlinien, welche Anforderungen an die Auswertbarkeit von Projektinformationen definieren, und einem Charakterisierungsschema, welches den Erfüllungsgrad dieser Richtlinien innerhalb einer Organisation beschreibt (vgl. [Riz06]).

4.4.1.1 Die sechs Live-Richtlinien

1. *Einziger gemeinsamer Vorfahre*
   Die erste Richtlinie besagt, dass alle Artefakte, Aktivitäten und Informationen, welche innerhalb des Softwarelebenszyklus auftreten, Instanzen der Oberkategorie *Work-Item* sind. Dazu gehören beispielsweise neben den Anforderungen, dem Quellcode und den Testplänen auch Fehlermeldungen, Aufgaben und Änderungsanträge.

   Work-Item

2. *Single Source*
   Von allen Work-Item-Instanzen gibt es jeweils nur ein einziges Exemplar und keine Kopien. Dadurch ist jede Projektinformation innerhalb des Projektes auch nur einmal vorhanden. Beispielsweise darf ein Testplan niemals um eine Kopie der Anforderungen aus der Anforderungsspezifikation ergänzt werden, sondern er darf nur (falls technisch möglich) mit der Original-Anforderungsspezifikation verknüpft sein.

3. *Single Repository*
   Aus logischer Sicht darf nur ein einziges Repository für die Versionierung und Verwaltung aller Work-Items vorhanden sein. Falls aus technischen Gründen mehrere Repository-Installationen nötig sein sollten (Lastverteilung, verteilte Entwicklungsarbeit), so müssen diese für den Anwender zu einer Benutzerperspektive integriert werden.

4. *Anpassbarkeit von Work-Items, Kategorien und Ausprägungen*
   Die vierte Richtlinie verlangt, dass Anwender eigene Ausprägungen von Work-Item-Kategorien definieren können, um spezielle Unternehmens- oder Projektanforderungen umzusetzen. Beispiele hierfür sind

   - ein „Änderungsantrag“,
   - ein „Testdatenbestand“, aber auch
   - ein „Kunde“.

   Letztendlich können somit alle Entitäten, welche innerhalb der Organisation oder eines Projektes von Bedeutung sind, auf Work-Items abgebildet werden.

5. *Live-Features*
   Da mit dem Live-Ansatz u.a. die Funktionalität von Entwicklungswerkzeugen beurteilt werden soll, werden in dieser Richtlinie sogenannte *Features* betrachtet, welche einzelne Operationen von Entwicklungsumgebungen beschreiben und sich damit von dem Feature-Begriff des *Feature-Driven Developments* (vgl. Abschnitt 4.2.4) unterscheiden. Ein Feature wird dabei als *live* bezeichnet, wenn sich eine Operation auf eine beliebige Instanz eines Work-Items anwenden lässt. Als Beispiel hierfür wird die Operation „*Show Progress*“ genannt, welche gewöhnlich auf eine Aufgabe angewendet wird, um deren Abarbeitungsgrad darzustellen. Wenn diese Operation als *Live-Feature* implementiert ist, so ist sie auf jedes beliebige Work-Item anwendbar. Somit wäre es möglich mit dieser Operation auch den Abarbeitungsgrad eines Testplans, einer Anforderungsspezifikation oder eines Änderungsantrages zu bestimmen.

Features als Funktionalitäten von Entwicklungsumgebungen

6. *Darstellung*
   Die Darstellung der Live-Features muss für unterschiedliche Benutzerrollen bzgl. des Inhalts und des Formats anpassbar sein. Beispielsweise sollen Informationen für die Managementsicht in aggregierter Form dargestellt werden, während für Entwickler auch eine Detailsicht auf Aufgabenebene vorhanden ist.

4.4.1.2 Das Live-Charakterisierungsschema

Neben den sechs Richtlinien enthält das Live-Modell ein Charakterisierungsschema, welches dazu dient, den Erfüllungsgrad der Live-Richtlinien innerhalb einer Softwareentwicklungsumgebung auszuwerten. Das Modell besteht aus fünf Ebenen, welche Kriterien bzgl. des Erfüllungsgrades des Live-Ansatzes enthalten. Jede Ebene verlängert dabei die Kriterien der vorhergehenden Ebene.

1. *Foundation-Level*
   Auf der ersten Ebene müssen die Richtlinie 1 (gemeinsamer Vorfahre) und die Richtlinie 2 (*Single Source*) erfüllt sein.

2. *Connection-Level*
   Bei dieser Ebene müssen die Work-Item über *Links* miteinander verbunden sein, wobei zwischen *Containment-Links* und *Impact-Links* unterschieden wird. Beispielsweise kann eine einzelne Anforderung über einen Containment-Link mit der Anforderungsspezifikation und über einen Impact-Link mit dem entsprechenden Unit-Test verbunden sein.

3. *Fusion-Level*
   Auf dieser Ebene wird die Richtlinie 3 (*Single Repository*) unterstützt. Gleichzeitig wird gefordert, dass tatsächlich auch alle Work-Items mittels des Versionsverwaltungssystems versioniert werden.

4. *Control-Level*
   Auf der vierten Ebene sollen die Work-Items mit Informationen zu Kosten, Zeit, Prioritäten, Support und Prüfverfahren versehen werden. Diese Informationen können nach [Riz06] beispielsweise Schätzwerte zur Fertigstellung, geplante Start- und Endtermine, Projektmeilensteine, erwartete und aktuelle Kosten, Nutzen, Prioritäten und Dringlichkeiten enthalten.

5. *Governance-Level*
   Auf der höchsten Ebene schließlich wird die Richtlinie 4 (Anpassbarkeit von Work-Items, Kategorien und Ausprägungen) unterstützt. Darüber hinaus sollen die Work-Items Risiko-, Ressourcen- und Finanzinformationen enthalten.

Bei dieser Darstellung fällt zunächst auf, dass die Richtlinien 5 und 6, welche die Live-Features betreffen, bei keiner der fünf Ebenen verlangt werden. Stattdessen kennt der Live-Ansatz fünf Live-Features, welche implizit beim Erreichen der jeweils nächsten Live-Ebene verfügbar sind.

Live Search: Ab dem Foundation-Level können Work-Items, und damit Projektinformationen, nach bestimmten Kriterien gesucht werden.

Live Trace: Ab dem Connection-Level unterstützen die Work-Items die rollenbasierte Navigation sowie Traceability- und Impact-Analysen.

Live Track: Da im Fusion-Level die Work-Items in demselben Versionierungssystem abgelegt werden, welches auch vom Änderungs- und Workflow-Management genutzt wird, ist ein entsprechendes Lebenszyklusmanagement gegeben. Beispielsweise können die Auswirkungen von Anforderungsänderungen verfolgt werden.

Live Plan: Ab dem Control-Level sollen Projektplanung und die Darstellung des Projektfortschritts mittels automatisierter Plandarstellung erfolgen. Dabei soll

der Plan automatisch erstellt und bei Änderungen auf Basis der in den Work-Items gespeicherten Informationen (z. B. Priorität, Dringlichkeit und Abhängigkeiten) einem Update unterzogen werden.

Live Dashboard: Im Governance-Level soll schließlich ein Live Dashboard zur Steuerung aller Projektaktivitäten in Echtzeit zur Verfügung stehen.

4.4.2 Beurteilung des Live-Ansatzes

Eine Hauptintention des Live-Ansatzes ist die Darstellung eines Regelwerkes, mit dem Softwareentwicklungsumgebungen hinsichtlich ihrer Eignung für die agile Softwareentwicklung eingeschätzt werden können. Diese Beurteilung stützt sich jedoch im Wesentlichen darauf, wie Informationen innerhalb der Softwareprojekte verwaltet und zugänglich gemacht werden. Andere Kriterien, wie beispielsweise die technischen Funktionalitäten der Entwicklungsumgebungen, werden bei diesem Ansatz gar nicht betrachtet. Stattdessen wird darauf hingewiesen, dass das Werkzeug *Polarion Development Productivity Platform* der Firma *Polarion Software*[5] den Live-Level 5 erfüllen würde (vgl. [Riz06]).

Widersprüche innerhalb der Live-Richtlinien

Auch erscheinen die Kriterien für die Einhaltung der Live-Richtlinien überarbeitungsbedürftig. So wird für den Control-Level verlangt, dass die Work-Items mit Informationen zu Kosten, Zeit und weiteren Attributen versehen werden. Dies würde jedoch teilweise der Single-Source-Richtlinie widersprechen, nach der alle Informationen in der Entwicklungsumgebung nur einmal vorkommen dürfen. Da Zeiterfassungsbelege und Kostenschätzungen ebenfalls als Projektinformationen und somit als Work-Items betrachtet werden müssen, dürfen die Artefakt-Work-Items (z. B. ein Stück Quelltext) nur einen Verweis auf die zugehörigen Controlling-Work-Items haben. In der Praxis ließe sich diese Verbindung beispielsweise dadurch herstellen, dass bei der Zeiterfassung in den jeweiligen Datensätzen das bearbeitete Work-Item bzw. die zugehörige Work-Item-Gruppe mit abgespeichert wird, wie dies auch im Abschnitt 4.1 gefordert wird.

Live-Ansatz als Reifegradmodell für die Softwaremessung

Wenn man jedoch den Live-Ansatz nicht als Beurteilungsregelwerk für Softwareentwicklungswerkzeuge, sondern als Reifegradmodell für die Auswertbarkeit von Projektinformationen betrachtet, so kann damit die Integrationsmöglichkeit der Softwaremessung in eine bestimmte Entwicklungsumgebung beurteilt werden. Insbesondere kann mit dem Live-Ansatz beurteilt werden, wie gut eine Entwicklungsumgebung die Ermittlung und Bereitstellung von auf Managementebene relevanten Softwaremaßen unterstützt.

- Wenn auf dem *Live Foundation-Level* die Richtlinien „Einziger Vorfahre“ und „Single-Source“ erfüllt sind, so ist es zunächst einmal möglich, bei der Zeiterfassung, innerhalb der Versionsverwaltung und beim Issue-Tracking einheitliche und eindeutige Verweise auf die betroffenen Work-Items als Kontextinformationen anzugeben. Somit kann festgestellt werden, welcher Zeitaufwand bei-

[5] `http://www.polarion.com`

spielsweise mit der Implementierung einer bestimmten Komponente verbunden war.

- Durch Links miteinander verbundene Work-Items, wie sie auf dem *Connection-Level* gefordert werden, ermöglichen Traceability- und Impact-Analysen. Zusätzlich könnte dabei beispielsweise aus der Controlling-Perspektive ermittelt werden, welche Mehrkosten durch einen Änderungswunsch entstanden sind.

- Und schließlich ermöglicht es das im *Fusion-Level* geforderte „Single Repository“ zusammen mit der Versionierung aller Work-Items, nachträglich die Entwicklung und insbesondere die Verschiebung von Aufwandsschwerpunkten zu analysieren.

4.5 Zusammenfassung

Zu Beginn dieses Kapitels wurde dargelegt, was im Rahmen dieser Arbeit unter Controlling-gerechter Softwaremessung verstanden werden soll. Hierbei geht es zunächst um die Erhebung der im Abschnitt 2.3 als Kernattribute bezeichneten Messgrößen wie Umfang oder Aufwand. Zusätzlich ist entscheidend, dass diese Maße zusammen mit ihrem Kontext betrachtet werden müssen. Es sind daher Maße wie der Aufwand für eine bestimmte Softwarekomponente oder die Änderung der Fehlerrate nach der Einführung von Peer-Reviews von Interesse.

Kernattribute

Ein wichtiges Ziel dieser Arbeit ist, eine Möglichkeit darzustellen, die in der jeweiligen Organisation als relevant erkannten Softwaremaße konsistent und einheitlich zu erheben. Dies beinhaltet insbesondere, dass die jeweiligen Softwaremaße grundsätzlich für jedes durchzuführende Projekt ermittelt werden sollen, um bestimmte Aspekte der einzelnen Projekte zu vergleichen. Da es möglich sein soll, diese Maße unabhängig von einem konkreten Projekt zu definieren, müssen sie sich auf Messgrößen beziehen, die bereits auf der Ebene des Softwareprozesses existieren.

Konsistente Softwaremessung

Im Abschnitt 4.2 wurden daher Möglichkeiten dargestellt, den Softwareentwicklungsprozess zu gliedern und zu strukturieren und ihn letztendlich auf eine generische Grundstruktur zurückzuführen. Die Elemente dieser Grundstruktur können als Anknüpfungspunkte für die Softwaremaße verwendet werden, wodurch diese projektunabhängig definiert werden können.

Anknüpfungspunkte

Im weiteren Verlauf des Kapitels wurde dann die Verwendung von

- Aktivitäten,
- Artefakt-Funktionalitäten und
- Projektphasen

als Anknüpfungspunkte für Softwaremaße erläutert. Dabei wurde insbesondere auf die Anwendbarkeit dieses Ansatzes im Rahmen verschiedener Vorgehensmodelle wie dem V-Modell XT, dem Rational Unified Process oder dem eXtreme Programming eingegangen.

Als Nachteil der ausschließlichen Betrachtung von bereits auf Prozessebene existierenden Messgrößen wurde die Schwierigkeit dargestellt, auf diese Weise den Pro-

Fortschrittsmessung

jektfortschritt zu bestimmen. Deshalb wurde mit dem Feature-Driven Development ein vielversprechender Ansatz vorgestellt, der es einerseits erlaubt, ein Projekt übersichtlich und in kundenverständlicher Form zu gliedern, und andererseits mittels der Features und Feature-Sets dem Projekt eine Strukur einbeschreibt, welche für die Fortschrittsmessung unmittelbar zugänglich ist.

Es wurde im weiteren Verlauf dargestellt, dass es aus der Sicht des Projekt-Controllings optimal wäre, die Anknüpfung von Softwaremaßen an Elemente der Prozessstruktur mit der Fortschrittsmessung durch Bestimmung des Abarbeitungsgrades der einzelnen Features bzw. Feature-Sets zu kombinieren. Dadurch werden für die Softwaremessung letztendlich multidimensionale Kontextinformationen eingesetzt.

Zum Abschluss des Kapitels wurde der sogenannte Live-Ansatz betrachtet. Dieser ist zwar als Qualitätsmodell für Entwicklungsumgebungen entworfen worden, jedoch wurde im vorhergehenden Abschnitt gezeigt, dass damit auch die Auswertbarkeit von Projektinformationen beurteilt werden kann. Insbesondere enthält der Live-Ansatz ein Reifegradmodell, mit dem die Integrierbarkeit der Softwaremessung in einzelne Entwicklungsumgebungen beurteilt werden kann.

Reifegradmodell für die Softwaremessung

5 Implementierung des Maven Measurement Frameworks

> Maven is a declarative project management tool that decreases overall time to market by effectively leveraging cross-project intelligence.
>
> *(aus [CMP+07])*

5.1 Die Projektverwaltungssoftware Maven

In diesem Kapitel wird eine Möglichkeit dargestellt, die innerhalb dieser Arbeit entwickelte Methode zur Softwaremessung mittels geeigneter Software-Werkzeuge zu realisieren. Die prototypische Implementierung dieses Ansatzes wurde auf ein Messwerkzeug für Java-Projekte beschränkt.

Ein Kerngedanke bei der Umsetzung war, die Implementierung möglichst flexibel zu halten, damit das Software-Messwerkzeug für die jeweiligen Projektbedürfnisse problemlos angepasst werden kann. Daher ist auch keine monolithische Anwendung entstanden, sondern eine Werkzeug-Framework, aus dem der Anwender einzelne Bausteine auswählt, um damit die jeweiligen Messungen durchzuführen.

Dieses Werkzeug-Framework, welches auf Basis der Projektverwaltungssoftware *Maven*[1] entstanden ist, soll im Weiteren als *Maven Measurement Framework* (MMF) bezeichnet werden.

Maven Measurement Framework

Maven selbst wird von einigen seiner Entwickler als *Project Management Framework* bezeichnet (vgl. [CMP+07], S. 22), wobei nach [CMP+07] auch dieser Begriff diese Software nur sehr unzureichend beschreibt.

Grundsätzlich kann Maven zunächst einmal als Weiterentwicklung des Build-Management-Werkzeuges *Ant*[2] betrachtet werden und dient somit zum automatisierten Übersetzen von Java-Quelltexten. Während Ant jedoch im Wesentlichen aus einer Sammlung von Anweisungen besteht, mit denen man den Ablauf des Build-Vorgangs definieren kann, enthält Maven entsprechende Verfahrensmuster, welche den Build-Vorgang standardisieren und die Projektverwaltung mittels wiederverwendbarer und gemeinsam nutzbarer Build-Strategien vereinfachen sollen (vgl. [CMP+07], S. 23 f.). Das Ziel der Maven-Entwickler war somit, einige *Best Practices* bzgl. der Verzeichnisverwaltung und des Build-Prozesses bei Java-Projekten in das Build-Management-Werkzeug zu integrieren.

[1] `http://maven.apache.org`
[2] `http://ant.apache.org`

5.1.1 Ant vs. Maven: Build-Management im Vergleich

5.1.1.1 Build-Management mit Ant

Um die Eigenschaften von Maven besser verdeutlichen zu können, soll an dieser Stelle zunächst die Arbeitsweise von Ant anhand eines typischen Ant-Build-Skripts betrachtet werden (siehe Listing 5.1).

```
1  <project name="dms" default="compile" basedir=".">
2
3    <property name="build.prod.dir" location="build/prod"/>
4    <property name="build.test.dir" location="build/test"/>
5    <property name="src.dir"        location="src"/>
6    <property name="test.dir"       location="test"/>
7    <property name="lib.dir"        location="lib"/>
8
9    <path id="project.classpath">
10     <pathelement location="${build.prod.dir}"/>
11     <pathelement location="${build.test.dir}"/>
12     <fileset dir="${lib.dir}">
13       <include name="*.jar"/>
14     </fileset>
15   </path>
16
17   <target name="prepare">
18     <mkdir dir="${build.prod.dir}"/>
19     <mkdir dir="${build.test.dir}"/>
20   </target>
21
22   <target name="compile" depends="prepare">
23     <javac srcdir="${src.dir}" destdir="${build.prod.dir}">
24       <classpath refid="project.classpath"/>
25     </javac>
26   </target>
27
28   <target name="compile-tests" depends="compile">
29     <javac srcdir="${test.dir}" destdir="${build.test.dir}">
30       <classpath refid="project.classpath"/>
31     </javac>
32   </target>
33
34 </project>
```

Listing 5.1: Ant-Build-Skript (aus [Cla05])

Targets

In einem Ant-Build-Skript werden die Tätigkeiten des Build-Vorganges mittels sogenannter *Targets* definiert. Beispielsweise werden durch das Target `prepare` die Verzeichnisse für den beim Kompilieren erzeugten Bytecode erzeugt. Durch die Targets `compile` und `compile-tests` wird anschließend der eigentliche Übersetzungsvorgang angestoßen. Dabei wird aufgrund der hinterlegten Abhängigkeiten bei Bedarf automatisch das jeweils vorher benötigte Target aufgerufen.

Tasks

Die Targets selbst bestehen aus sogenannten *Tasks*. Dabei handelt es sich um mittels entsprechender Java-Klassen implementierte Kommandos, welche die einzelnen Aufgaben des Build-Prozesses (z. B. Erzeugen von Verzeichnissen, Übersetzen von Sourcecode) durchführen. Die Tasks werden mittels zugehöriger XML-Elemente und -Attribute konfiguriert. Weiter gilt es als guter Stil, sämtliche Pfadangaben in Form von Variablen (sogenannter *Properties*) zu hinterlegen.

Das in Listing 5.1 dargestellte Ant-Build-Skript erwartet also die Source-Dateien für die eigentliche Java-Applikation und die dazugehörigen Unit-Tests in den angegebenen Verzeichnissen, legt bei Bedarf die gewünschten Ziel-Verzeichnisse an und übersetzt die Java-Sourcen getrennt nach Produktiv- und Test-Code. Bei der Übersetzung benötigte Bibliotheken müssen im Verzeichnis `lib` zur Verfügung stehen.

Obiges Ant-Build-Skript kann dabei als „kanonisch“ bezeichnet werden, da jedes Java-Projekt im Prinzip derartig aufgebaut ist, was insbesondere für die Trennung der Produktiv- und Test-Sourcen und der zugehörigen Bytecode-Verzeichnisse gilt. Als Konsequenz muss für jedes Java-Projekt ein sehr ähnliches Ant-Skript erstellt werden.

5.1.1.2 Build-Management mit Maven

Konvention statt Konfiguration

An dieser Stelle setzten nun die Entwickler von Maven durch Praktizierung des Ansatzes „*Konvention statt Konfiguration*“ an und legten als Konvention fest, dass die Verzeichnisstruktur eines mit Maven verwalteten Java-Projekts grundsätzlich wie in Abbildung 5.1 auszusehen hat.

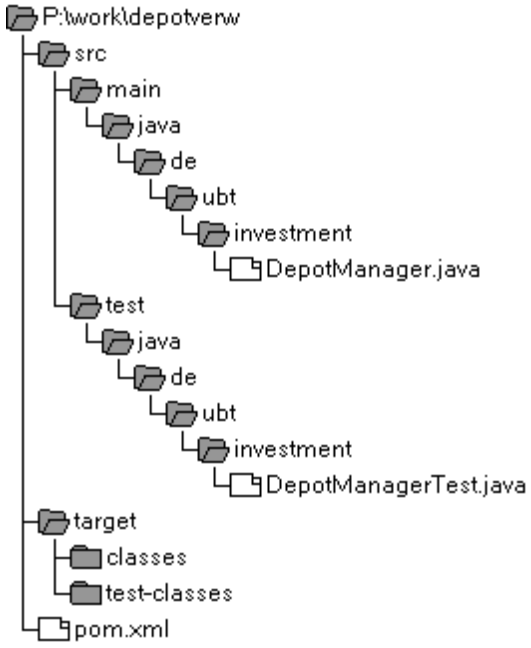


Abbildung 5.1: Verzeichnisstruktur eines Maven-Projekts

Empfohlene Verzeichnisstruktur

Die von Maven erwartete Verzeichnisstruktur enthält zumindest die Ordner `src` und `target`, in denen die Quelldateien verwaltet bzw. die im Rahmen des Build-Prozesses generierten Dateien abgelegt werden. Die genannten Ordner werden jeweils in Bezug auf Produktiv- und Test-Code weiter untergliedert. Die Java-Sourcen für die eigentliche Anwendung werden somit grundsätzlich im Ordner

```
src/main/java
```

verwaltet. Parallel zum Java-Ordner sind noch andere Verzeichnisse denkbar, die z. B. XML-Dateien enthalten, aus denen dann entsprechende Java-Sourcen generiert werden, wie es beispielsweise beim Einsatz objekt-relationaler Mapping-Frameworks üblich ist (vgl. [Bos03]).

Mit dieser nahezu verpflichtend geltenden Verzeichnis-Konvention erübrigen sich alle in Listing 5.1 aufgeführten Pfadangaben, mit Ausnahme der Ablage von Code-Bibliotheken, deren Verwaltung mit Maven später dargestellt werden soll. Zusätzlich erleichtert diese Vorgabe den Entwicklern die Orientierung in neuen Projekten.

5.1.1.3 Mavens Build-Lifecycle

Auch die im Ant-Build-Skript aufgeführten Targets kommen grundsätzlich bei allen Java-Projekten in ähnlicher Weise vor, da der Build-Prozess in der Regel aus den Aktionen

- Vorbereitung,
- Kompilierung,
- Test,
- Konfektionierung des Installationspakets und
- der abschließenden Installation

besteht.

Standard-Build-Lifecycle

Dem trägt Maven in Form eines *Standard-Build-Lifecycles* Rechnung, der aus 21 einzelnen Phasen[3] besteht, welche für jedes Projekt in der gleichen Reihenfolge durchlaufen werden. Abbildung 5.2 zeigt eine Teilmenge dieser Phasen aus dem Standard-Build-Lifecycle.

Maven Plugins

Maven Goals

Diese Phasen können als Erweiterungspunkte für den durch Maven gesteuerten Build-Prozess betrachtet werden. Die eigentlichen Operationen des Build-Prozesses sind in sogenannten *Plugins* implementiert. Dabei handelt es sich um Java-Klassen mit definierten Schnittstellen. Jedes Plugin implementiert dabei ein oder mehrere sogenannte *Goals*. Beispielsweise enthält das *Maven-Compiler-Plugin* die Goals *compile* und *test-compile*, mit denen die Anwendungs-Quelltexte bzw. die Unit-Test-Quelltexte kompiliert werden. Diese Goals werden nun an bestimmte Phasen des Build-Lifecycles gebunden (vgl. Abbildung 5.2). So sind standardmäßig das Goal *compiler:compile*[4] der Compile-Phase und das Goal *compiler:test-compile* der

[3] Stand vom August 2007 (vgl. [CMP+07])

[4] Bei der Beschreibung von Maven-Goals werden per Konvention der Plugin-Name und die Bezeichnung des Goals durch einen Doppelpunkt getrennt notiert.

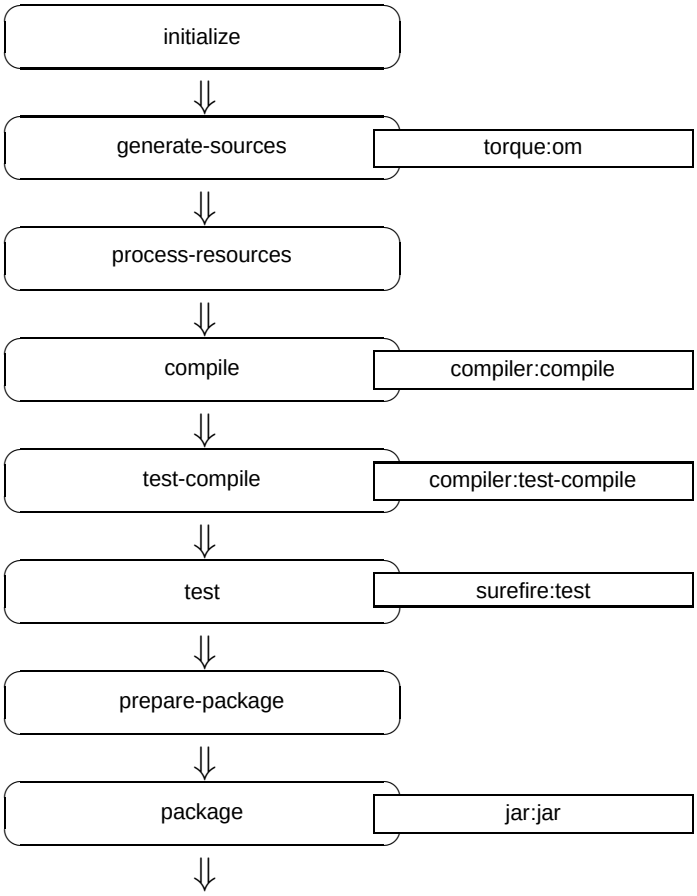


Abbildung 5.2: Teilmenge von Mavens Default Build-Lifecycle

Test-Compile-Phase zugeordnet. Beim Durchlaufen der einzelnen Phasen des Build-Prozesses werden nun die an die Phasen gebundenen Goals aufgerufen. Beim Durchlaufen von Phasen, an denen kein Goal angebunden ist, werden auch keine Aktionen durchgeführt.

Im Vergleich zu dem Ant-Build-Skript auf Seite 118 ist daher für das Kompilieren der Produktiv- und Test-Quellcode-Dateien keinerlei Konfigurationsaufwand erforderlich. Der Speicherort der Quellcode-Dateien ist durch Konvention vorgegeben und der Ablauf des Build-Processes durch den Standard-Build-Lifecylce.

5.1.2 Mavens Project-Object-Model

5.1.2.1 Grundlegende Projekt-Konfiguration

Trotzdem kommt auch Maven nicht ohne Konfiguration aus, jedoch wird versucht, gemäß dem Prinzip „*Konvention statt Konfiguration*“, die Konfigurationsdateien möglichst klein zu halten. Der zentrale Ablageort für Konfigurationsdaten ist eine als

Project Object Model

Project Object Model (POM) bezeichnete XML-Datei. Ein Bespiel einer POM-Datei für eine Depotverwaltungsanwendung ist in den Listings 5.2 und 5.3 dargestellt.

Projektkoordinaten

Das POM eines Maven-Projekts enthält zunächst einmal Informationen über den Namen des Projekts (`<name>`), eine eindeutige Bezeichnung für die Organisation (`<groupId>`) und eine eindeutige Basisbezeichnung für das im Rahmen des Projekts erzeugte Artefakt (`<artifactId>`). Im angegebenen Beispiel wird im Rahmen des Projekts letztendlich ein Jar-Archiv mit der Depotverwaltungsanwendung erzeugt, welches den Dateinamen `depotverw-0.1.jar` erhält.

Deklaration von Abhängigkeiten

Der Ablageort der Quelltextdateien muss, sofern er den Maven-Konventionen entspricht, nicht angegeben werden. Jedoch müssen externe Bibliotheken, welche von dem Projekt verwendet werden und von denen das Projekt somit abhängig ist, im POM deklariert werden. Dies erfolgt im Abschnitt `<dependencies>` der Datei. Bei Verwendung von Maven muss der Entwickler die benötigten Bibliotheken jedoch nicht selbst bereitstellen, sondern sie werden im Rahmen des Build-Vorganges durch Maven selbständig von einem oder mehreren zentralen Repositories heruntergeladen. Im angegebenen Beispiel sind davon die *JUnit*-Bibliothek, der *PostgreSQL*-JDBC-Treiber und das objekt-relationale Mapping-Tool *Torque* betroffen, die jeweils mit der exakten Versionsnummer aufgeführt sind. Somit erklärt sich auch die Bedeutung der *GroupID* für die aktuell zu erstellende Applikation. Die im Beispiel erstellte Depotverwaltungssoftware kann beim Abschluss des Projektes ebenfalls in einem Bibliotheks-Repository veröffentlicht werden. Dabei kann es sich um ein öffentlich verfügbares, zentrales Repository oder auch um ein organisationsinternes Repository handeln, in denen das Jar-Archiv der Depotverwaltungssoftware über die oben beschriebenen Bezeichner eindeutig aufgefunden werden kann.

Bedeutung der GroupID

Beispielhafte Plugin-Konfiguration

Das Listing 5.3 zeigt die Konfiguration der im Rahmen des Projektes verwendeten Maven-Plugins. Das in Abbildung 5.2 dargestellte Compiler-Plugin müsste normalerweise nicht explizit konfiguriert werden, da es standardmäßig an die Compile-Phase des Build-Prozesses gebunden ist. Jedoch können auch, wie in diesem Beispiel, für die Standard-Plugins zusätzliche Informationen, wie beispielsweise die Version des zu erzeugenden Java-Objekt-Codes angegeben werden.

Konfiguration eines O/R-Mapping-Tools

Bedeutsamer ist jedoch die Konfiguration der zusätzlich in den Build-Prozess eingebundenen Maven-Plugins. In diesem Beispiel handelt es sich dabei um das *Torque-Maven-Plugin*[5]. Torque ist ein objekt-relationales Mapping-Tool, welches zur Laufzeit einer Applikation Datensätze einer relationalen Datenbank auf Java-Objekte abbildet (vgl. [Bos03]). Dadurch entfällt in der Java-Applikation die Verwendung von SQL-Statements, welche ansonsten einen Bruch innerhalb des objekt-orientierten Anwendungsdesigns bedeuten würden.

Generierung von Java-Code und SQL-Code

Bei der Verwendung von Torque wird in mehreren XML-Dateien das Relationenschema der Datenbank definiert. Zu diesem Relationenschema werden dann durch die Goals `torque:om` und `torque:sql` sowohl die zugehörigen Java-Klassen als auch die SQL-Anweisungen zum Erzeugen der entsprechenden Datenbanktabellen erzeugt. Letztere werden dann durch das Goal `torque:sqlExec` ausgeführt.

[5] `http://db.apache.org/torque`

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0">
3
4    <modelVersion>4.0.0</modelVersion>
5    <groupId>de.ubt.investment</groupId>
6    <artifactId>depotverw</artifactId>
7    <packaging>jar</packaging>
8    <version>0.1</version>
9    <name>Depotverwaltung</name>
10
11   <dependencies>
12     <dependency>
13       <groupId>junit</groupId>
14       <artifactId>junit</artifactId>
15       <version>3.8.1</version>
16       <scope>test</scope>
17     </dependency>
18     <dependency>
19       <groupId>postgresql</groupId>
20       <artifactId>postgresql</artifactId>
21       <version>8.2-504.jdbc3</version>
22     </dependency>
23     <dependency>
24       <groupId>torque</groupId>
25       <artifactId>torque</artifactId>
26       <version>3.3-RC2</version>
27     </dependency>
28   </dependencies>
29
30   <build>
31     <plugins>
32       <!-- hier erfolgt die Konfiguration der Plugins -->
33     </plugins>
34   </build>
35
36   <reporting>
37     <plugins>
38       <plugin>
39         <groupId>metrics.plugin</groupId>
40         <artifactId>halstead-plugin</artifactId>
41         <version>2.0</version>
42       </plugin>
43     </plugins>
44   </reporting>
45
46 </project>
```

Listing 5.2: POM eines Maven-Projekts (Depotverwaltung)

Konfiguration des Torque-Plugins

Dazu wird zunächst das Torque-Maven-Plugin im Build-Abschnitt des POM deklariert (Listing 5.3, Zeilen 3 ff.) und die genannten Goals des Plugins werden im Abschnitt `<executions>` des POM an die Phase `generate-sources` des Build-Prozesses angebunden (Zeilen 22 ff.). Die Konfiguration des Plugins erfolgt ebenfalls im POM, wobei dem Plugin mitgeteilt wird, welche Datenbank verwendet werden soll (Abschnitt `<configuration>`, Zeilen 7 ff.).

Zusätzlich wird im Abschnitt `<dependencies>` (Zeilen 14 ff.) für Maven die Information hinterlegt, dass zur Verwendung des Torque-Maven-Plugins die Jar-Bibliothek des *PostgreSQL*-JDBC-Treibers notwendig ist. Der Ablageort der XML-Datenbankschema-Dateien bzw. der generierten SQL-Anweisungen muss nicht konfiguriert werden, da es hierfür entsprechende Konventionen (Default-Einstellungen) gibt, welche in der jeweiligen Plugin-Dokumentation nachgelesen werden können.

POM als vollständige und konsistente Projektbeschreibung

Im Idealfall können alle Operationen des Build-Prozesses durch entsprechende Maven-Plugins abgedeckt werden. Dies beinhaltet also nicht nur den eigentlichen Übersetzungsvorgang, sondern auch die Generierung von Sourcecode, die Einrichtung von Datenbanken oder auch das einfache Kopieren oder Umbenennen von Dateien. Liegen diese Tools als Maven-Plugins vor, so erfolgt die Konfiguration dieser Tools und damit die Beschreibung des kompletten Build-Prozesses ausschließlich innerhalb der POM-Datei. Zusätzlich sind auch alle Informationen über evtl. Abhängigkeiten der verwendeten Tools (z. B. benötigte Datenbanktreiber) im POM enthalten. Diese Abhängigkeiten können von Maven durch Herunterladen der benötigten Bibliotheken aus den entsprechenden Repositories automatisiert aufgelöst werden. Damit wird das Projekt vollständig und konsistent durch das Project-Object-Model beschrieben.

5.1.2.2 Weitere Projektinformationen

Neben den bereits aufgeführten Inhalten kann die POM-Datei eines Maven-Projekts noch viele weitere Informationen enthalten. In Abbildung 5.3 (Seite 126) sind die einzelnen Konfigurationsblöcke der POM-XML-Datei dargestellt, wobei diejenigen Blöcke- und Unterblöcke, die zur Verwaltung von Projektinformationen dienen, entsprechend detaillierter dargestellt sind, als Blöcke, in denen hauptsächlich der Build-Prozess selbst konfiguriert wird.

Repository-Koordinaten

Innerhalb des POM-Blocks *POM Relationships* wird zum einen das Projekt selbst mit eindeutigen Bezeichnern versehen (siehe Abschnitt 5.1.2.1, Seite 121). Zusätzlich werden in diesem Abschnitt die Beziehungen zu anderen Projekten deklariert. Der Teilblock *Coordinates* gibt dabei an, unter welchen „Koordinaten“ sich die resultierende Anwendungsbibliothek in einem Maven-Repository wieder auffinden lassen wird. Der Teilblock *Dependencies* beschreibt, wie weiter oben dargestellt, Abhängigkeiten zu anderen Projekten, welche von Maven automatisch aufgelöst werden können. Die Abschnitte *Multi-Module* und *Inheritance* repräsentieren die Möglichkeiten, mit Maven Haupt- und Teilprojekte zu verwalten bzw. Projekteigenschaften zu vererben. Im ersten Fall können durch einen einzigen Maven-Aufruf die Build-Prozesse für alle Teilprojekte angestoßen werden. Im zweiten Fall werden die Build-

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.db.torque</groupId>
      <artifactId>torque-maven-plugin</artifactId>
      <version>3.3-RC2</version>
      <configuration>
        <targetDatabase>postgresql</targetDatabase>
        <driver>org.postgresql.Driver</driver>
        <url>jdbc:postgresql://localhost/depotverw</url>
        <user>investor</user>
        <password>investor</password>
      </configuration>
      <dependencies>
        <dependency>
          <groupId>postgresql</groupId>
          <artifactId>postgresql</artifactId>
          <version>8.2-504.jdbc3</version>
          <type>jar</type>
        </dependency>
      </dependencies>
      <executions>
        <execution>
          <phase>generate-sources</phase>
          <id>Generate Database Stuff</id>
          <goals>
            <goal>om</goal>
            <goal>sql</goal>
            <goal>sqlExec</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
     </plugin>
  </plugins>
</build>
```

Listing 5.3: Maven-Plugin-Konfiguration

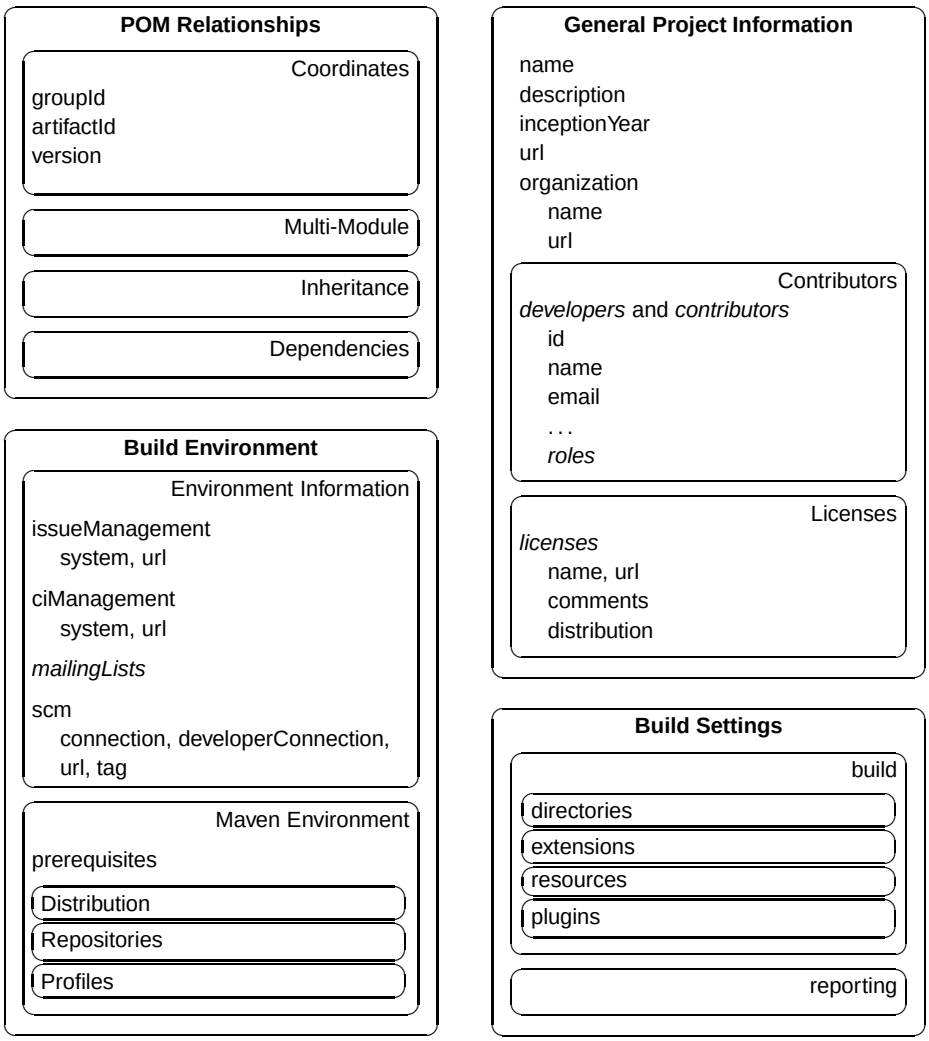


Abbildung 5.3: Konfigurationsblöcke des Maven Project-Object-Models (in Anlehnung an [Son07])

Prozesse der einzelnen Projekte zwar separat angestoßen, jedoch gibt es gemeinsame Projekteigenschaften, die im Vater-Projekt gepflegt werden.

General Project Information

Im POM-Block *General Project Information* wird im Wesentlichen das Projekt aus organisatorischer Sicht beschrieben. Neben einer detaillierten Angabe der Organisation, innerhalb derer das Projekt bearbeitet wird, können hier alle beteiligten Entwickler und sonstige Projektbeteiligte (z. B. Übersetzer, Graphiker) aufgeführt werden. Im Lizenz-Abschnitt kann die Lizenz, unter der das zu entwickelnde Produkt verwendet werden kann, aufgeführt werden. Insbesondere kann hier hinterlegt werden, ob es sich um ein kommerzielles Produkt oder um freie Software handelt.

Build Setting

Einige der *Build-Settings* in der POM-Datei wurden im Rahmen des obigen Beispiels bereits erläutert. In diesem Abschnitt werden hauptsächlich zusätzlich benötigte Maven-Plugins konfiguriert. Auch können hier beispielsweise die Maven-Konventionen für die Verwendung bestimmter Verzeichnisse überschrieben bzw. ergänzt werden.

Informationen zur Entwicklungsumgebung

Build Environment

Für die Integration von Softwaremessung ist insbesondere der Abschnitt *Build Environment* des Project-Object-Models von Bedeutung, da hier mehrere Komponenten der Entwicklungsumgebung definiert werden, welche als Quellen für die Softwaremessung dienen.

Zugriff auf Artefakte über SCM-System

Ein diesbezüglich besonders wichtiger Eintrag ist der Verweis auf das *Software-Configuration-Management-System* (SCM). Dadurch ist es über das Maven-SCM-Plugin möglich, auf das innerhalb des Projekts verwendete Versionsverwaltungssystem zuzugreifen und Aktionen wie *Check-out*, *Commit* oder die Abfrage der Commit-Informationen durchzuführen. Der Zugriff erfolgt dabei generisch, d. h. beim Aufruf des betreffenden SCM-Kommandos muss dem Aufrufenden nicht bekannt sein, welches SCM-System in dem konkreten Projekt verwendet wird. Diese Information ist in der POM-Datei hinterlegt, so dass die Maven-Laufzeitumgebung den generischen Aufruf eines SCM-Kommandos automatisch an eine entsprechende abgeleitete Klasse des SCM-Plugins delegieren kann.

Somit kann unter Verwendung des Maven-SCM-Plugins, wie im Abschnitt 4.2 beschrieben, der Umfang von Code-Artefakten, welche bestimmte Funktionalitäten implementieren oder die im Rahmen bestimmter Aktivitäten erstellt wurden, ermittelt werden. Dazu müssen die betroffenen Artefakte zunächst durch Auswertung der Commit-Informationen identifiziert werden, so dass sie anschließend bzgl. des Umfangs vermessen werden können.

Zugriff auf das Issue-Tracking-System

Zu den Informationen über die Entwicklungsumgebung gehört auch die Angabe des verwendeten *Issue-Management-* oder *Tracking*-Systems, d. h. desjenigen Systems, mit dem Fehlermeldungen und Änderungsanforderungen verwaltet werden. Leider gibt es für den Zugriff auf Issue-Management-Systeme keine allgemeinen Standards, welche für Systeme verschiedener Hersteller gelten würden, um bestimmte Arten von Einträgen abfragen und analysieren zu können. Hier verwendet jedes System eigene Arten von Tracking-Records mit unterschiedlichen Attributen und somit unterschiedlichen Möglichkeiten der Abfrage. Bekannte derartige Systeme sind

beispielsweise *Bugzilla*[6], *trac*[7] oder *Rational ClearQuest*[8]. Damit müssen für jedes Tracking-System eigene Konnektoren entwickelt werden, um die Informationen für die gewünschten Softwaremaße abfragen zu können.

Interessante Softwaremaße, welche über das Issue-Tracking-System ermittelt werden können, sind z. B. die *Defect-Removal-Effectiveness* (siehe Abschnitt 2.3.5.3) oder die Anzahl der Änderungsanforderungen und deren Verteilung über die Projektphasen.

Continuous Integration

Schließlich kann im Abschnitt *Build Environment* der POM-Datei auch das eingesetzte *Continuous-Integration-System* referenziert werden, welches der kontinuierlichen Überwachung der fehlerfreien Durchführbarkeit des Build-Prozesses dient.

Da üblicherweise an einem Entwicklungsprojekt mehrere Entwickler beteiligt sind, die aber jeweils nur an einem Teil des Programm-Codes arbeiten und auch nur diesen regelmäßig kompilieren und testen, besteht die Gefahr, dass einzelne Änderungen zu einem inkonsistenten Gesamtsystem führen. Dies kann sich dadurch äußern, dass sich das Gesamtsystem nicht mehr kompilieren lässt, weil sich in einem einzelnen Modul eine Schnittstelle geändert hat, oder dass nicht mehr alle Unit-Tests ohne Fehler durchlaufen werden können, weil sich die Bedeutung eines Schnittstellenparameters geändert hat.

Ein Continuous-Information-Server (CI-Server) wird entweder durch einen Trigger informiert, wenn im Versionsverwaltungsystem eine neue Version einer Quelldatei verfügbar ist, oder wird selbständig in einem vorgegebenen Zeitintervall aktiv. In jedem Fall checkt der CI-Server den kompletten Sourcecode aus und startet den Build-Prozess samt Unit-Tests für die gesamte Anwendung. Das Ergebnis wird üblicherweise mittels einer Web-Status-Seite dargestellt. Zusätzlich bieten die meisten CI-Server an, bei auftretenden Fehlern im Rahmen des Build-Prozesses einen bestimmten Personenkreis (per E-Mail, SMS oder sonstiges) zu informieren.

Nightly Build

Wird ein CI-Server im Rahmen von Open-Source-Projekten eingesetzt, so kann dieser auch als Nebenprodukt in regelmäßigen Abständen eine aktuelle Entwickler-Version der zu implementierenden Anwendung als entsprechend verpacktes Softwarepaket im Internet zur Verfügung stellen. Bekannte CI-Server sind beispielsweise *AnthillPro*[9], *CruiseControl*[10] oder *Apache Continuum*[11].

Broken Build

Falls der vollständige Build-Prozess nicht fehlerfrei beendet werden kann, so bedeutet dies, dass das Gesamtsystem nicht mehr konsistent ist. Daher müssen die Fehler, welche den Abbruch des Build-Prozesses verursacht haben, möglichst zeitnah behoben werden, da ansonsten die Gefahr besteht, dass die ab diesem Zeitpunkt erbrachten Entwicklungsaufwände vergebens sind, da sie wieder rückgängig gemacht werden müssen. Beispielsweise müssten, um auf obiges Beispiel zurückzukommen, alle Quelltexte, welche auf den inkonsistenten Schnittstellenparametern aufbauen,

[6] `http://www.bugzilla.org`
[7] `http://trac.edgewall.org/`
[8] `http://www.ibm.com/software/awdtools/clearquest`
[9] `http://www.anthillpro.com`
[10] `http://cruisecontrol.sourceforge.net`
[11] `http://maven.apache.org/continuum`

nochmals überarbeitet oder unter Umständen auch gelöscht werden.

Damit muss zum einen die Zeitdauer, welche benötigt wird, um einen gestörten Build-Prozess wieder zu korrigieren, als Fehlleistungsaufwand betrachtet werden (vgl. [KB07], S. 80 ff.) und sollte daher entsprechend erfasst und protokolliert werden. Zum anderen ist die Anzahl bzw. die Rate der Build-Prozess-Störungen ein Maß für die Qualität des Entwicklungsprozesses selbst und sollte somit ebenfalls beobachtet werden.

Fehlleistungsaufwand

Leider gibt es auch bei den CI-Servern keinen einheitlichen Standard bzgl. der Benachrichtigungsmöglichkeiten im Fehlerfall bzw. der Möglichkeiten, den aktuellen Status des CI-Servers abzufragen, so dass auch hier für entsprechende Analysen eigene Konnektoren zu entwickeln sind. Jedoch bietet Maven die Möglichkeit, diese Attribute des CI-Systems, d. h. die Art (E-Mail, SMS, etc.) und die Bedingungen (Erfolg, Warnung, Fehler) der Benachrichtigungen des CI-Systems innerhalb der POM-Datei zu pflegen, so dass sie entsprechend ausgewertet werden können.

An dieser Stelle sei erwähnt, dass von Alberto Savoia ([Cla05], S. 132 ff.) ein sehr kreativer Ansatz beschrieben wird, die Zeitdauer, innerhalb derer der Build-Prozess gestört ist, zu messen. Die Grundidee dabei ist, dass die Tatsache, dass der Build-Prozess nicht fehlerfrei durchgeführt werden kann, zunächst einmal nicht so kritisch ist. Problematisch für die Produktivität der Entwicklergruppe wird es erst, wenn dieser Zustand für längere Zeit fortbesteht. Savoia bedient sich zunächst eines kleinen Java-Programmes, welches den Status des Build-Prozesses im Abstand weniger Minuten abfragt. Falls dabei festgestellt wird, dass der Build-Prozess gestört ist, wird über einen elektronischen Schalter eine Lava-Lampe so lange mit Strom versorgt, bis der Build-Prozess wieder funktioniert. Dadurch wird eine kurze Störung des Build-Prozesses lediglich durch das Leuchten der Lava-Lampe signalisiert. Sollte die Störung länger bestehen bleiben, erhitzt sich deren Inhalt und es entstehen die typischen Blasen in der Lava-Lampe.

Kreativer Ansatz zur Überwachung des Build-Prozesses

Bei der Verwendung von Maven können somit die Meta-Informationen des verwendeten SCM-, Issue-Tracking- bzw. CI-Systems innerhalb des Project-Object-Models gepflegt werden. Mittels geeigneter Maven-Plugins, deren Entwicklung im nächsten Abschnitt beschrieben wird, ist auch ein Zugriff auf die in diesen Systemen abgelegten Informationen selbst möglich.

Verwaltung von Meta-Informationen der Entwicklungsumgebung

Diese Ansammlung von Meta-Informationen über die Entwicklungsumgebung ist erweiterbar. Wie in Listing 5.3 auf Seite 125 ersichtlich ist, erfolgt die Konfiguration zusätzlicher Maven-Plugins, wie die des Maven-Torque-Plugins, in der POM-Datei. Somit könnten für ein Maven-Zeiterfassungs-Plugin, auch die Attribute für die Abfrage des eingesetzten Zeiterfassungssystems in der POM-Datei gepflegt werden.

Insgesamt ist es somit möglich, sämtliche Meta-Informationen, welche zur Ermittlung von Softwaremaßen benötigt werden, innerhalb des Project-Object-Models und damit in einer einzigen Datei zu pflegen. Die Ermittlung der Softwaremaße selbst kann dann über Maven-Plugins durchgeführt werden, welche ihre Konfigurationsdaten der POM-Datei entnehmen.

Verwaltung sämtlicher Meta-Informationen

5.1.3 Generierung von Projekt-Berichten

Die im Project-Object-Model abgelegten Informationen dienen neben der Steuerung des Build-Prozesses auch zur Generierung von Projektberichten. Dazu kann Maven so konfiguriert werden, dass im Rahmen des Build-Prozesses mehr oder weniger ausführliche Projektberichte in Form von HTML-Dokumenten erzeugt werden, welche dann automatisiert in die entsprechenden Verzeichnisse eine Web-Servers übertragen werden können. Standardmäßig enthalten die so generierten Projektseiten diejenigen Informationen, welche in den POM-Blöcken *General Project Information* und *Build Environment* (siehe Abbildung 5.3) enthalten sind und im vorhergehenden Abschnitt beschrieben wurden. Ein Ausschnitt eines solchen Standard-Project-Reports ist in Abbildung 5.4 zu sehen, welche die Liste der Softwareentwickler im *Apache Commons Math* Projekt[12] zeigt.

Standardberichte

The Team

A successful project requires many people to play many roles. Some members write code or documentation, while others are valuable as testers, submitting patches and suggestions.

The team is comprised of Members and Contributors. Members have direct access to the source of a project and actively evolve the code-base. Contributors improve the project through submission of patches and suggestions to the Members. The number of Contributors to the project is unbounded. Get involved today. All contributions to the project are greatly appreciated.

Members

The following is a list of developers with commit privileges that have directly contributed to the project in one way or another.

Name	Id	Email	Organization	Roles	TZ Offset	Time
Albert Davidson Chou	achou	achou at apache dot org				Unknown
Mark Diggory	mdiggory	mdiggory at apache dot org				Unknown
Robert Burrell Donkin	rdonkin	rdonkin at apache dot org				Unknown
Tim O'Brien	tobrien	tobrien at apache dot org				Unknown
Luc Maisonobe	luc	luc at apache dot org				Unknown
J. Pietschmann	pietsch	j3322ptm at yahoo dot de				Unknown
Phil Steitz	psteitz	psteitz at apache dot org				Unknown
Brent Worden	brentworden	brentworden at apache dot org				Unknown

Abbildung 5.4: Entwicklerliste als Teil des Maven Standard-Project-Reports

Plugin-Reports

Zusätzlich zu den Standard-Reports können natürlich auch von allen Maven-Plugins Projektberichte generiert werden. So kann das PMD-Maven-Plugin[13] verwendet werden, um den Sourcecode nach bestimmten Regelverletzungen zu untersuchen. Dies können beispielsweise Namenskonventionen (z. B. mind. drei Zeichen für Variablennamen) aber auch nahezu beliebige andere (projektinterne) Programmierrichtlinien (z. B. max. 10 Methodenparameter) sein. In Abbildung 5.5 ist ein beispielhafter Ausschnitt eines derartigen Berichts dargestellt, der Regelverstöße wie einen leeren Catch-Block im Rahmen des Java-Exception-Handlings aufdeckt.

[12] `http://commons.apache.org/math`
[13] `http://pmd.sourceforge.net`

PMD Results

The following document contains the results of PMD.

Summary

Files	Errors
12	19

Files

Files	Violations
org/apache/commons/math/MathException.java	1
org/apache/commons/math/MaxIterationsExceededException.java	1
org/apache/commons/math/estimation/GaussNewtonEstimator.java	2
org/apache/commons/math/linear/BigMatrixImpl.java	3
org/apache/commons/math/linear/MatrixUtils.java	2
org/apache/commons/math/linear/QRDecompositionImpl.java	1
org/apache/commons/math/linear/RealMatrixImpl.java	3
org/apache/commons/math/ode/SwitchingFunctionsHandler.java	1
org/apache/commons/math/random/EmpiricalDistributionImpl.java	2
org/apache/commons/math/random/ValueServer.java	1
org/apache/commons/math/stat/Frequency.java	1
org/apache/commons/math/stat/descriptive/moment/VectorialCovariance.java	1

org/apache/commons/math/MathException.java

Violation	Line
Avoid empty catch blocks	96

org/apache/commons/math/MaxIterationsExceededException.java

Violation	Line
No need to import a type that's in the same package	20

org/apache/commons/math/estimation/GaussNewtonEstimator.java

Violation	Line
Avoid unused local variables such as 'bDecrementData'	157
Avoid unused local variables such as 'wggRow'	174

Abbildung 5.5: PMD-Report über Regelverletzungen im Apache-Commons-Math-Projekt (Ausschnitt)

5.2 Erweiterbarkeit von Maven

5.2.1 Maven Plugins

Maven-Kern

Wie bereits dargestellt, werden die eigentlichen Build-Funktionalitäten von Maven mittels Plugins implementiert. Der Maven-Kern dient dagegen hauptsächlich dazu, die POM-Datei einzulesen und zu analysieren, Projektabhängigkeiten zu verwalten und die einzelnen Plugins zu starten (vgl. [CMP+07], S. 134 f.). Welche Plugins verwendet werden, wird gesteuert, indem einzelne Plugin-Goals mit einer Phase des Maven-Build-Lifecycles verknüpft werden. Sobald diese Phase beim Durchlaufen des Build-Prozesses erreicht wird, werden die verknüpften Goals und damit die entsprechenden Plugins aufgerufen (vgl. auch Abschnitt 5.1.1.3, S. 120 f.).

Maven-Mojos

Jedes Goal innerhalb eines Maven-Plugins wird durch ein sogenanntes *Mojo* implementiert. Die Bezeichnung *Mojo* ist eine Anspielung auf den Begriff *POJO*[14], welcher für *plain old java object* steht, also ein einfaches Java-Objekt ohne umfangreiche externe Abhängigkeiten, wie sie beispielsweise bei EJBs *Enterprise Java Beans* gegeben sind (vgl. [Ric06]).

AbstractMojo als Basisklasse

Ein Maven-Mojo, welches ein Goal implementiert, ist daher eine Java-Klasse, welche im Wesentlichen von der Klasse `AbstractMojo` aus dem Maven-Framework abgeleitet sein muss. Damit besteht ein Maven-Plugin aus einem oder mehreren Mojos und evtl. notwendigen Hilfsklassen, welche zusammen mit der Datei `plugin.xml`, in der die Goals und Parameter des Plugins beschrieben sind, zu einem Jar-Archiv gebündelt werden. Ein sehr einfaches Mojo ist in Listing 5.4 abgebildet.

Dieses Mojo erwartet als Parameter einen String, welcher dann innerhalb von Mavens Logging-Strom mit ausgegeben wird. Dieses Mojo zeigt auch die Art und Weise, mit der innerhalb des Maven-Frameworks Parameter an Mojos und damit an Goals übergeben werden. Dabei bedient sich Maven des *Dependency-Injection*-Prinzips

Dependency Injection

(vgl. [Fow04]). Bei diesem Prinzip wird die Verantwortung für die Erzeugung und Initialisierung von Objekten, welche bei der objektorientierten Programmierung üblicherweise über einen Konstruktor erfolgt, dem Objekt entzogen und dem umgebenden Framework übertragen. Bei Maven werden also Mojos nicht durch parametrisierbare Konstruktor-Methoden initialisiert, sondern das Maven-Framework erzeugt ein Mojo-Objekt und sorgt für die Initialisierung der Mojo-Attribute. Die Motivation für dieses Vorgehen ist, dass dadurch der Programmcode dieser Objekte nicht an bestimmte Interface-Definitionen gebunden ist, welche beispielsweise die Anzahl der Parameter vorgeben würden. Beim Dependency-Injection-Prinzip werden daher die Attribute eines zu erzeugenden Objekts extern konfiguriert, was typischerweise mittels XML-Dateien erfolgt. Somit kann ein Mojo grundsätzlich eine beliebige Anzahl Attribute unterschiedlichster Typen haben. Diese werden in der Datei `plugin.xml` deklariert und das Maven-Framework sorgt zur Laufzeit des Plugins dafür, dass die Mojo-Attribute entsprechend initialisiert werden.

Im obigen Beispiel (Listing 5.4) wird durch die *JavaDoc*-Annotation `@parameter` deklariert, dass das nachfolgende String-Attribut `message` über einen Parameter des

[14] `http://www.martinfowler.com/bliki/POJO.html`

```java
package de.ubt.mavenplugins;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;

/**
 * Ausgabe eines Strings
 *
 * @goal echo
 */
public class EchoMojo extends AbstractMojo {

        /**
         * Auszugebender Text
         *
         * @parameter expression="${echo.message}"
         *            default-value="Hallo!"
         */
        private String message;

        public void execute() throws MojoExecutionException {
                getLog().info(message);
        }
}
```

Listing 5.4: Einfaches Maven-Mojo

Mojos initialisiert wird. Durch die Deklaration `expression="${echo.message}"` wird festgelegt, dass beim Aufruf des Plugins von der Kommandozeile aus mittels

Konfiguration auf Kommandozeile

```
-Decho.message="Hier kommmt die Nachricht"
```

dieser Parameter konfiguriert werden kann. Dadurch wird die *Expression*

```
${echo.message}
```

definiert, deren Wert dem Mojo-Parameter zugewiesen wird.

Zusätzlich wird durch die Parameter-Annotation implizit festgelegt, dass der Goal-Parameter im Plugin-Abschnitt der POM-Datei konfiguriert werden kann, wie es bereits anhand des Maven-Compiler-Plugins in Listing 5.3 dargestellt wurde:

Konfiguration über POM-Datei

```
<configuration>
  <message>Hier steht der Nachrichtentext</message>
</configuration>
```

Schließlich wird noch ein Default-Wert für den Parameter definiert, welcher zum Tragen kommt, falls zur Laufzeit weder die Expression `echo.message` definiert noch eine Konfiguration innerhalb des POM vorhanden ist. Aber auch der Default-Wert

Default-Wert

wird erst zur Laufzeit vom Maven-Framework an das Mojo übergeben. Dadurch ist es möglich, durch die Deklaration

```
@parameter default-value="${project.name}"
```

das entsprechende Attribut des Mojos mit einem Default-Wert zu initialisieren, welcher erst zur Laufzeit des Mojos bekannt ist. Die Expression `${project.name}` wird vom Maven-Framework selbst erzeugt und enthält den aktuellen Projektnamen, der wiederum dem POM des aktuellen Projekts entnommen ist.

Drei Möglichkeiten der Parameterübergabe

Somit gibt es drei Möglichkeiten, um zur Laufzeit eines Plugins Parameter an dessen Mojos zu übergeben:

- Kommandozeilenparameter
- Konfiguration über Einträge in der POM-Datei
- Default-Werte (Wert u. U. erst durch die Projektumgebung bestimmt)

In allen Fällen enthält das Mojo nicht selbst den Code zur Initialisierung der Attribute, sondern die Initialisierung erfolgt gemäß dem *Dependency-Injection*-Prinzip durch das umgebende Maven-Framework.[15] Dies ermöglicht die Erstellung spezieller Plugins zur Ermittlung von Softwaremaßen. Die Messfunktionen können als Plugin-Goals implementiert werden, welche dann aus dem Project-Object-Model ihre Konfigurationsinformationen beziehen.

5.2.2 Beispiel-Plugin: Berechnung der Halstead-Maße

Als beispielhafte Implementierung eines Maven-Plugins zur Softwaremessung soll an dieser Stelle ein Plugin zur Berechnung der Halstead-Software-Maße [Hal77] vorgestellt werden. Dieses wurde von Lorenz Singer im Rahmen seiner Bachelor-Arbeit [Sin06] an der Universität Bayreuth entwickelt.

Halstead-Softwaremaße

Die Halstead-Maße wurden 1977 veröffentlicht und gehören mit zu den bekannteren Softwaremaßen (vgl. [Zus98], S. 1). Sie basieren auf der Annahme, dass die ausführbaren Programmteile aus Operatoren und Operanden aufgebaut sind (vgl. [Wol03]). Dabei werden typischerweise Variablen und Konstanten als Operanden betrachtet. Die restlichen Programmbestandteile sind somit Operatoren (vgl. [Zus98], S. 37). Die Halstead-Maße selbst basieren auf folgenden Basismaßen:

- Anzahl der verwendeten unterschiedlichen Operatoren n_1
- Anzahl der verwendeten unterschiedlichen Operanden n_2
- Gesamtzahl der verwendeten Operatoren N_1
- Gesamtzahl der verwendeten Operanden N_2

[15] Die Objektvariable `message` in der Klasse `EchoMojo` ist zwar als *private* deklariert. Trotzdem ist es über die *Java-Reflection-API* (`http://java.sun.com/docs/books/tutorial/reflect`) möglich, von außen auf Objektvariablen zuzugreifen und diese zu modifizieren. Eine beispielhafte Anleitung dazu findet sich u. a. in [Ull07].

Damit lassen sich u. a. die folgenden Halstead-Maße berechnen:

Implementierungslänge:	$N = N_1 + N_2$
Vokabular:	$n = n_1 + n_2$
Halstead-Länge:	$H_L = n_1 \cdot \log_2 n_1 \; + \; n_2 \cdot \log_2 n_2$
Halstead-Volumen:	$H_V = N \cdot \log_2 n$

Kritik an den Halstead-Maßen

Zusätzlich zu diesen Maßen hat Halstead auch davon abgeleitete Maße für Schwierigkeit, den Intelligenzgehalt und den Implementierungsaufwand von Programmen angegeben (vgl. [Hal77]). Einige der Halstead-Maße, insbesondere die abgeleiteten Maße, wurden in der Literatur oft kritisiert und teilweise als Beispiele für „*confused and inadequate measurement*“ (siehe [FP98]) bezeichnet.

Betrachtung der Implementierungslänge

Jedoch wurde von Wolle in [Wol03] dargestellt, dass sich insbesondere die Implementierungslänge N als geeigneteres Maß für die Längenmessung von Java-Applikationen als das weit verbreitete LOC-Maß erweist. Diese Ansicht wird von Zuse in [Zus03] bestätigt. Zusätzlich wird darauf hingewiesen, dass sich damit gemäß [Zus98] (Kapitel 8) die Implementierungslänge zur Abschätzung des Wartungsaufwandes für Java-Applikationen empfiehlt. Die Implementierungslänge N hat insbesondere gegenüber dem LOC-Maß den Vorteil, dass sie weit weniger stark vom Programmierstil (z. B. Klammersetzung) abhängt als das LOC-Maß.

Trotzdem ist sie maschinell recht einfach zu ermitteln, nachdem festgelegt wurde, welche Java-Sprachelemente als Operatoren bzw. als Operanden gezählt werden. In [Wol03] findet sich beispielsweise die Inkonsistenz, dass Variablendeklarationen als Operatoren gezählt werden, wenn es sich um eine primitive Variable handelt (`int`, `byte`, `boolean`, etc.), sie jedoch als Operanden zählen, wenn es sich um Referenzvariablen (z. B. `String`) handelt. Demgegenüber werden von Singer die Deklarationen von Variablen und Methoden einheitlich als Operanden gezählt, was auch dem Originalansatz[16] von Halstead entspricht (vgl. [Sin06], S. 32 ff.). Diese Unterscheidung macht sich jedoch nur bei der Halstead-Länge H_L bemerkbar. Für die Implementierungslänge N führen beide Zählweisen zum gleichen Ergebnis.

Halstead-Länge und -Volumen

Die Halstead-Länge und das Halstead-Volumen sollen an dieser Stelle auch nicht weiter betrachtet werden, da deren Bedeutung sehr umstritten ist bzw. empfohlen wird, zumindest auf erstere noch eine Korrekturfunktion anzuwenden (vgl. [Zus98], S. 546 f., [Wol03]).

Verwendung des Halstead-Metrics-Plugin

Das im Rahmen von [Sin06] entwickelte Halstead-Metrics-Plugin wird über das Goal `MetricsReport` aufgerufen. Dieses Goal kann unter anderem über die in Listing 5.5 dargestellten Parameter konfiguriert werden.

Dabei werden zunächst die Verzeichnisse für die zu vermessenden Java-Dateien als Parameter definiert, wobei als Voreinstellungen das Source-Verzeichnis für die eigentliche Java-Applikation und das zugehörige Unit-Tests-Verzeichnis hinterlegt

[16] Halstead verwendet in [Hal77] nur Sprachbeispiele für *Algol*, *Fortran* und *PL/1*.

```
1  /**
2   * The source directory to browse.
3   * @parameter default-value="${project.build.sourceDirectory}"
4   */
5  private String sourceDir;
6
7  /**
8   * The test directory to browse.
9   * @parameter default-value="${project.build.testSourceDirectory}"
10  */
11 private String testDir;
12
13 /**
14  * Defines if tests should be analysed.
15  * @parameter default-value="true"
16  */
17 private boolean analyseTests;
18
19 /**
20  * Specifies the directory where the report will be generated
21  *
22  * @parameter default-value="${project.reporting.outputDirectory}"
23  * @required
24  */
25 private File outputDirectory;
```

Listing 5.5: Parameter des Halstead-Metrics-Plugin (Auszug)

Plugin auf beliebige Projekte anwendbar

sind. Diese Vorgaben können jedoch über Einträge in der POM-Datei überschrieben werden. Dadurch ist es möglich, das Halstead-Metrics-Plugin auch auf beliebige andere Java-Projekte anzuwenden, welche nicht Maven als Build-Tool verwenden.

Die Ausgabe erfolgt standardmäßig in dem Verzeichnis, in dem auch die übrigen Reports von Maven abgelegt werden. Schließlich kann noch konfiguriert werden, ob die Halstead-Maße nur für den produktiven Sourcecode oder auch unter Einbeziehung des Test-Sourcecodes berechnet werden sollen.

Beispiel-Report

Die Abbildung 5.6 zeigt beispielhaft einen Ausschnitt aus einem mit diesem Plugin erstellten Maven-Report. Die Einbindung des Plugins in die POM-Datei ist in Listing 5.2 (S. 123) ab Zeile 36 dargestellt.

Zum Abschluss der Betrachtungen dieses Plugins bleibt festzustellen, dass dieses Plugin in erster Linie als Nachweis für die Verwendbarkeit von Maven für die Softwaremessung implementiert wurde. Die Halstead-Maße wurden ausgewählt, weil insbesondere in [Wol03] die Implementierungslänge N und die (korrigierte) Halstead-Länge H_L als im Vergleich zu den *Lines of Code* robustere Maße zur Umfangsmessung von Java-Programmen empfohlen werden. Insbesondere der Halstead-Länge wird diesbezüglich noch weiteres Potenzial zugesprochen, da dieses Maß evtl. doppelt vorhandene Code-Abschnitte nicht mehrfach zählt, da sie allein von dem im

Robuste Maße zur Umfangsmessung

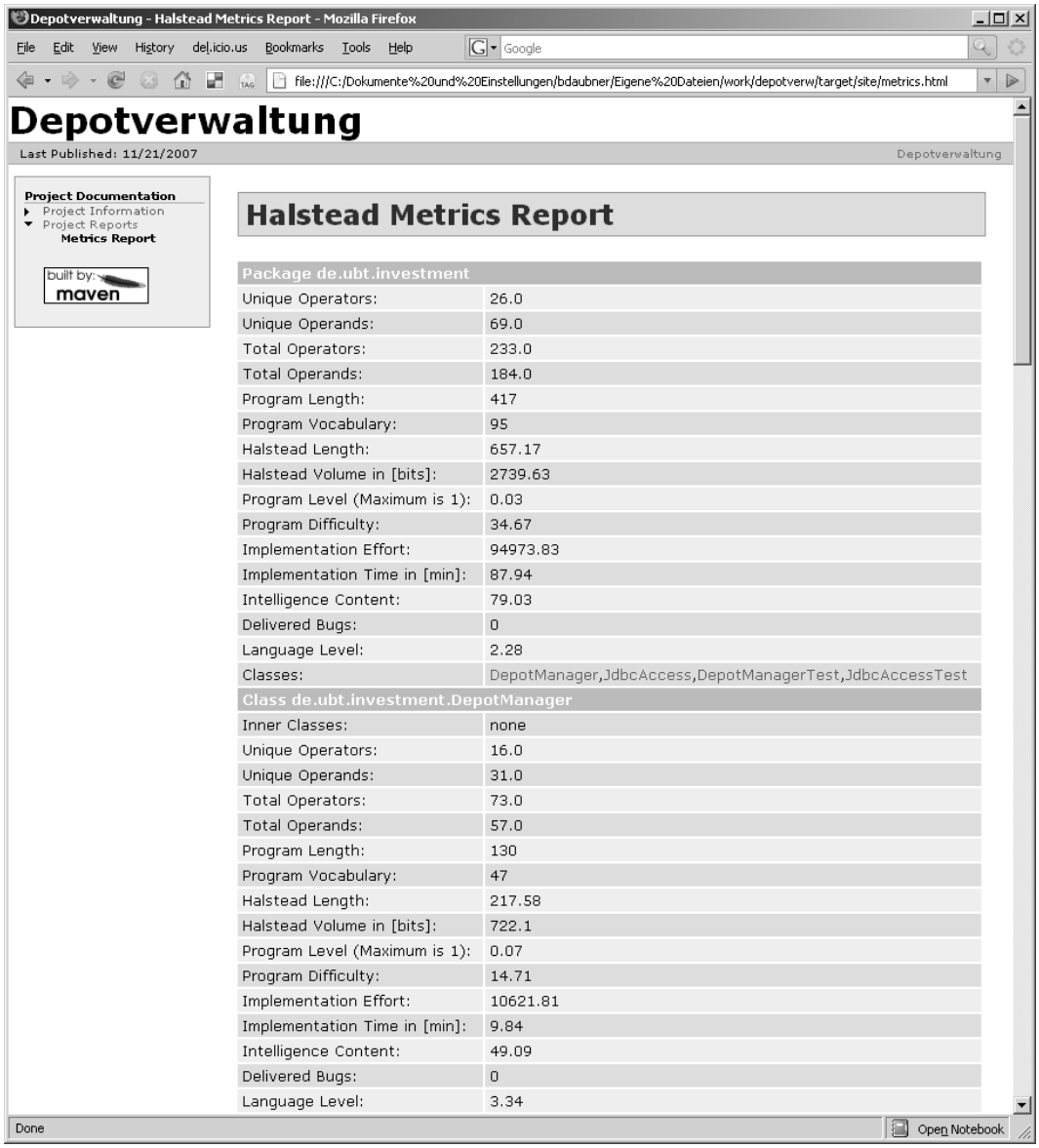


Abbildung 5.6: Durch das Halstead-Metrics-Plugin erzeugter Maven-Report (Ausschnitt)

Programm verwendeten Vokabular abhängig ist. Für die praktische Anwendung dieses Maßes sind jedoch noch weitere Studien notwendig (vgl. [Wol03]).

Implementierungslänge als Umfangsmaß

Die Implementierungslänge N, d. h. die Summe aller verwendeten Operatoren und Operanden, ist jedoch als Längenmaß für Sourcecode akzeptiert, und hat gegenüber den *Lines of Code* den Vorteil, dass dieses Maß vom Programmierstil unabhängig ist und dass lange und kurze Ausdrücke unterschiedlich gezählt werden (vgl. [Wol03] und [Zus03]). Da dieses Maß jedoch äquivalent zu den *Non-commenting Source Statements* (NCSS) [Lee06] ist, wurden diese im Rahmen dieser Arbeit als Längenmaß eingesetzt, da das zugehörige Maven-Plugin regelmäßig aktualisiert wird.

5.3 Messwertgeber als Maven-Plugins

Nachdem im vorhergehenden Abschnitt gezeigt wurde, dass mittels Maven-Plugins grundsätzlich beliebige Softwaremaße automatisiert ermittelt werden können, soll an dieser Stelle nun der Begriff des *Messwertgebers* eingeführt werden.

Messwertgeber

Ein Messwertgeber ist ein Maven-Plugin, welches ein bestimmtes Softwaremaß berechnen kann. Die Bezeichnung *Messwertgeber* wird in der *Messtechnik* für Geräte zur Bestimmung physikalischer oder meteorologischer Größen wie beispielsweise Stromstärke, Druck, Temperatur oder Niederschlagsmenge verwendet. Und ähnlich, wie z. B. ein Windstärkemesser entweder in Bodennähe oder auf einem erhöhten Punkt positioniert werden kann, kann ein Maven-Messwertgeber so konfiguriert werden, dass er nur Messwerte berücksichtigt, die zu einem bestimmten Kontext gehören.

Kontext

Ein Kontext wird dabei durch die im Abschnitt 4.2 dargestellten Kontextinformationen definiert, mittels derer die zu messenden Entitäten vorher gekennzeichnet wurden.

Ein durch einen Messwertgeber ermitteltes Softwaremaß ist somit gekennzeichnet

- durch die Art der Messung, die der Messwertgeber durchführt, und
- durch den Kontext, auf den die Messung eingeschränkt wird.

Konfigurierbarkeit von Messwertgebern

Somit ist es möglich, einen Messwertgeber, der den Umfang bzw. die Umfangsänderung von Sourcecode-Elementen bestimmen kann, derart zu konfigurieren, dass seine Messungen auf den Kontext der Analyse-Aktivitäten einschränkt sind. Die Umfangsmessung selbst wird dabei durchgeführt, indem die zu messenden Sourcecode-Elemente aus dem Versionsverwaltungssystem ausgecheckt und vermessen werden.

Um dabei einen bestimmten Kontext berücksichtigen zu können, muss bei den Commit-Informationen mit angegeben worden sein, im Rahmen welcher Aktivität die jeweiligen Sourcecode-Änderungen durchgeführt wurden. Durch Abfrage des Versionsverwaltungssystems, welches ebenfalls in der Maven-POM-Datei referenziert wird, können anhand der Commit-Informationen die im Rahmen bestimmter Aktivitäten erzeugten oder veränderten Sourcecode-Dateien mit den zugehörigen Versionsnummern identifiziert werden und durch sukzessives *Aus-checken* der betroffenen Versionen kann deren Größenänderung bestimmt werden (vgl. [DHW06a], [DHW06c]).

5.3.1 Konfiguration und Funktion von Messwertgebern

Am Beispiel der Auswertung der Aktivitäts-Kontextinformationen soll die Konfiguration eines Messwertgebers dargestellt werden. Da innerhalb dieses Beispiels die Projektaktivitäten als Anknüpfungspunkte für die Sourcecode-Umfangsmessung verwendet werden, müssen die Commit-Informationen im Versionsverwaltungssystem um die PSP-Codes der Projektaktivitäten ergänzt worden sein (vgl. Abschnitt 4.2, S. 92). Der Standard-Projektstrukturplan habe in diesem Beispiel nur eine Gliederungsebene, wie er beispielsweise im Abschnitt 4.3.1 (S. 106 f.) beschrieben wurde.

Verteilung der Programmiertätigkeiten

Bei diesem PSP finden Programmiertätigkeiten innerhalb der Aktivitäten

- Analyse (in Form von *Prototyping*),
- Implementierung und
- Test (in Form von Programmkorrekturen)

statt. Somit ist es aufschlussreich zu ermitteln, wie sich die im Rahmen dieser Aktivitäten erzeugten Code-Umfänge auf die drei Aktivitäten verteilen. Insbesondere erscheint interessant, ob vielleicht ein relativ großer Anteil des letztendlichen Code-Umfangs bereits im Rahmen des Prototyping entstanden ist. Auch kann auf diese Weise untersucht werden, ob im Rahmen der Programmkorrekturen nur einzelne Programmzeilen überarbeitet wurden, wodurch sich der Gesamtumfang nicht geändert hätte, oder ob sich der Programmumfang beim Einpflegen der Korrekturen signifikant vergrößert hatte, was darauf deuten würde, dass dabei zusätzliche bzw. vorher nicht berücksichtigte Anforderungen implementiert wurden (*Requirements Creep*, vgl. S. 109).

Abgrenzung von Programmiertätigkeiten im Rahmen von Implementierungs- oder Test-Aktivitäten

Im Bezug auf einen einzelnen Programmierer kann dabei folgendermaßen zwischen Programmiertätigkeiten im Rahmen von *Implementierungs-* und *Test-*Aktivitäten unterschieden werden, damit die SCM-Commit-Informationen mit den korrekten Kontextinformationen gekennzeichnet werden können:

Alle Programmänderungen, welche ein Programmierer durchführt, weil er mit dem gerade erstellten Programmabschnitt nicht zufrieden ist oder weil einzelne Unit-Tests fehlgeschlagen sind, welche der Programmierer in kurzen Abständen auf seine Programmsourcen anwendet, zählen als *Implementierungs*-Aktivitäten. Wenn dagegen im Rahmen der *Continuous-Integration*-Tests (vgl. S. 128) festgestellt wird, dass der Build-Prozess des Gesamtsystems nicht mehr erfolgreich durchlaufen werden kann (*Broken Build*, vgl. S. 128), so zählen die zur Korrektur dieser Fehler nötigen Programmiertätigkeiten als *Test-*Aktivitäten. Gleiches gilt für die Behebung von Fehlern, welche im Rahmen von System- oder Akzeptanztests festgestellt werden.[17] Diese Einteilung lässt sich zumindest teilweise auch im *Rational Unified Process* wiederfinden (vgl. [Kru04], S. 193 ff.). Auch hier gehört die Durchführung von Unit-Tests und die Durchführung der daraus resultierenden Code-Korrekturen zur Implemen-

[17] Beim Systemtest wird das Gesamtsystem gegen die Anforderungen oder die Spezifikation getestet. Beim Akzeptanztest wird das System durch den Anwender (Kunden) getestet (vgl. [SL05]). Der Unterschied zwischen den kontinuierlichen Integrationstests und einem Systemtest ist dabei fließend (vgl. [KK05], S. 154 f.), da mit entsprechend ausgearbeiteten Unit-Test-Suiten auch das Gesamtsystem automatisiert getestet werden kann.

tierungsdisziplin (vgl. ebd., S. 195). Die Zuordnung von Programmieraufwänden im Rahmen der Fehlerbehebung zu den Testaktivitäten wird im RUP dagegen nicht erwähnt, jedoch erscheint diese Zuordnung aus Abgrenzungsgründen geboten. Zum einen wird bei der Fehlerbehebung keine neue Funktionalität implementiert, sondern es handelt sich aus betriebswirtschaftlicher Sicht um Nacharbeit.[18] Darüber hinaus muss der überarbeitete Code anschließend erneut getestet werden.

Nacharbeit

Die Aktivitäten *Implementierung* und *Test* finden somit im Rahmen der Programmiertätigkeiten abwechselnd bzw. – bei der Betrachtung mehrerer Programmierer – sogar gleichzeitig statt. Technisch hat dies zur Folge, dass auch die Versionsverwaltungs-Commits abwechselnd Implementierungs- und Test-Aktivitäten betreffen, wie es beispielhaft in Tabelle 5.1 dargestellt ist. Die Lücken innerhalb der Versionsnummern betreffen Commits, welche mit dem hier betrachteten Projekt, bei dem das Versionsverwaltungssystem *Subversion*[19] eingesetzt wurde, nichts zu tun haben.

Revision	Datum	Aktivität
...	...	...
159	2005-03-09	MB-Impl
158	2005-03-08	MB-Test
157	2005-03-07	MB-Impl
156	2005-03-07	MB-Impl
...	...	...
148	2005-02-23	MB-Impl
147	2005-02-23	MB-Impl
146	2005-02-23	MB-Test
...	...	...
142	2005-02-18	MB-Impl
141	2005-02-18	MB-Impl
140	2005-02-18	MB-Test
139	2005-02-17	MB-Impl
138	2005-02-17	MB-Impl
...	...	...

Tabelle 5.1: Versionsverwaltungs-Commits mit Aktivitätsinformationen

Umfangsänderung im Rahmen einer SCM-Version

Falls nun der Umfang des Sourcecodes, der beispielsweise im Rahmen des Commits mit der Versionsnummer 158 hinzugekommen ist, gemessen werden soll, so muss zunächst die Version 157 ausgecheckt und deren Umfang gemessen werden. Anschließend wird die Version 158 selbst ausgecheckt und vermessen. Die Größendifferenz ist

[18] Anders liegt der Sachverhalt, wenn bei einem Akzeptanztest festgestellt wird, dass die ursprünglich definierten Anforderungen unzureichend oder falsch waren. In diesem Fall wird der Kunde üblicherweise einen Change-Request stellen. Die daraus resultierenden Programmieraktivitäten zählen dann als Implementierungs-Aktivitäten.

Bei agilen Softwareentwicklungsprozessen wird die permanente Überarbeitung bzw. Abänderung der Anforderungen ohnehin als erwünscht betrachtet, so dass auch hier entsprechend auf Implementierungs-Aktivitäten kontiert wird.

[19] `http://subversion.tigris.org`

dann der Code-Umfang, der im Rahmen der Test-Aktivitäten (Korrekturarbeiten) hinzugekommen oder gelöscht worden ist.

Soll nicht die Änderung des Programmumfanges, sondern der Umfang der Änderungen, d. h. die Anzahl der geänderten Code-Zeilen ermittelt werden, muss dagegen ein *Diff* bezüglich der genannten Versionen durchgeführt werden.

Messwertgeber werden jedoch in der Regel so konfiguriert, dass als Anknüpfungspunkt für die Softwaremessung nicht ein einzelner Commit, sondern eine bestimmte Aktivität verwendet wird. Im obigen Beispiel wäre dies der Umfang des im Rahmen der Testaktivitäten erzeugten Sourcecodes. Da die Commit-Informationen im Versionsverwaltungssystem um die Aktivitätsinformationen ergänzt wurden, können mittels des PSP-Codes `MB-Test` die Commits identifiziert werden, innerhalb derer Sourcecode-Änderungen auf Grund von Testaktivitäten durchgeführt wurden.

Testaktivitäten als Anknüpfungspunkt

Listing 5.6 zeigt die Verankerung des Maven-Sourcecode-Umfang-Messwertgebers an den Test-Aktivitäten[20] des Beispielprojekts.

```
1  <reporting>
2      <plugins>
3          <plugin>
4              <groupId>de.ubt.mmf</groupId>
5              <artifactId>mmf-scm-ncss</artifactId>
6              <configuration>
7                  <wbsCode>MB-Test</wbsCode>
8                  <scmUser>mmf-user</scmUser>
9                  <scmPassword>geheim</scmPassword>
10             </configuration>
11         </plugin>
12     </plugins>
13 </reporting>
```

Listing 5.6: Konfiguration eines Messwertgebers

5.3.2 Details der Messwertgeber-Implementierung

An dieser Stelle sollen nun einige Implementierungsdetails von Maven-Messwertgebern betrachtet werden. Dadurch sollen sowohl die Vorteile der Verwendung des Maven-Frameworks dargestellt als auch einige Restriktionen erläutert werden, die sich durch die Architektur von Maven ergeben.

In Listing 5.4 (S. 133) wurde bereits der grundsätzliche Aufbau eines Maven-Mojos dargestellt. Im Gegensatz zu einfachen Mojos, welche nur eine bestimmte Aufgabe ausführen und von der in Listing 5.4 aufgeführten Klasse `AbstractMojo` abgeleitet sind, werden Mojos, welche einen Report implementieren, von der Klasse `AbstractMavenReport` – einer direkten Unterklasse von `AbstractMojo` – abgeleitet.

Eigene Basis-Klasse für Report-Mojos

[20] Gemessen wurde hierbei die Veränderung des Sourcecode-Umfanges im Rahmen der Entwicklung der im Abschnitt 3.3.1 erwähnten Anwendung *Metrics Builder* (siehe S. 59).

Diese Klasse enthält neben der von `AbstractMojo` geerbten `execute()`-Methode noch eine als

```
protected abstract void executeReport(Locale locale)
```

deklarierte Methode, über die die Ausführung des Mojos gestartet wird.

Schwierige Parameter-Initialisierung

An dieser Stelle zeigt sich auch ein Einschränkung der Mojo-Architektur und der Parameter-Initialisierung mittels *Dependency Injection*, welche im Abschnitt 5.2.1 (S. 132) beschrieben wurde. Da die Attribute eines Mojos normalerweise durch das Maven-Framework mit Informationen aus dem *Project Object Model* initialisiert werden, haben Maven-Mojos in der Regel keine *Setter*-Methoden und keine parametrisierbaren Konstruktoren. Damit gibt es keine praktikable Möglichkeit, die Attribute eines Mojo-Objekts von einem anderen Objekt aus zu setzen bzw. zu modifizieren.

Es gibt beispielsweise das Mojo

```
org.apache.maven.scm.plugin.CheckoutMojo,
```

welches das Goal `checkout` innerhalb der Methode

```
protected void checkout() throws MojoExecutionException
```

implementiert. Dieses Mojo dient dazu, eine bestimmte Version des Projekts aus dem Versionsverwaltungssystem auszuchecken. Jedoch kann diese `checkout()`-Methode, welche normalerweise beim Ausführung dieses Goals über die `execute()`-Methode aufgerufen wird, nicht von außerhalb der Klasse oder von einer abgeleiteten Klasse aus verwendet werden, da man die dazu notwendigen Objekt-Attribute, wie

- SCM-Verzeichnis,
- SCM-User oder
- SCM-Passwort,

welche als `private` deklariert sind, nicht initialisieren kann.

Bereitstellung entsprechender Hilfsklassen

Daher müssen entsprechende Hilfsklassen implementiert werden, welche – ähnlich den Original-Mojos – die API-Funktionen des Maven-Frameworks aufrufen. Eine im Rahmen des *Maven Measurement Frameworks* (MMF) verwendete Implementierung einer solchen Hilfsklasse ist in Abbildung 5.7 in Gestalt der Klasse `ScmHelper` dargestellt. Dieses UML-Diagramm beschreibt die wichtigsten Klassen des *Maven-NCSS-Report-Plugins*, wobei die Abbildung 5.7 aus Gründen der besseren Übersichtlichkeit nur einen Auszug der eigentlichen UML-Repräsentation der beteiligten Klassen enthält.

So enthält das Maven-Framework in dem Paket `org.apache.maven.scm` die Klassen

- `ScmRepository`,
- `ScmProvider`,
- `ScmManager`,
- `ChangeSet` und
- `ChangeFile`,

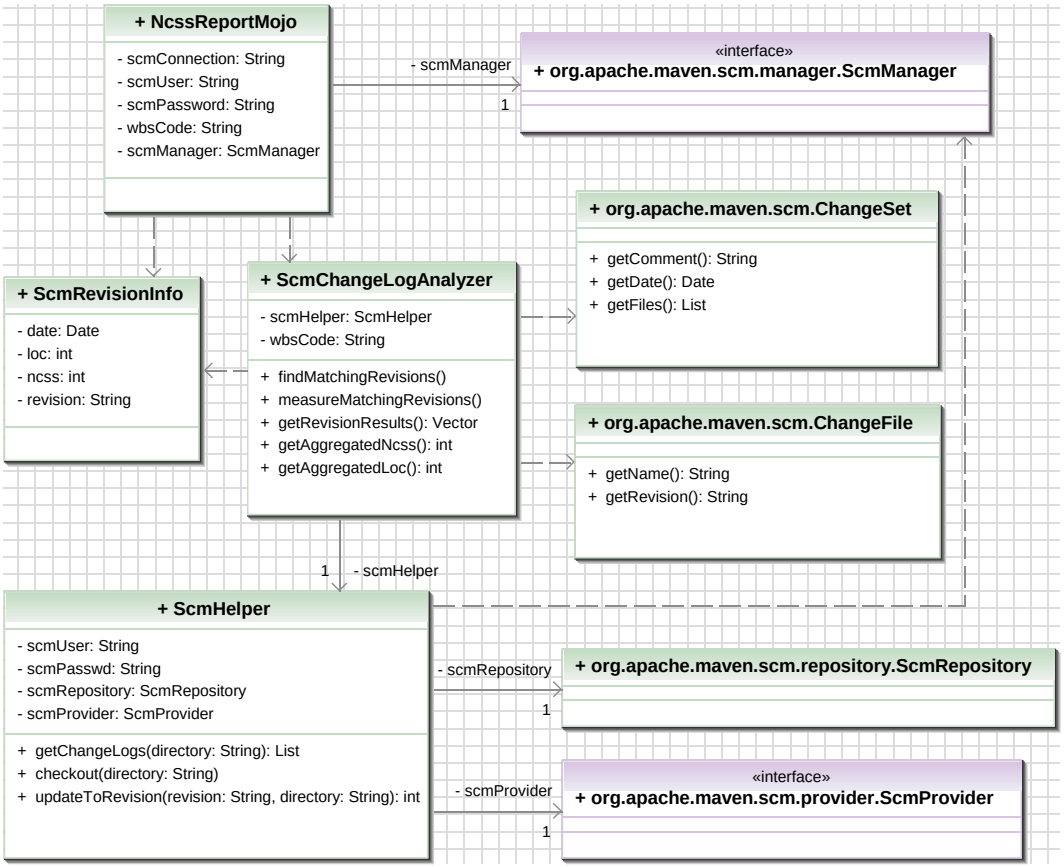


Abbildung 5.7: UML-Diagramm des Maven-NCSS-Report-Plugins (Auszug)

welche Methoden und Objekte zum Zugriff auf Versionsverwaltungssysteme (*SCM-Systems*) zur Verfügung stellen.

Handle-Objekte für den Repository-Zugriff

Die API-Methoden arbeiten dabei unabhängig vom verwendeten Versionsverwaltungssystem. Ein Objekt vom Typ `ScmRepository` stellt dabei ein *Handle* für die Repository-Instanz des Versionsverwaltungssystems dar. Das Interface `ScmProvider` enthält dagegen die Methoden für den Zugriff auf das Versionsverwaltungsystem (*checkout*, *update*, *changeLog*, usw.). Über ein Objekt vom Typ `ScmManager`, welches durch das Maven-Framework instantiiert wird, ist es schließlich möglich, zu einer gegebenen *SCM-Connection-URL*, welche in der POM-Datei konfiguriert wird und einen Identifikator für das verwendete Versionsverwaltungssystem enthält, die zugehörigen Handles vom Typ `ScmRepository` und `ScmProvider` zu erzeugen.

Diese Initialisierungsarbeiten für die `ScmProvider`- und `ScmRepository`-Objekte, welche beispielsweise von den Maven-Plugins `ChangeLogMojo` und `CheckoutMojo` im Hintergrund automatisch durchgeführt werden, müssen in der Klasse `ScmHelper` nachimplementiert werden. Der Grund dafür ist, wie oben bereits erwähnt, dass es bei den genannten Plugins konzeptionell nicht vorgesehen ist, von außen auf deren

Methoden und Objektvariablen zuzugreifen.

Anschließend können die üblichen Versionsverwaltungsfunktionen über das erzeugte Objekt vom Typ `ScmProvider` durchgeführt werden. Eine Auswahl dieser API-Funktionen[21] ist in Tabelle 5.2 (S. 145) dargestellt.

Beispielsweise gibt die Methode `changeLog(...)` ein Objekt vom Typ `ScmChangeLogScmResult` zurück. Dieses wiederum liefert ein `ChangeLogSet` zurück, welches eine Liste der eigentlichen `ChangeSet`-Objekte enthält (vgl. Abbildung 5.7). Das `ChangeSet`-Objekt enthält nun die eigentlichen Informationen zu einem einzelnen Commit-Vorgang innerhalb des Versionsverwaltungssystems. Dazu gehören

- der Autor (d. h. derjenige, der den Commit durchgeführt hat),
- das Datum und die Uhrzeit des Commits,
- die Commit-Information (*commit message*, *versioning log message*) und
- die im Rahmen des Commits betroffenen Dateien.

Die Dateiinformationen sind dabei in `ChangeFile`-Objekten gekapselt (vgl. Abbildung 5.7).

Ermittlung von Code-Umfangsänderungen

Wird nun, wie in Listing 5.6 (S. 141) dargestellt, der Messwertgeber zur Ermittlung von Sourcecode-Umfangsänderungen an den Test-Aktivitäten verankert, müssen die folgenden Funktionen der Klasse `ScmChangeLogAnalyzer` verwendet werden:

`findMatchingRevisions()`
: Anhand des PSP-Codes werden zunächst die Versionen ermittelt, welche im Rahmen der Test-Aktivitäten erzeugt wurden.

`measureMatchingRevisions()`
: Wie auf S. 140 beschrieben, wird für alle betroffenen Versionen zunächst die vorhergehende Version ausgecheckt und diese vermessen, um danach die aktuell betrachtete Version auszuchecken und ebenfalls zu vermessen. Anschließend kann die in der aktuellen Version entstandene Umfangsänderung berechnet werden.

`getRevisionResults()`
: Für die weitere Auswertung bzw. zur Erzeugung entsprechender Reports kann eine Liste von `ScmRevisionInfo`-Objekten zurückgegeben werden, welche für jede Version, auf die obiger PSP-Code passt, die eben berechneten Größenänderungen in den Maßen *LOC* oder *NCSS* enthält. Zusätzlich können die aggregierten Maße für alle betroffenen Versionen abgefragt werden.

Diese Aktionen werden durch das eigentliche Maven-Plugin `NcssReportMojo` aufgerufen. Aus den Daten in den `ScmRevisionInfo`-Objekten wird dabei ein entsprechender Web-Report generiert, der in Abbildung 5.8 (S. 146) exemplarisch dargestellt ist. Hier wurde die (aggregierte) Umfangsänderung für alle Versionen ermittelt, deren PSP-Code auf das Suchmuster `MB-23` passt.

[21] `http://maven.apache.org/scm/projects/apidocs/index.html`

AddScmResult	add(ScmRepository repository, ScmFileSet fileSet, String message) Adds the given files to the source control system
BranchScmResult	branch(ScmRepository repository, ScmFileSet fileSet, String branchName) Create a branch of the source file with a certain branch name
ChangeLogScmResult	changeLog(ScmRepository repository, ScmFileSet fileSet, ScmVersion startVersion, ScmVersion endVersion) Returns the changes that have happend in the source control system between two tags
CheckInScmResult	checkIn(ScmRepository repository, ScmFileSet fileSet, String message) Save the changes you have done into the repository
CheckOutScmResult	checkOut(ScmRepository repository, ScmFileSet fileSet, ScmVersion version) Create a copy of the repository on your local machine
DiffScmResult	diff(ScmRepository scmRepository, ScmFileSet scmFileSet, ScmVersion startVersion, ScmVersion endVersion) Create a diff between two branch/tag/revision
ExportScmResult	export(ScmRepository repository, ScmFileSet fileSet) Create an exported copy of the repository on your local machine
ListScmResult	list(ScmRepository repository, ScmFileSet fileSet, boolean recursive, ScmVersion version) List each element (files and directories) of `fileSet` as they exist in the repository
RemoveScmResult	remove(ScmRepository repository, ScmFileSet fileSet, String message) Removes the given files from the source control system
StatusScmResult	status(ScmRepository repository, ScmFileSet fileSet) Returns the status of the files in the source control system
UpdateScmResult	update(ScmRepository repository, ScmFileSet fileSet, ScmVersion version) Updates the copy on the local machine with the changes in the repository

Tabelle 5.2: Durch das Interface `ScmProvider` zur Verfügung gestellte Versionsverwaltungsfunktionen (Auszug aus der API-Dokumentation)

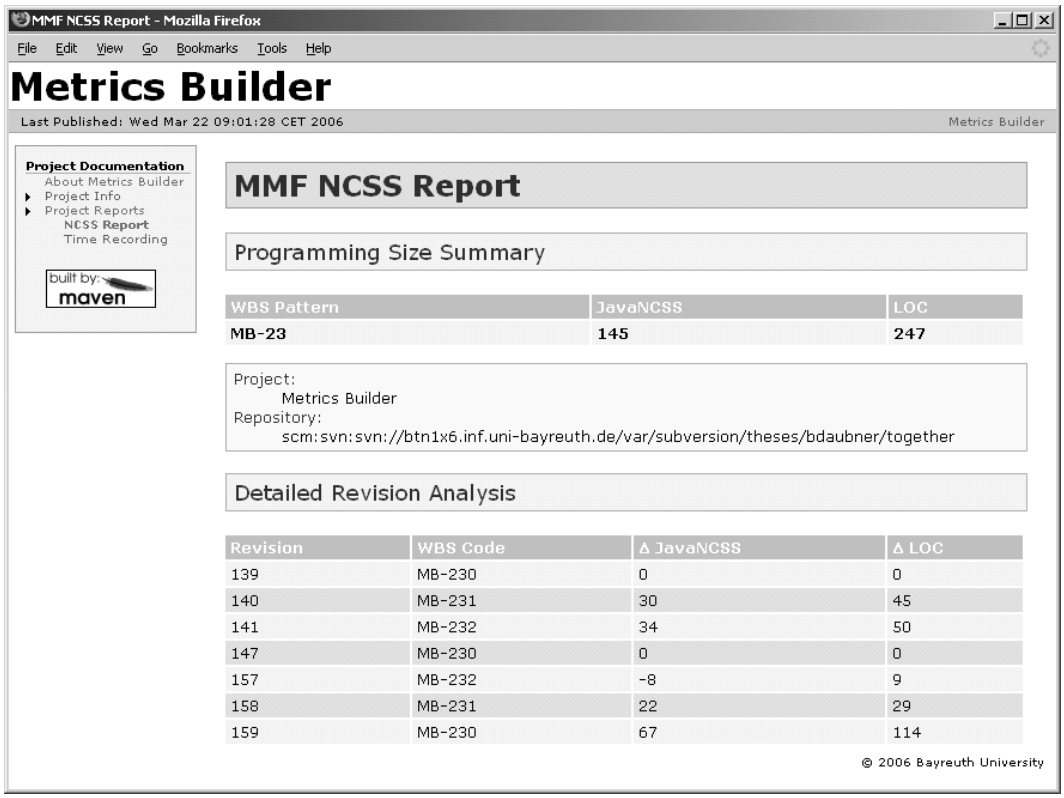


Abbildung 5.8: Untersuchung des Sourcecode-Umfanges für ein Teilprojekt einer Java-Implementierung

5.4 Skriptbasierende Konfiguration von Messwertgebern

Im vorhergehenden Abschnitt wurde die Implementierung einzelner Messwertgeber innerhalb des *Maven Measurement Frameworks* (MMF) dargestellt. Das MMF stellt dabei eine Reihe API-Funktionen zur Verfügung, welche Teilaufgaben im Rahmen des Messvorganges erfüllen. Diese werden innerhalb eines Maven-Plugins derart kombiniert, dass durch das Plugin ein bestimmtes Softwaremaß ermittelt werden kann und das Plugin somit als Messwertgeber fungiert.

Gleichzeitig wurde darauf hingewiesen, dass Maven-Plugins sich nicht gegenseitig aufrufen und dabei Parameter übergeben können, wodurch die Funktionen mehrerer Plugins kombiniert werden könnten.[22]

[22] Es gibt jedoch ein Teilprojekt innerhalb des *Apache Maven Projekts*, innerhalb dessen der sogenannte *Maven Embedder* entwickelt wird:
`http://maven.apache.org/guides/mini/guide-embedding-m2.html`
Mittels des Maven-Embedders können Maven-Goals von beliebigen Anwendungen (und damit auch von Mojos) aus aufgerufen werden. Jedoch war dieses Projekt in dem Zeitraum, als das MMF entstand, noch nicht hinreichend entwickelt. Auch zum derzeitigen Stand ist damit ein Austausch von Parametern nur sehr eingeschränkt möglich.

Somit müsste für jedes Softwaremaß und für jede Kombination von Softwaremaßen ein eigenes Maven-Plugin implementiert werden. Dieses würde jedoch in der Regel nur einige wenige MMF-API-Aufrufe enthalten, um dann beispielsweise die Ergebnisse zweier Messfunktionen (arithmetisch) zu kombinieren. Für die Entwicklung solcher Plugins müsste eine entsprechende Entwicklungsumgebung vorgehalten werden und neue Plugins müssten, bevor sie innerhalb eines Maven-gestützten Entwicklungsprojektes verwendet werden können, zunächst im organisationsinternen Maven-Repository installiert werden.

Aufwendige Komination von Softwaremaßen

Daher wurde im Rahmen des MMF eine Möglichkeit entwickelt, die tatsächliche Funktionalität von Messwertgebern zur Laufzeit des Projekts zu konfigurieren. Dies wird erreicht, indem die Ablauflogik des Messwertgebers mittels der Skriptsprache *Groovy* definiert werden kann.

5.4.1 Die Skriptsprache *Groovy*

Die Ursprünge der Skriptsprache *Groovy*[23] gehen zurück auf das Jahr 2003. Die Softwareentwickler James Strachan und Bob McWhirter erkannten, dass bei einem typischen Java-Projekt auf Komponenten verschiedener Frameworks zurückgegriffen wird und diese im Wesentlichen nur noch zusammengefügt werden. Diese Tätigkeit sollte durch eine Skriptsprache unterstützt werden (vgl. [Kön07], S. xxi). Groovy-Skripte werden vor ihrem Ablauf in Java-Bytecode übersetzt, welcher von der *Java Virtual Machine* ausgeführt wird. Während Groovy somit auf Bytecode-Ebene äquivalent zu Java ist, gibt es auf Syntax-Ebene deutliche Unterschiede zu Java.

Herkunft von Groovy

Beispielsweise unterstützt Groovy eine dynamische Typisierung von Variablen, bei der der Typ der Variablen erst zur Laufzeit durch den Typ des zugewiesenen Objekts festgelegt wird. Eine andere Spracherweiterung stellen die *Closures* dar. Dabei handelt es sich um Code-Blöcke, welche wie Objekte behandelt werden und beispielsweise als Parameter in Methodenaufrufen verwendet werden können.

Groovy vs. Java

Ähnliche Ansätze bietet zwar auch die Skriptsprache *BeanShell*[24], deren Syntax viel ähnlicher zu Java ist, als dies bei Groovy der Fall ist. Jedoch erfreut Groovy sich insbesondere seit der Veröffentlichung des Web-Entwicklungs-Frameworks *Grails* einer steigenden Beliebtheit bei Java-Entwicklern, da damit das Entwicklungsparadigma von *Ruby on Rails* [THB+06] mit den Möglichkeiten der umfangreichen Java-Bibliotheken verknüpft werden kann (vgl. [KA07]).

Umfeld von Groovy

Einbindung von Groovy in Java-Programme

Was Groovy für die Verwendung im Rahmen des *Maven Measurement Frameworks* besonders auszeichnet, ist die Möglichkeit, Groovy-Skripte in Java-Programme einbetten zu können. Dadurch ist es möglich, einen Teil der Ablauflogik einer Java-Anwendung variabel halten zu können, indem dieser Teil als Groovy-Skript implementiert wird und daher vom Benutzer der Anwendung angepasst werden kann.

[23] `http://groovy.codehaus.org`
[24] `http://beanshell.org`

Addition mittels Groovy-Skript

Die Listings 5.7 und 5.8 sollen diese Vorgehensweise illustrieren. In Listing 5.7 wird eine Klasse `GroovyCaller` implementiert, welche zwei Integerwerte an ein Groovy-Skript übergibt, welches durch seinen Dateinamen spezifiziert wird. Innerhalb des Skripts werden die zwei Werte arithmetisch miteinander verknüpft (z. B. addiert) und das Ergebnis wird an das aufrufende Java-Programm zurückgegeben. Zur Übergabe der Parameter an das Groovy-Skript werden diese in einem `Binding`-Objekt gekapselt.

Binding-Objekt für den Datenaustausch

Innerhalb des Groovy-Skripts (Listing 5.8) kann nun über die Variablennamen (`summand1` und `summand2`) auf die im Binding-Objekt gespeicherten Integerwerte zugegriffen werden. In Groovy werden dabei auch primitive Datentypen intern als Objekte verwaltet (Zeile 7). Zusätzlich ist es möglich, Operatoren zu überladen, so dass auch die Addition zweier Objekte möglich ist (Zeile 10).

Die Auslagerung der Ablauflogik in das Groovy-Skript ermöglicht es, die Implementierung der arithmetischen Operationen jederzeit zu überarbeiten und zu modifizieren. Dazu muss weder der Java-Programm-Code neu kompiliert werden noch der Quelltext des Java-Programmes verfügbar sein.

5.4.2 Implementierung von Messwertgebern mittels Groovy

Auslagerung der Ablauflogik

Auf gleiche Weise lässt sich auch die Ablauflogik eines Messwertgebers in ein Groovy-Skript auslagern. Das in Java implementierte Maven-Plugin dient dabei nur noch zum Sammeln der System- und der Projektkonfiguration und delegiert den eigentlichen Messvorgang an ein Groovy-Skript. Daher soll ein derartiges Plugin im Weiteren als *Meta-Mojo* bezeichnet werden.

Meta-Mojo

Das Meta-Mojo enthält dazu die in den Listings 5.9 oder 5.10 dargestellte Parameterliste, mittels derer die Projektinformationen durch das Maven-Framework an das Plugin übergeben werden. Hierzu zählen insbesondere die im *Project Object Model* konfigurierten Daten wie

- *Resource Identifier* für die Versionsverwaltungs-, Bug-Tracking- und Zeiterfassungssysteme,
- der Anknüpfungskontext für die Softwaremaße (WBS-Code, Phasenbezeichnungen, etc.) und
- generierte Laufzeitobjekte (z. B. `ScmManager`-Objekt, vgl. S. 143).

Zusätzlich muss in der POM-Datei auch der Pfad des Groovy-Skripts konfiguriert werden, in dem der Ablauf der eigentlichen Softwaremessung implementiert ist.

Initialisierung des Meta-Mojos durch das Maven-Framework

Unter Verwendung einiger der in Abbildung 5.7 (S. 143) dargestellten Klassen aus dem MMF-API können dann sowohl das Meta-Mojo (Listing 5.9) als auch die Ablaufimplementierung in Groovy (Listing 5.10) sehr kompakt und übersichtlich gehalten werden. Die Objekt-Variablen des Meta-Mojos werden durch das Maven-Framework initialisiert. Anschließend können Hilfs- und Handle-Objekte wie das `ScmHelper`- bzw. das `ScmChangeLogAnalyzer`-Objekt erzeugt werden. Im letztgenannten Objekt sind alle Informationen gekapselt, die für den Zugriff auf das Versionsverwaltungssystem notwendig sind, und können somit bequem an das Groovy-Skript übergeben werden.

```
1 // file: GroovyCaller.java
2
3 import java.io.File;
4 import java.io.IOException;
5
6 import org.codehaus.groovy.control.CompilationFailedException;
7
8 import groovy.lang.Binding;
9 import groovy.lang.GroovyShell;
10
11 public class GroovyCaller {
12
13     public static void main(String[] args)
14       throws CompilationFailedException, IOException {
15
16         File skript = new File("auswertung.groovy");
17
18         Binding binding = new Binding();
19         binding.setVariable("summand1", new Integer(2));
20         binding.setVariable("summand2", new Integer(3));
21
22         GroovyShell shell = new GroovyShell(binding);
23
24         // Das Skript berechnet 'summand1 + summand2'
25         Integer result = (Integer) shell.evaluate(skript);
26         assert result.equals(new Integer(5));
27     }
28
29 }
```

Listing 5.7: Aufruf eines Groovy-Skripts durch ein Java-Programm

```
1 // file: auswertung.groovy
2
3 Vector<Integer> summanden = new Vector<Integer>();
4 summanden.add(summand1);
5 summanden.add(summand2);
6
7 def summe = 0; // Summe ist ein Integer-Objekt!
8
9 for (summand in summanden) { // foreach-Schleife
10   summe += summand;  // Addition zweier Objekte!
11 }
12
13 return summe;
```

Listing 5.8: Einfaches Groovy-Skript zur Addition zweier Werte

Fixe und variable Bestandteile des Frameworks

Es sei an dieser Stelle noch einmal betont, dass bei der Durchführung der Softwaremessung, d. h. zur Laufzeit des Softwaremessungs-Framework, alle Java-Klassen in kompilierter Form im lokalen Maven-Repository der Organisation vorliegen und auch nicht modifiziert werden. Das Groovy-Skript, welches im Rahmen der Softwaremessung aufgerufen wird, befindet sich dagegen im Projektverzeichnis und kann von den die Messung durchführenden Projektmitarbeitern jederzeit überarbeitet und modifiziert werden.

Das in Listing 5.10 dargestellte Beispiel bezieht sich auf dieselben Umfangsänderungen, die bereits in dem Report in Abbildung 5.8 (S. 146) dargestellt wurden. Zunächst werden dabei über das `ScmChangeLogAnalyzer`-Objekt alle Versionen innerhalb des Versionsverwaltungssystems ermittelt, bei denen die beim Commit angegebene Kontextinformation auf das Suchmuster `MB-23` passt, und deren Umfangsänderung vermessen (vgl. Abbildung 5.8). Anschließend wird die (absolute) durchschnittliche Umfangsänderung ermittelt.

Verknüpfung verschiedener Softwaremaße

Auf diese Weise können nun auch mehrere Softwaremaße miteinander verknüpft werden. Genauso, wie über API-Aufrufe die Umfangsänderungen des Sourcecodes ermittelt werden können, könnten auch die entsprechenden Arbeitszeitdaten aus der Zeiterfassung abgefragt werden. Zusammen mit den Umfangsdaten ließen sich daraus Produktivitätskennzahlen berechnen.[25]

In Listing 5.10 wird übrigens nur die Ablauflogik der Softwaremessung dargestellt. Die gewonnenen Daten können nun entweder

- an ein Report-Objekt übergeben werden, welches daraus eine Web-Darstellung erzeugt,
- in einer Datenbank zur späteren Auswertung abgelegt werden oder
- an ein anderes Groovy-Skript zur Weiterverarbeitung übergeben werden.

Alternative zur Verwendung einer Skript-Sprache

Um Softwaremaße variabel definieren zu können, wäre anstelle der Verwendung einer Skriptsprache auch eine deklarative Semantik für die Definition von Messwertgebern denkbar. Grundsätzlich ist es genauso möglich, die Verknüpfung von Softwaremaßen mittels eines entsprechenden XML-Schemas oder einer geeigneten deklarativen Sprache zu definieren.

Generic Metric Extraction Framework

Dieser Weg wird beispielsweise von Alikacem und Sahraoui beschritten. In [AS06] beschreiben sie ein *Generic Metric Extraction Framework* für Sourcecode-Softwaremaße. Dazu führen sie den zu untersuchenden Sourcecode zunächst in eine auf einem generischen Metamodell basierende Code-Repräsentation über. Auf diese Code-Repräsentation können dann Ausdrücke, welche in einer *Metric Description Language* formuliert sind, angewendet werden, um dadurch die gewünschten Softwaremaße zu berechnen. Die *Metric Description Language* selbst besteht aus einer Menge von Primitiven und Operatoren, mit denen die gesuchten Softwaremaße beschrieben werden können.

[25] Im Rahmen dieser Arbeit wurden jedoch nur die Methoden zum Zugriff auf das Versionsverwaltungssystem auch tatsächlich als API-Funktionalitäten implementiert.

```
1  public class MetaMojo extends AbstractMojo {
2
3      [...] // Objekt-Variablen des Plugins
4
5      public void execute() throws MojoExecutionException {
6
7          File scriptFile = new File(basedir, scriptname);
8
9          ScmHelper scmHelper = configureScmHelper();
10         ScmChangeLogAnalyzer changeLogAnalyzer =
11           prepareChangeLogAnalyzer(scmHelper, wbsCode);
12
13         Binding binding = new Binding();
14         binding.setVariable("changeLogAnalyzer",
15                             changeLogAnalyzer);
16
17         GroovyShell shell = new GroovyShell(binding);
18
19         Integer result = (Integer) shell.evaluate(scriptFile);
20
21     }
22
23 }
```

Listing 5.9: Aufruf eines Groovy-Skripts durch ein Maven-Plugin

```
1  // file: mmfscript.groovy
2
3  import de.ubt.mmf.scm.*;
4
5  changeLogAnalyzer.findMatchingRevisions();
6  changeLogAnalyzer.measureMatchingRevisions();
7
8  Vector<ScmRevisionInfo> revisionList =
9    changeLogAnalyzer.getRevisionResults();
10
11 int sum = 0;
12
13 for (revInfo in revisionList) {
14     sum += Math.abs(revInfo.getNcss());
15 }
16
17 double average = sum/revisionList.size();
18
19 [...]
```

Listing 5.10: Implementierung eines Messwertgebers als Groovy-Skript

Beispielsweise kann mit diesen Ausdrücken die *Anzahl der unabhängigen Klassen* (NIC) bestimmt werden, d. h. derjenigen Klassen, welche weder von anderen Klassen abgeleitet sind, noch als Oberklasse für andere Klassen fungieren. Der entsprechende Ausdruck sieht dann folgendermaßen aus:

Number of Independent Classes

$$\text{NIC} = |\ \text{forAll}(x : \text{classes}();\ |\text{parent}(x)| == 0\ \&\&\ |\text{children}(x)| == 0;\ SET+ = x)\ |$$

Alikacem und Sahraoui heben als Vorteile ihres Ansatzes hervor, dass Softwaremaße mit ihrer *Metric Description Language* sehr präzise definiert werden könnten und die Sprache auch für Nicht-Programmierer verwendbar sei.

Gründe für operationale Semantik

Im Gegensatz zu [AS06] stehen jedoch bei der vorliegenden Arbeit anstelle von Sourcecode-Maßen eher die auf Managementebene relevanten Softwaremaße im Vordergrund, wo eine operationale Semantik für die Definition von Softwaremaßen mehr Flexibilität verspricht. Da die arithmetischen und mengenwertigen Operationen, die bei der Berechnung der in dieser Arbeit beschriebenen Softwaremaße auftreten, relativ einfach sind, sollte es für den Anwender übersichtlicher und zielführender sein, die Softwaremaße mittels einer Programmiersprache definieren zu können.

5.5 Zusammenfassung

In diesem Kapitel wurde das Konzept des im Rahmen dieser Arbeit implementierten *Maven Measurement Frameworks* dargestellt. Basierend auf den im Kapitel 4 vorgestellten *Anknüpfungspunkten für Softwaremaße* werden durch einen Messwertgeber nur diejenigen Entitäten bei der Softwaremessung berücksichtigt, welche in einem bestimmten Kontext innerhalb des Softwareentwicklungsprozesses entstanden sind.

Gründe für die Verwendung von Maven

Für die Implementierung der Messwertgeber wurde auf die Projektverwaltungssoftware *Maven* zurückgegriffen. Die Motivation dafür ist zum einen, dass Maven ohnehin schon in vielen Projekten als Build-Management- und Projektverwaltungs-Software eingesetzt wird. Dadurch sind viele Projektinformationen, die für die Softwaremessung benötigt werden, im *Project Object Model* (POM) hinterlegt. Das POM selbst wird dabei in Form einer XML-Datei gepflegt. Darüber hinaus verfügt Maven über wichtige API-Funktionalitäten, welche den Zugriff auf Systeme der Projektinfrastruktur, wie beispielsweise das Versionsverwaltungssystem, ermöglichen. Diese API-Funktionalitäten werden auch beständig von der Maven-Community weiterentwickelt und um neue Funktionen ergänzt. Und schließlich ist Maven selbst aufgrund seiner auf Plugins basierenden Architektur modular aufgebaut und kann daher erweitert und um Funktionen zur Softwaremessung ergänzt werden.

Hart-codierte Messwertgeber

Ein Messwertgeber kann zunächst einmal als Maven-Plugin in der Programmiersprache Java implementiert werden. Das Plugin erfüllt seine Messaufgaben, indem es entsprechende Methoden des Maven-API oder des MMF-API aufruft, welche jeweils Funktionen zum Zugriff auf die zu messenden Entitäten zur Verfügung stellen. Derartige Messwertgeber stehen als kompilierte Java-Klassen im Maven-Repository der jeweiligen Organisation zur Verfügung und können zur Laufzeit des zu messenden Projektes über die POM-Datei konfiguriert werden.

Variable Messwertgeber

Eine variablere Form von Messwertgebern kann mit Hilfe der Skriptsprache *Groovy* implementiert werden. Dabei wird nur das *Meta-Mojo*, welches zum Aufrufen eines Groovy-Skripts dient, in Java implementiert und im Maven-Repository installiert. Die eigentliche Ablauflogik für die Softwaremessung wird durch das Groovy-Skript definiert, welches im Maven-Verzeichnis des zu messenden Softwareprojektes abgelegt ist und jederzeit zur Laufzeit des Projektes angepasst werden kann.

In der Praxis wird man beide Methoden zur Implementierung von Messwertgebern parallel benutzen. Für Softwaremaße, welche für jedes Projekt grundsätzlich in gleicher Weise ermittelt werden sollen, ist es von Vorteil, wenn der zugehörige Messwertgeber als (qualitätsgeprüftes) Java-Maven-Plugin im Maven-Repository installiert ist und von den Projektbeteiligten nicht modifiziert werden kann. Auf der anderen Seite ermöglichen es die als Groovy-Skripte implementierten Messwertgeber, bei der Durchführung von Softwaremessungen auf projektspezifische Besonderheiten einzugehen und diese bei der Ablauflogik des Messwertgebers zu berücksichtigen.

Darüber hinaus ist es grundsätzlich auch möglich, die Messwertgeber in der Sprache Groovy zu implementieren, aber dann die Groovy-Skripte zusammen mit den Maven-Plugins im Maven-Repository zu installieren, wo sie zur Laufzeit des Projektes nicht mehr modifiziert werden können. Eine Motivation dafür könnte sein, dass Groovy explizit zum Verknüpfen einzelner Komponenten innerhalb eines Frameworks entworfen wurde, so dass die Implementierung von Messwertgebern mittels Groovy sehr effizient durchzuführen ist.

6 Evaluierung und Einsatzmöglichkeiten

> A problem is a chance for you to do your best.
>
> *(Duke Ellington)*

6.1 Studentische Softwareentwicklungsprojekte

Das aktuelle Kapitel ist der praktischen Evaluierung und der Darstellung von Einsatzmöglichkeiten des in dieser Arbeit vorgestellten „leichtgewichtigen“ Verfahrens zur Softwaremessung gewidmet. Die praktische Erprobung dieses Ansatzes erfolgte innerhalb zweier studentischer Softwareentwicklungsprojekte, welche im Rahmen der Informatik-Ausbildung an der Universität Bayreuth durchgeführt wurden. Bei den Projekten handelt es sich um ein während einer Bachelor-Arbeit und um ein als Softwarepraktikum durchgeführtes Softwareentwicklungsprojekt.

Zwar wäre es grundsätzlich wünschenswert gewesen, diesen Ansatz auch in Kooperation mit entsprechenden Industriepartnern zu testen, jedoch konnte keine auf dem Gebiet der Softwareentwicklung tätige Firma für dieses Vorhaben gewonnen werden. Die Gründe dafür sind zum einen in dem nicht unerheblichen Aufwand zu suchen, der bei einem Industriepartner im Rahmen der Bereitstellung des notwendigen Datenmaterials angefallen wäre. Darüber hinaus müsste ein Industriepartner bei einer derartigen Zusammenarbeit auch entsprechende organisationsinterne Betriebsinformationen preisgeben, was in der Regel nicht im Interesse von im kommerziellen Umfeld arbeitenden Unternehmen ist.

Hindernisse bei der Gewinnung externer Industriepartner

6.1.1 Monitoring eines eXtreme-Programming-Projekts

In der Bachelor-Arbeit „Fortschrittsmessung bei agilen Softwareentwicklungsprojekten unter Verwendung von Open-Source-Werkzeugen“ [Har06], welche von Holger Hartmann an der *Universität Bayreuth* angefertigt wurde, wurden beispielhaft Möglichkeiten untersucht, den Projektfortschritt eines kleinen Softwareentwicklungsprojektes zu ermitteln und zu visualisieren.

Das Softwareprojekt bestand darin, ein Zeiterfassungssystem zu entwickeln, welches auf die Bedürfnisse von Softwareentwicklern zugeschnitten sein sollte. Dazu sollte insbesondere bei jedem Zeiterfassungsdatensatz – zusätzlich zu den Personen- und Zeit-Informationen – verpflichtend eine Tätigkeitskategorie (z. B. Programmierung, Test, Besprechung) und optional eine Arbeitspaketnummer erfasst werden können.

Entwicklung eines Zeiterfassungssystems

Das Entwicklungsprojekt wurde iterativ – einem agilen Softwareentwicklungsansatz folgend – umgesetzt. Dabei wurde im Rahmen mindestens wöchentlich stattfindender Besprechungen zwischen dem Programmierer und dem Auftraggeber verein-

bart, welche Anforderungen die Anwendung grundsätzlich zu erfüllen hat und welche innerhalb der jeweils nächsten Woche implementiert werden sollten. Es wurde zwischen Anforderungen unterschieden, welche durch Analyse sogenannter *User-Stories* gewonnen wurden, wie sie auch beim *eXtreme-Programming* [Bec04] zur Anwendung kommen, und solchen, welche explizit formuliert waren.

Beispiel einer User-Story

Aus den User-Stories wurde beispielsweise die Anforderung abgeleitet, dass es möglich sein soll, bei der Zeiterfassung ein Arbeitspaket mit anzugeben. Dieses Feld muss jedoch optional sein, da es auch Tätigkeiten gibt, welche keinem Arbeitspaket aus einem Projekt zugeordnet werden können (z. B. Einspielen von Service-Packs auf dem Arbeitsplatzrechner). Jedoch soll grundsätzlich eine Tätigkeitskategorie angegeben werden müssen, um die Zeiterfassung lückenlos durchführen zu können. Eine geeignete Tätigkeitskategorie für die gerade genannte Tätigkeit wäre z. B. „Systempflege“.

Beispiel einer expliziten Anforderung

Eine explizite Anforderung an das zu entwickelnde System war dagegen, dass die Anwendung über eine Property-Datei konfiguriert werden kann, auf die eine entsprechende Umgebungsvariable verweist.

Die Anwendung selbst besteht aus einer API-Bibliothek, welche entsprechende Methoden für die Zeiterfassung zur Verfügung stellt, und einem Kommandozeilen-Tool, welches dieses API verwendet und die Benutzung der Zeiterfassungsfunktionen von der Kommandozeile aus erlaubt. Die Zeiterfassungdatensätze werden in einer *PostgreSQL*-Datenbank[1] abgelegt.

Manuelle Ermittlung der Softwaremaße

An dieser Stelle ist anzumerken, dass das Monitoring des Projektverlaufes nicht mittels des Maven-Measurement-Frameworks (MMF) durchgeführt wurde, sondern manuell erfolgte, weil das MMF zum Zeitpunkt der Anfertigung der Bachelor-Arbeit noch gar nicht entsprechend fertiggestellt war. Trotzdem wird innerhalb der vorliegenden Arbeit dieses Softwareentwicklungsprojekt als Beispiel für die Anwendbarkeit des MMF aufgeführt, da die nachfolgend beschriebenen Messungen ebenso mit dem MMF hätten durchgeführt werden können. Die Beispiele aus diesem Projekt zeigen somit auch, an welchen Stellen des Softwareentwicklungsprozesses das MMF nutzbringend eingesetzt werden kann.

Abbildung 6.1 zeigt die Entwicklung der Anzahl der Anforderungen im zeitlichen Verlauf des genannten Projektes. Wie gerade beschrieben wurden die Anforderungen manuell gezählt. Genauso gut wäre es jedoch möglich gewesen, die Anforderungen in einem Issue-Tracking-System zu verwalten und mittels eines geeigneten Messwertgebers zu zählen. Die Zahl der Anforderungen stieg dabei im Projektzeitraum kontinuierlich bis zu einer Gesamtzahl von 37 Anforderungen an. Davon waren 25 Anforderungen aus User-Stories abgeleitet und 12 Anforderungen explizit durch den Auftraggeber gestellt.

In Abbildung 6.2 ist die zugehörige Entwicklung des Code-Umfanges zu sehen, welche in *NCSS* gemessen wurde. Dabei wurde zwischen dem Sourcecode für die Anwendung selbst und dem Test-Code unterschieden. Zusätzlich ist die Entwicklung des Gesamt-Code-Umfanges aufgeführt.

[1] `http://www.postgresql.org`

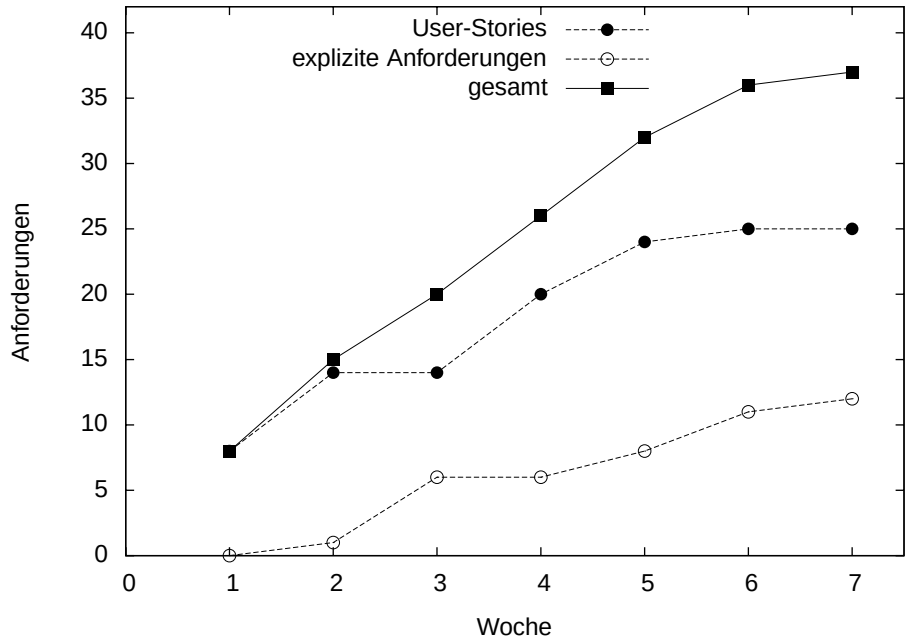


Abbildung 6.1: Entwicklung der Anforderungen (aus [Har06])

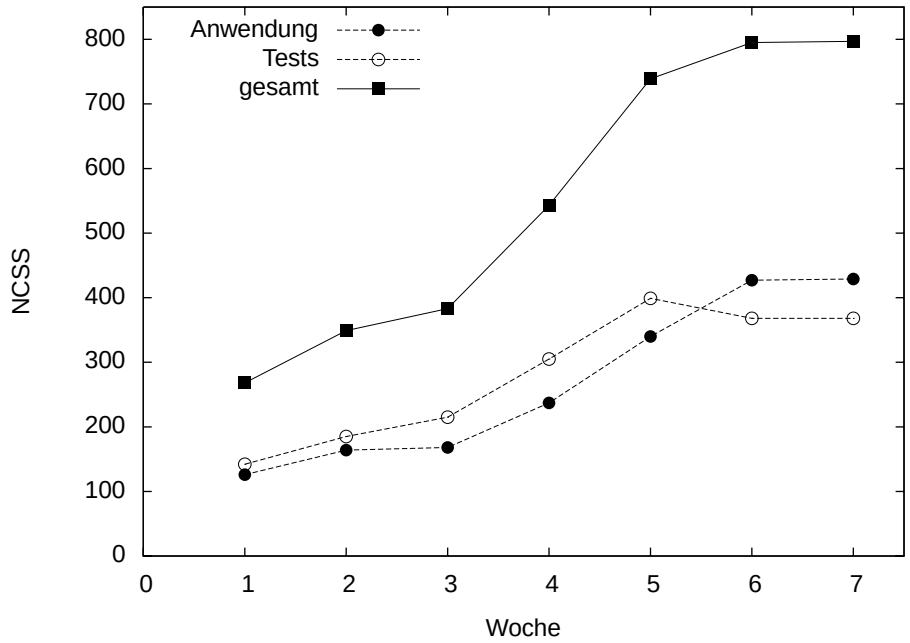


Abbildung 6.2: Entwicklung des Code-Umfangs (aus [Har06])

Die Messungen wurden jeweils am Ende der betrachteten Woche durchgeführt. Da das Projekt relativ klein war, konnten in jeder Woche immer alle Anforderungen vollständig implementiert werden, die in der jeweils betrachteten Woche zusammen mit dem Auftraggeber erarbeitet worden waren. Somit entspricht der pro Woche dargestellte Code-Umfang allen bis zur jeweiligen Woche angegebenen Anforderungen.

Berechnung der NCSS pro Anforderung

Daher ist es auch möglich, den Quotienten aus NCSS und der Anzahl der Anforderungen zu berechnen. Dieses Maß gibt an, wieviele Code-Statements durchschnittlich zur Implementierung einer Anforderung nötig waren.

Im dargestellten Fall wurde der Gesamt-Code-Umfang zur Berechnung dieses Quotienten verwendet und nicht nur der zur eigentlichen Anwendung gehörende Sourcecode. Denn insbesondere bei der agilen Softwareentwicklung hängen der Produktiv- und der Test-Code stark voneinander ab. Da das Design der Anwendung nicht bis ins kleinste Detail spezifiziert und modelliert wird, ergeben sich viele Programmier-Details erst durch das Wechselspiel von Codierung und Unit-Tests. Beispielsweise wird die Notwendigkeit einzelner Code-Elemente oftmals erst in Folge eines fehlgeschlagenen Unit-Tests erkannt. Somit ist es gerechtfertigt, den Gesamt-Code-Umfang in Beziehung zu den damit implementierten Anforderungen zu setzen.

Stabilisierung des untersuchten Quotienten

Die Entwicklung des Quotienten aus den NCSS und der Anzahl der Anforderungen ist in Abbildung 6.3 dargestellt. Im dargestellten Projekt hat sich die Anzahl der Code-Statements, welche zur Implementierung einer einzelnen Anforderung notwendig sind, bei ca. 21,5 Statements eingependelt. Der Quotient war zu Beginn des Projektes deutlich höher, da hier zuerst einmal das Grundgerüst der Anwendung inklusive der Anbindung an die Datenbank erstellt wurde. Anschließend wurden im Rahmen des agil durchgeführten Entwicklungsprozesses kontiniuerlich die in der jeweiligen Woche neu hinzugekommenen Anforderungen implementiert.

Das dargestellte Projekt ist zwar zu klein, um wirklich statistisch fundierte Aussagen über die Anzahl der Code-Statements zu machen, welche für die Implementierung einer einzelnen Anforderung notwendig sind. Außerdem würde in der betrieblichen Praxis viel mehr der zugehörige zeitliche Aufwand interessieren, der dann entsprechend verrechnet werden müsste. Dennoch zeigt dieses Beipiel bereits die grundsätzlichen Anwendungsmöglichkeiten des innerhalb dieser Arbeit dargestellten Software-Messungs-Frameworks.

Die oben dargestellten Messungen können durch das MMF automatisiert und ohne zusätzlichen Aufwand für die Projektbeteiligten durchgeführt werden. In diesem Beispiel wurde nur auf die Software-Quelltexte im Versionsverwaltungssystem und die Anforderungen im Issue-Tracking-System zugegriffen und deren Umfang gemessen. Wenn während des Projekts noch zusätzlich Kontextinformationen wie beispielsweise die *Artefakt-Funktionalität* (vgl. Abschnitt 4.2.2) gepflegt worden wären, hätte man die Umfangsmaße auch daran anknüpfen können, um eine feingranularere Auswertung zu erhalten.

Grundsätzlich wäre es für ein Unternehmen von großem Wert, wenn man aus der Anzahl der Anforderungen zumindest grob den Code-Umfang oder sogar den zu

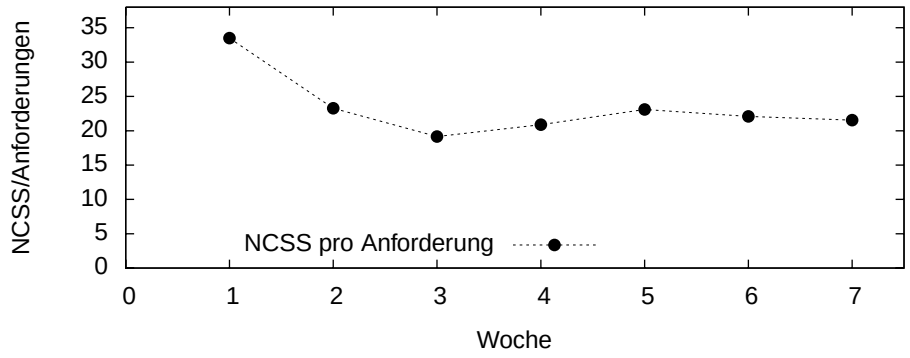


Abbildung 6.3: Entwicklung des Quotienten aus NCSS und Anforderungen

Anzahl der Anforderungen als Indikator für den Projektumfang

erwarteten Aufwand ableiten könnte. Eine solche Möglichkeit wäre sowohl bei der Angebotserstellung als auch bei der Bepreisung von Change-Requests von Vorteil.

In der Praxis ist nicht zu erwarten, dass sich ein exakter Wert von 21,5 NCSS pro Anforderung wie in obigen Beispiel einstellt, der dann auch noch für alle Software-Domänen gültig ist. Dass der betrachtete Quotient im obigen Beispiel so schnell konvergierte ist evtl. durch den Umstand zu erklären, dass nur ein einziger Entwickler an diesem Projekt beteiligt war, der seinen persönlichen Programmierstil anwendet.

Abhängigkeit von der Artefakt-Funktionalität

Bei der Analyse größerer Projekte sollte daher die Untersuchung des Verhältnisses zwischen Code-Umfang und der Anzahl der Anforderungen in Abhängigkeit von der jeweiligen Artfakt-Funktionalität, wie beipielsweise

- Anwendungslogik,
- Präsentationsschicht oder
- Datenhaltung,

vorgenommen werden. Insbesondere im Kontext der Präsentationsschicht könnte sich dabei zeigen, dass sich keine konkrete Beziehung zwischen den beiden Maßen nachweisen lässt, weil bei der Implementierung der Präsentationsschicht in der Regel GUI-Design-Werkzeuge eingesetzt werden, welche den Code entsprechend generieren. Daher sollte zur Abschätzung des Aufwandes für die GUI-Implementierung auf andere Verfahren wie die Function-Point-Methode zurückgegriffen werden, welche konkrete Zählregeln für GUI-Elemente beinhaltet (vgl. [BF00], S. 223 ff.).

Im Gegensatz dazu ist beim Sourcecode, der die Anwendungslogik betrifft, eher zu erwarten, dass sich eine Beziehung zwischen der Anzahl der Anforderungen und dem Code-Umfang erkennen lässt. Auch das oben dargestellte Projekt zur Implementierung eines Zeiterfassungssystems war, nachdem die Datenbank-Anbindung grundsätzlich lauffähig war, auf die Implementierung der Anwendungslogik fokusiert.

Notwendigkeit langfristiger Untersuchungen

Allerdings wird man in einem Softwareunternehmen erst nach mehreren durchgeführten und vermessenen Softwareprojekten entsprechende Trends bzgl. der hier betrachteten Beziehung erkennen können. Aber auch die Möglichkeit, eine Spann-

weite angeben zu können, innerhalb derer sich der Code-Umfang bzw. der Aufwand für die Implementierung einer Anforderung bewegen, ist für die Projektplanung von hohem Wert.

6.1.2 Monitoring zweier Programmier-Teams

Bei einem weiteren studentischen Softwareentwicklungsprojekt wurde die Performance zweiter Entwicklungsteams verglichen. Dabei wurde insbesondere verglichen, wie sich der Umfang des durch die beiden Teams erzeugten Sourcecodes auf die Artefakt-Funktionalitäten *Anwendungslogik* und *Visualisierung* verteilte.

Implementierung eines Strategiespiels

Dieses Softwareentwicklungsprojekt war Inhalt eines Software-Praktikums, welches die Studenten im Rahmen der Informatik-Ausbildung an der *Universität Bayreuth* durchführen mussten. Die Studenten bildeten dabei zwei Teams, welche jeweils einen „intelligenten“ Client für das Strategiespiel *Mancala*[2] (Bohnenspiel) entwickeln mussten. Bei diesem Spiel müssen zwei konkurrierende Spieler eine Menge von Spielsteinen (i. d. R. Bohnen), welche sich in verschiedenen Mulden des Spielbrettes befinden, nach bestimmten Regeln umverteilen. Bei jedem Zug wählt ein Spieler eine zu leerende Bohnenmulde derart, dass bei der anschließenden Umverteilung möglichst viele Bohnen in der eigenen Gewinn-Mulde landen.

Ein sogenannter Mancala-Server, welcher die aktuelle Konfiguration des Spiels verwaltet und Methoden zum Umverteilen der Bohnen bereitstellt, wurde den Studenten zur Verfügung gestellt.

Anforderungen an die Anwendung

Die Aufgabe der Studenten war, eine Mancala-Client-Anwendung zu entwickeln, welche, die folgenden Aufgaben erfüllen sollte:

- Über eine graphische Benutzeroberfläche soll der aktuelle Zustand des Spieles visualisiert werden. Insbesondere soll die Verteilung der Bohnen auf die einzelnen Mulden der beiden Spieler dargestellt werden.

- Der Mancala-Client soll einem menschlichen Spieler eine Auswahlmöglichkeit für die nächste zu leerende Bohnen-Mulde zur Verfügung stellen.

- Der Mancala-Client soll selbständig eine Spielstrategie entwickeln und daraus den nächsten Spielzug ableiten können. Insbesondere soll er aufgrund des aktuellen Spielzustandes eine Mulde auswählen, deren Bohnen gemäß der Spielregeln so auf die anderen Mulden verteilt werden, dass für den gegnerischen Spieler zum Ende des Spiels möglichst wenige Bohnen übrig bleiben.
  Auf diese Weise kann ein Mancala-Client entweder gegen einen menschlichen Spieler oder gegen einen zweiten Mancala-Client spielen.

Die Benutzeroberfläche sollte basierend auf dem *Model-View-Controller*-Design-Pattern (vgl. [FF04]) umgesetzt werden. Dazu sollte der Mancala-Client vom Server informiert werden, wenn sich der Spielzustand geändert hat. Der Mancala-Client

[2] `http://de.wikipedia.org/wiki/Mancala`

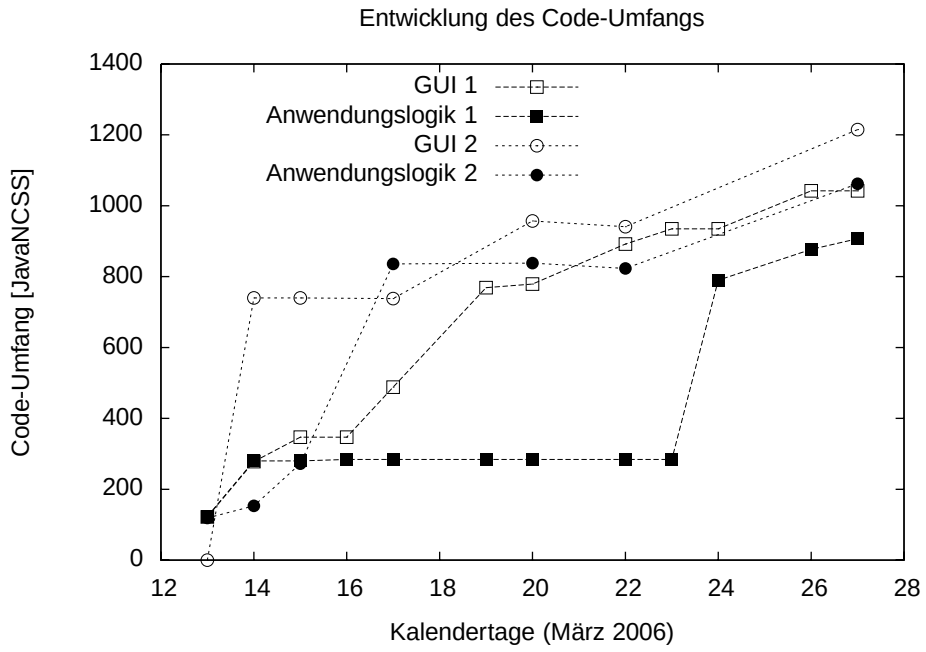


Abbildung 6.4: Vergleich des Entwicklungs-Outputs zweier Teams

fragt dann vom Server die aktuelle Verteilung der Bohnen ab und stellt diese graphisch auf dem Bildschirm dar.

Künstliche Intelligenz und Spielbäume

Für die Berechnung des nächsten günstigen Spielzuges musste eine Spielstrategie implementiert werden. Dazu wird auf Basis des aktuellen Spielzustands und der nun möglichen eigenen und gegnerischen Züge ein *Spielbaum* (vgl. [Gib06]) aufgebaut. Die Knoten des Spielbaumes repräsentieren mögliche Spielstellungen, welche dann entsprechend bewertet werden. Aus der Bewertung dieser erreichbaren Spielstellungen folgt die Entscheidung über den nächsten Zug.

Angabe der Kontext-informationen bei der Versionierung

Während der Projektdurchführung wurde mittels des MMF die Entwicklung des Code-Umfangs gemessen. Die Teams mussten, wenn neu erstellter oder veränderter Sourcecode im Versionsverwaltungssystem versioniert wurde, bei den Commit-Informationen die Artefakt-Funktionalität der zu versionierenden Code-Änderung angeben. Diese beschränkten sich, da der Mancala-Server bereits vorhanden war, auf die Funktionalitäten

- Visualisierung (Graphische Benutzeroberfläche) und
- Anwendungslogik.

Diese Artefakt-Funktionalitäten wurden dann auch als Anknüpfungspunkte für die Umfangsmessung verwendet. Damit konnte bei jedem Commit-Vorgang festgestellt werden, wie sich der Umfang des erstellten Sourcecodes auf die beiden Funktionen verteilte. Der Verlauf der Entwicklung dieser Code-Umfänge ist in Abbildung 6.4 dargestellt.

Hier zeigt sich, dass das Team 1 anfangs seine Programmieraktivitäten auf die Entwicklung der graphischen Benutzeroberfläche konzentriert hat. Hier steigt der Sourcecode-Umfang während der gesamten Entwicklungszeit nahezu linear an. Im Gegensatz dazu lassen sich bei Team 1 im Bereich der Anwendungslogik bis zum elften Projekttag (23. März) keine Code-Zuwächse erkennen.

Unterschiedliche Implementierungsstrategien

Bei Team 2 sind dagegen während der gesamten Projektphase sowohl im Bereich der Anwendungslogik als auch im Bereich der Benutzeroberfläche kontinuierliche Weiterentwicklungen des Sourcecodes erkennbar. Insbesondere der Umfang derjenigen Sourcecode-Komponenten, welche der Anwendungslogik zuzuordnen sind, ist bereits ab dem zweiten Projekttag auf einem hohen Niveau. Zusätzlich scheinen diese Code-Komponenten auch beständig weiterentwickelt worden zu sein, da sich ihr Umfang mit jedem Commit-Vorgang etwas geändert hat.

Bestätigung der Messergebnisse

Diese durch Softwaremessung gewonnenen Erkenntnisse haben sich auch bei der praktischen Prüfung der durch die beiden Team erstellten Anwendungen bestätigt. So zeichnete sich der Mancala-Client von Team 1 durch eine optisch ansprechende graphische Benutzeroberfläche aus, die mehrere durch den Anwender auswählbare *Skins* besaß. Dagegen erwies sich der Mancala-Client von Team 2 in Bezug auf die Spielstärke als bedeutend leistungsfähiger. Insbesondere in der Konfiguration, bei der beide Mancala-Clients bedienerlos gegeneinander spielen, hat der Mancala-Client von Team 2 grundsätzlich gewonnen.

Auch dieses Softwareprojekt war relativ klein, so dass es schwierig ist, die bei der Softwaremessung gewonnenen Erkenntnisse zu verallgemeinern. Da es sich um das erste Softwareentwicklungsprojekt handelte, welches von den Studenten gemeinsam als Programmierteam zu bearbeiten war und bei dem ein Versionsverwaltungssystem eingesetzt werden sollte, könnten auch gewisse Ungenauigkeiten bei der Kennzeichnung der Commit-Informationen vorliegen. Allerdings kann auch in der betrieblichen Praxis nicht immer garantiert werden, dass derartige Verwaltungsinformationen von allen Mitarbeitern mit der gebotenen Sorgfalt gepflegt werden.

Erkennen von Fehlentwicklungen

Insgesamt kann jedoch festgehalten werden, dass es mittels des dargestellten und relativ einfachen Messverfahrens grundsätzlich möglich ist, Fehlentwicklungen im Laufe der Projektentwicklung zu erkennen. So ist es auch im professionellen Umfeld denkbar, dass sich Softwareentwickler im Rahmen eines Projektes zunächst derjenigen Domäne zuwenden, mit der sie mehr Erfahrung haben oder im Rahmen derer sie weniger Probleme erwarten, so dass andere Funktionsbereiche der zu entwickelnden Anwendung vernachlässigt werden.

6.2 Einsatzmöglichkeiten des Maven Measurement Frameworks

6.2.1 Integrierte Softwaremessung

Eines der Hauptanliegen dieser Arbeit war es, einen leichtgewichtigen Ansatz zur Softwaremessung darzustellen, welcher sowohl eine Werkzeugunterstützung als auch

eine Beschreibung der Vorgehensweise beinhalten sollte (vgl. [DHW06b]).

KMU als Zielgruppe dieses Ansatzes

Dabei sollte die Werkzeugunterstützung für die Softwaremessung möglichst einfach in die Softwareentwicklungsumgebung integrierbar sein. Auf Grund der Gesamtkonzeption dieses Ansatzes erscheint er insbesondere für kleine und mittlere Unternehmen (KMU) in der Softwareindustrie geeignet. In derartigen Unternehmen ist auch die Bereitschaft, *Open-Source-Software* zur Software-Entwicklung einzusetzen, recht groß. Dies liegt daran, dass das Budget für IT-Investitionen bei den KMU in der Regel sehr beschränkt ist und diese daher eher geneigt sind, kostenlose Open-Source-Entwicklungswerkzeuge wie das Versionsverwaltungssystem *Subversion* einzusetzen.

Umgekehrt ist man bei größeren Unternehmen eher geneigt, kommerzielle Produkte einzusetzen. Diese sind zwar mit hohen Lizenzkosten verbunden, jedoch besteht bei Problemen (vermeintlich) die Möglichkeit, sich an den Hersteller zu wenden.

Zunehmende Verbreitung von Maven

So ist es nicht unwahrscheinlich, dass eine Organisation, welche den in dieser Arbeit beschriebenen Ansatz zur Softwaremessung umsetzen möchte, ohnehin bereits *Maven* als Build-Werkzeug einsetzt. Auch die Verwaltung der von der Organisation selbst erstellten Programm-Bibliotheken, welche auch in anderen Projekten wieder verwendet werden sollen, wird durch Maven stark vereinfacht.

In Abbildung 6.5 ist eine mögliche Integration des *Maven Measurement Frameworks* in die Werkzeuglandschaft eines Softwareentwicklungsunternehmens dargestellt.

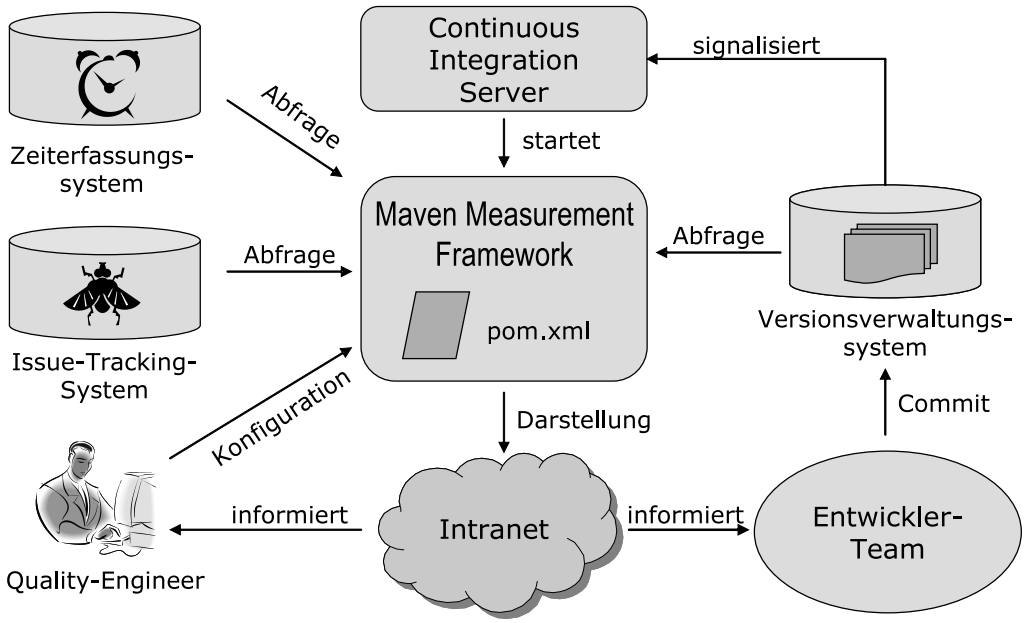


Abbildung 6.5: Integrierte Softwaremessung (vgl. [DHW06b])

Wie im Kapitel 4 dargestellt, stammen die Informationen, welche im Rahmen der Softwaremessung analysiert werden, aus folgenden Sub-Systemen der Entwicklungsumgebung:

Versionsverwaltungssystem
: Das Versionsverwaltungssystem enthält die verschiedenen Versionen der im Rahmen des Projektes bearbeiteten Sourcecode-Elemente. Zusätzlich enthalten die Commit-Informationen Angaben über die Projekt-Aktivitäten, innerhalb derer die Modifikationen durchgeführt wurden, und die Artefakt-Funktionen, welche die modifizierten Sourcecode-Elemente erfüllen.

Zeiterfassungssystem
: Auch die Datensätze im Zeiterfassungssystem enthalten Informationen über die durchgeführten Aktivitäten und die dabei implementierten Artefakt-Funktionalitäten.

Issue-Tracking-System
: Im Issue-Tracking-System können neben Fehlermeldungen auch Änderungs- und weitere Kunden-Anforderungen verwaltet werden. Dadurch besitzen diese Elemente eine eindeutige ID, welche in den Commit-Informationen und in den Zeiterfassungsdatensätzen referenziert werden können. Somit ist es auch möglich, den Zeitaufwand oder den Code-Umfang, den eine Fehlerbehebung verursacht, zu bestimmen.

Konfiguration der Softwaremessung durch Quality-Engineer

Der Projektleiter oder ein Qualitätsbeauftragter des Projekts, welche beide im Weiteren als *Quality-Engineer* bezeichnet werden sollen, können die für sie interessanten und zu erhebenden Softwaremaße konfigurieren. Dabei kann der Quality-Engineer sowohl auf im Unternehmensrepository abgelegte Standard-Messwertgeber zurückgreifen als auch mittels *Groovy* eigene Messwertgeber implementieren (vgl. Abschnitt 5.4.2, S. 148 ff.). Die Konfiguration der Messwertgeber und damit die Definition der zu erhebenden Softwaremaße erfolgt in der POM-Datei des Projektes zusammen mit anderen projektrelevaten Informationen (vgl. Abschnitt 5.1.2.1, S. 121 ff.).

Falls ein *Continuous-Integration-System* (vgl. S. 128) eingesetzt wird, bietet es sich an, darüber auch die Erhebung der Softwaremaße zu starten. Typischerweise ist ein Continuous-Integration-Server so konfiguriert, dass er durch einen das Projekt betreffenden Commit im Versionsverwaltungssystem getriggert wird, das Projekt neu zu übersetzen und alle Tests durchzuführen. Daran anschließend kann der Continuous-Integration-Server auch die Softwaremessungen durch das MMF starten.

Das MMF führt die Softwaremessungen durch, indem es aus den oben genannten Sub-Systemen der Entwicklungsumgebung die relevanten Informationen abruft. Die Messergebnisse selbst können graphisch aufbereitet[3] und im Intranet der Organisation veröffentlicht werden.

[3] Die Mess-Diagramme in diesem Kapitel wurden beispielsweise mit dem Funktionenplotter *Gnuplot* (`http://www.gnuplot.info`) erstellt.

6.2.2 Persistierung der Messergebnisse

Das Maven-Measurement-Framework ist als leichtgewichtiger Ansatz konzipiert, um grundsätzlich zunächst einmal den Projektmitarbeitern ein Werkzeug an die Hand zu geben, mit dem der aktuelle Projektverlauf analysiert werden und evtl. Schwachstellen aufgedeckt werden können (vgl. Abschnitt 6.1.2).

Eine objektive Beurteilung des aktuellen Projektes ist jedoch nur auf Basis eines Erfahrungsschatzes bezüglich einer genügend großen Menge vergleichbarer Projekte möglich. Dieser ist auch notwendig, um die noch verbleibenden Phasen eines aktuellen Projektes oder auch zukünftige Projekte prognostizieren zu können. Auch um die Auswirkungen von (Prozess-)Verbesserungsbemühungen beurteilen zu können, muss auf entsprechende Vergleichswerte zugegriffen werden können.

Somit ist es grundsätzlich wünschenswert, die ermittelten Softwaremaße nicht nur aktuell mittels eines Intranet-Dashboards zu visualisieren, sondern sie dauerhaft in einer Datenbank abzulegen. Damit wäre es möglich, die abgeschlossenen Projekte später bzgl. Fragestellungen auszuwerten, die aktuell noch gar nicht bekannt sind bzw. für deren Beantwortung ein hinreichend großer Datenbestand notwendig ist.

Eine derartige Fragestellung ist beispielsweise die Frage nach der optimalen Iterationsgröße bei iterativen Projekten. Hierzu könnte untersucht werden, in wie fern Softwaremaße wie

- die Produktivität (vgl. Abschnitt 2.3.3),
- die Fehlerrate (vgl. Abschnitt 2.3.5.2) oder
- die Termintreue (vgl. Abschnitt 2.3.4.1)

von der (durchschnittlichen) Länge der Iterationen abhängen. Falls es derartige Korrelationen gibt, könnte man untersuchen, ob eher kleinere oder größere Iterationen für den Gesamtprojektverlauf günstiger sind.

Allerdings war es nicht Ziel dieser Arbeit, ein Konzept für ein derartiges Data-Warehouse zu konzipieren. Mit Hinblick auf die Darstellungen im Abschnitt 2.2.1.1 (GQM-Methode, S. 22 ff.) und im Abschnitt 2.2.3 (QBench-Ansatz, S. 29 ff.) erscheint es fraglich, ob es zielführend ist, alle erhebbaren Maße zunächst einmal automatisiert ermitteln zu lassen und in einem Data-Warehouse abzulegen.

Case-based Reasoning

Die Verwaltung von Wissen mit dem Ziel, Probleme durch Analogieschluss zu lösen, d. h. zur Lösung eines gegebenen Problems (z. B. Aufwandsschätzung) auf die Erfahrungen mit einem ähnlichen und früher bereits gelösten Problem zurückzugreifen, gehört zur Theorie des *Case-based Reasonings* (Fallbasiertes Schließen, vgl. [AP94]).

Die Anwendung des Case-Based Reasonings im Rahmen des Software-Engineerings (vgl. [She02]) führt zur Implementierung sogenannter *Experience Bases* als Teil einer technischen und organisatorischen Infrastruktur, welche als *Experience Factory* bezeichnet wird (vgl. [ADH+01]).

Experience Factory

Die Bezeichnung *Experience Factory* geht zurück auf Victor Basili, Gianluigi Caldiera und Dieter Rombach (vgl. [BCR94]). Bei diesem Ansatz wird davon ausgegangen, dass es zur Verbesserung von Softwareprozessen nötig ist, kontinuierlich evaluierte Projekt-Erfahrungen in einem als *Experience Base* bezeichnetem Repo-

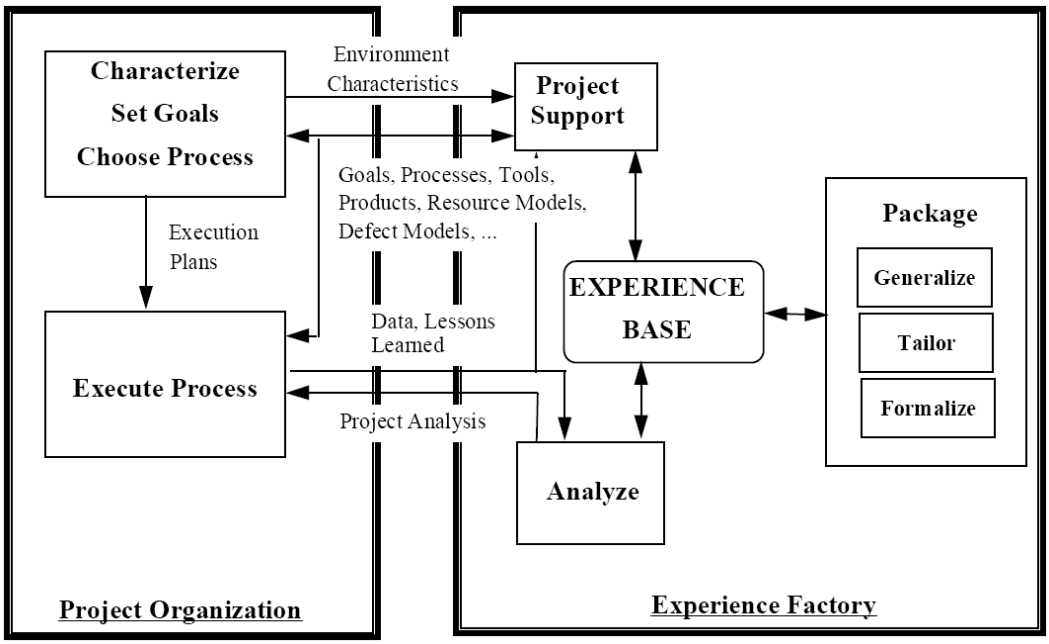


Abbildung 6.6: Experience Factory (aus [BCR94])

sitory zu sammeln. Das systematische Sammeln dieser wiederverwertbaren Erfahrungen, soll von einer als *Experience Factory* bezeichneten Organisationsstruktur durchgeführt werden, welche von den einzelnen Projekt-Teams logisch getrennt ist (vgl. [BCR94]). Dabei soll die Speicherung der gesammelten Projektinformationen nicht automatisiert erfolgen, sondern die Erfahrungen sollen einzeln validiert und zu *Erfahrungspaketen* gebündelt werden (vgl. Abbildung 6.6).

Post-Mortem-Analyse mit dem MMF

Das Maven-Measurement-Framework kann sicherlich im Rahmen einer solchen Experience-Factory Verwendung finden. Insbesondere ist es auch nach dem Projektende jederzeit möglich, mittels des MMF zusätzliche Analysen durchzuführen. Das MMF persisistert zwar selbst keine Daten, greift aber ausschließlich auf Daten zu (Versionsverwaltung, Zeiterfassung, Issue-Tracking), welche bereits in persistierter Form vorliegen.

Die Fragen, welche Softwaremaße jedoch innerhalb einer Experience-Factory erhoben werden sollen und wie daraus Analyse- und Bewertungsmethoden für zukünftige Projekte abgeleitet werden können, lassen Raum für weitere Forschungsarbeiten.

6.3 Grenzen des beschriebenen Ansatzes

Bei der Betrachtung der Grenzen des in dieser Arbeit beschriebenen Ansatzes zur Softwaremessung muss zwischen

- der Vorgehensweise, die Softwaremaße an bestimmte Anknüpfungspunkte im Rahmen des Softwareentwicklungsprozesses zu binden, und

- der Umsetzung mittels des Maven-Measurement-Frameworks

unterschieden werden.

Universeller Ansatz

Die Vorgehensweise selbst ist zunächst einmal universell einsetzbar, da letztendlich nur gefordert wird, dass die Berechnung der interessierenden Softwaremaße auf einen bestimmten Kontext eingeschränkt erfolgen soll. Für die Berechnung der Softwaremaße selbst ist es jedoch erforderlich, dass die zu messenden Entitäten (z. B. Commits im Versionsverwaltungssystem, Zeiterfassungsdatensätze) mit den entsprechenden Kontextinformationen versehen wurden. Nur so können die für die Berechnung der Softwaremaße relevanten, d. h. zu einem bestimmten Kontext gehörenden Entitäten maschinell identifiziert und vermessen werden. Das Vorhandensein einer entsprechenden Infrastruktur, bestehend aus Versionsverwaltungs-, Zeiterfassungs- und Issue-Tracking-System, wird an dieser Stelle vorausgesetzt.

MMF-Konfiguration baut auf Build-Konfiguration auf

Die Umsetzung des Ansatzes in Form des Maven-Measurement-Frameworks orientiert sich dagegen an einer für die Java-Programmierung typischen Entwicklungsumgebung. Eine wichtige Motivation für die Verwendung von Maven als Basis für ein Softwaremessungswerkzeug war beispielsweise, dass in dem von Maven verwendeten *Project Object Model* bereits viele wichtige Projektinformationen (Versionsverwaltungssystem, Entwickler) ohnehin hinterlegt sind und entsprechend abgerufen werden können.

Vermessung von C++-Projekten grundsätzlich möglich

Für Softwareprojekte, welche auf anderen Programmiersprachen basieren, wird Maven in der Regel nicht als Build-Management-Software eingesetzt. Für *C++* Projekte gibt es zwar eine rudimentäre Unterstützung[4], jedoch wird diese seit 2006 nicht mehr weiterentwickelt. So entfällt bei anderen Programmiersprachen der positive Nebeneffekt, dass die Konfiguration des Build-Tools bereits einen Teil der Konfiguration des Messwerkzeugs enthält. Trotzdem könnte das MMF auf ein Versionsverwaltungssystem zugreifen, welches C++-Code enthält.

Hinzu kommt, dass das MMF intensiven Gebrauch von denjenigen Funktionalitäten macht, welche über das Maven-API bzw. die zahlreichen Maven-Plugins zur Verfügung gestellt werden. Dazu zählt beispielsweise das Starten und die Analyse von Unit-Tests oder die Ermittlung des Code-Umfangs mittels des JavaNCSS-Plugins. Diese Funktionalitäten müssten für andere Programmiersprachen erst Nachimplementiert werden.

MMF-Einsatz im Rahmen von ERP-Softwareentwicklung

Sehr unbefriedigend erscheinen dagegen die Anwendungsmöglichkeiten des MMF, wenn die Softwareentwicklung auf Großrechner-Systemen oder anderen proprietären Entwicklungsplattformen wie SAP R/3 stattfindet.

Beispielsweise gibt es auf der Großrechner-Betriebssystemplattform *BS2000*[5] von Fujitsu-Siemens kein Versionsverwaltungssystem, welches mit *Subversion* oder *CVS* vergleichbar wäre. Stattdessen werden unter BS2000 Sourcecode-Dateien als Elemente von Programmbibliotheken unter Verwendung des Bibliotheksverwaltungssystems LMS (*Library Maintenance System*, vgl. [Fuj05]) verwaltet. Ein „Checkout“ auf Bibliotheksebene ist dabei nicht möglich. Stattdessen ist die Verwaltung einzelner Ver-

[4] `http://mojo.codehaus.org/maven-native/native-maven-plugin`
[5] `http://www.fujitsu-siemens.de/products/bs2000`

sionen zusammen mit einem sogenannten „Ausleihverfahren“ nur auf Elementebene vorgesehen.

Versionsverwaltung unter SAP R/3

Innerhalb eines SAP R/3 Systems wird die Rolle des Software-Repositorys von dem zum R/3 System gehörenden relationalen Datenbanksystem eingenommen. In diesem Datenbanksystem sind somit nicht nur die betriebswirtschaftlichen Daten des R/3 Systems, sondern auch Programme, Eingabemasken, Tabellendefinitionen und Datentypen abgelegt. Um unabhängig vom verwendeten Datenbanksystem auf diese Daten zugreifen zu können, wird mit dem *Data Dictionary* eine allgemeine und systemunabhängige Schnittstelle zu diesem Datenbestand bereitgestellt (vgl. [Mat02]).

Neben dem Data Dictionary hat bei der Entwicklung unter SAP R/3 auch das Korrektur- und Transportsystem, welches in der Anwendung *Transport Organizer* zusammengefasst ist, eine zentrale Bedeutung. Zusätzlich zur Versionsverwaltung stellt der Transport Organizer Möglichkeiten zur Verfügung, Programmänderungen auf andere SAP Systeme und insbesondere vom Testsystem auf das Produktivsystem zu übertragen (vgl. [DH04]).

Beiden Systemen BS2000 und SAP R/3 ist somit gemein, dass man einerseits von außen kaum auf das Versionsverwaltungssystem zugreifen kann und dieses auch anders organisiert ist, als es in Java-Umgebungen üblich ist. In beiden Fällen würde es sich somit eher anbieten, ein benötigtes Softwaremessungs-Werkzeug nativ für die jeweilige Plattform zu entwickeln. Dies erscheint insbesondere im SAP R/3 Umfeld interessant, da sich dort zusammen mit dem Zeiterfassungs- und dem Issue-Tracking-System alle für die Softwaremessung relevanten Sub-Systeme auf einer Plattform integrieren lassen.

7 Zusammenfassung und Ausblick

> Not everything that counts can be counted, and not everything that can be counted counts.
>
> *(Albert Einstein)*

Softwaremetrie ist der Versuch, Softwareentwicklungsprozesse kontrollierbar zu machen und Möglichkeiten aufzuzeigen, die Durchführung von Softwareprojekten zu beurteilen. Dies beinhaltet sowohl die Beurteilung des Entwicklungsprozesses selbst als auch dessen Ergebnisse.

Scheitern von Softwaremessungsvorhaben

Allerdings scheitern viele Unternehmen bei der Implementierung der für sie passenden Softwaremetrieprozesse (vgl. [HMW06], [EDBSt05]). Ebert et al. beschreiben in [EDBSt05] einige der Hauptrisiken bei der Umsetzung von Softwaremessungsvorhaben.

So würden Softwaremaße erhoben werden, für die es keine konkrete Verwendung gibt, da die Ziele der Softwaremessung nicht festgelegt wurden. Oder es würden von der Geschäftsführung Ziele festgelegt werden, welche entweder absolut unrealistisch sind, oder zumindest auf dem aktuellen Reifegrad des Unternehmens kaum umgesetzt werden können (vgl. [EDBSt05], S. 3).

Trudel und Tardif sehen insbesondere für kleinere und mittlere Unternehmen als weiteres Problem, dass diese im Gegensatz zu größeren Organisationen kein dediziertes Personal im Bereich der Softwaremessung haben. Stattdessen wird dort dem Projektleiter oder einzelnen Team-Mitarbeitern die Aufgabe zugeteilt, ein Softwaremessungsprogramm aufzusetzen und zu koordinieren. Leider wird eine derartige Vorgehensweise oft durch zu aggressive Projekttermine zunichte gemacht (vgl. [TT06]).

Erfolgsfaktoren

Umgekehrt werden

- eine tiefe Integration der Softwaremessung in den Softwareentwicklungsprozess,
- die Beteiligung der Mitarbeiter an der Definition der Messstrategien und
- die (anfängliche) Konzentration auf wenige einleuchtende Softwaremaße

als Erfolgsfaktoren für Softwaremetrieprojekte genannt (vgl. [EDBSt05], [TT06], [KO06]).

7.1 Zielsetzung dieser Arbeit

Das Ziel der vorliegenden Arbeit ist es, einen werkzeugunterstützten Ansatz zur Softwaremessung darzustellen und prototypisch zu implementieren. Wie bereits in

Anforderungen

Kapitel 1 dargestellt, gehört zu den wesentlichen Anforderungen an das in dieser Arbeit entwickelte Softwaremessungs-Framework

- die weitestmögliche Automatisierung des Messprozesses,
- die Standardisierung der Messdurchführung, um
- die Messergebnisse über Projektgrenzen hinweg vergleichen zu können, und
- eine einfache Integrierbarkeit des Ansatzes in bestehende Umgebungen.

Da die Erhebung von Softwaremaßen grundsätzlich zielorientiert durchgeführt werden soll, wurden im Kapitel 2 verschiedene Ansätze zur Definition von Strategien zur Softwaremessung dargestellt. Dabei wurde festgestellt, dass einige Softwaremaße, welche in dieser Arbeit als Kernattribute bezeichnet werden, grundsätzlich erfasst werden sollten. Diese fünf Messgrößen

- Umfang,
- Aufwand,
- Qualität,
- kalendermäßige Zeitdauer und
- Produktivität

bilden einen sinnvollen Grundstock bei der Einführung von Softwaremetrieprozessen innerhalb einer Organisation.

Ansatz dieser Arbeit

Als entscheidende Faktoren bei der Umsetzung dieses Ansatzes wurden die Konzentration auf diese wenigen, auf Managementebene relevanten, Softwaremaße und die Anknüpfung der Softwaremaße an Kontextinformationen, welche bereits auf Prozessebene existieren, herausgearbeitet. Dadurch ist es möglich, die zu erhebenden Softwaremaße unabhängig von einem konkreten Projekt zu definieren, so dass diese bei verschiedenen Projekten in konsistenter, einheitlicher Weise verwendet werden können.

Allerdings setzt diese Vorgehensweise voraus, dass die zu messenden Entitäten des Softwareentwicklungsprojektes mit entsprechenden Kontextinformationen versehen sind. Daher wird in dieser Arbeit vorgeschlagen, insbesondere die Commit-Informationen im Versionsverwaltungssystem und die Arbeitszeitdatensätze im Zeiterfassungssystem mit den relevanten Kontextinformationen zu versehen. Als besonders bedeutend für die spätere softwaremetrische Auswertung wurden dabei die Kontextinformationen

- Projektaktivität und
- Artefakt-Funktionalität

identifziert.

In Kapitel 3 wurden verschiedene Typen von Messwerkzeugen voneinander abgegrenzt. Dabei wurde schließlich das innerhalb dieser Arbeit entwickelte Messwerkzeug als *Kommandozeilen-Leitstand* charakterisiert. Ein Projektleitstand zeichnet sich gemäß den Ausführungen in Kapitel 3 dadurch aus, dass im Gegensatz zu einfachen Dashboards auch zur Laufzeit der Anwendung zu berechnende und zu visualisierende Softwaremaße neu definiert werden können. Zusätzlich ist es ein wesentliches Merkmal dieser Werkzeugimplementierung, dass durch die Verwendung der Projektverwaltungssoftware *Maven* als Basis-Framework die Softwaremessung

in den Entwicklungsprozess integriert wird. Dadurch sinken insbesondere in den Organisationen, in denen Maven ohnehin bereits eingesetzt wird, die Hürden für die Einführung der Softwaremessung.

Auch bietet die dargestellte Implementierung eine sehr große Flexibilität bei der Definition von Softwaremaßen. Insbesondere durch die Verwendung der Skriptsprache *Groovy* ist es möglich, Softwaremaße auch zur Laufzeit des Projektes neu zu definieren. Die skriptbasierte Definition von Softwaremaßen mag zwar auf den ersten Blick weniger komfortabel als die Verwendung einer graphischen Benutzeroberfläche erscheinen, jedoch lassen sich insbesondere komplexere Maße leichter mittels einer Formel als durch eine Kombination graphischer Symbole darstellen.

Beitrag dieser Arbeit

Weder die Erfassung von Zeitaufwänden im Rahmen von Softwareprojekten noch die Attributierung von Programmierartefakten mit Informationen über die implementierten Funktionalitäten sind neue Ansätze. Neu an dem in dieser Arbeit dargestellten Ansatz ist die integrierte Betrachtung derartiger Informationen im Hinblick auf die automatisierte Auswertbarkeit im Rahmen der Softwaremessung.

Durch die Verfügbarkeit dieser Kontextinformationen ist es beispielsweise möglich, automatisiert den Aufwand für einzelne Komponenten des erstellten Systems zu ermitteln. Oder es kann festgestellt werden, wie sich die Umfänge der erstellten Softwarekomponenten auf die einzelnen Phasen des Projektzyklus verteilen.

Für die Verwaltung dieser Kontextinformationen muss – hauptsächlich durch die Softwareentwickler – ein gewisser Mehraufwand erbracht werden, da beispielsweise die Zeiterfassungsdatensätze und die Commit-Informationen um die entsprechenden Angaben ergänzt werden müssen. Dies ist jedoch eine allgemein akzeptierte Vorgehensweise (vgl. [Sel05]). Insbesondere ist dieser Mehraufwand gering im Verhältnis zu den dadurch gewonnenen Auswertungsmöglichkeiten.

Tätigkeitsprotokollierung

Grundsätzlich gibt es mit dem Werkzeug *Hackystat* (vgl. [Joh01]) einen Ansatz, jede einzelne Tätigkeit der beteiligten Softwareentwickler mitzuprotokollieren. Dabei wird über entsprechende Sensoren neben den Programmiertätigkeiten, der Ausführung von Unit-Tests und dem Zugriff auf das Versionsverwaltungssystem z. B. auch die Verwendung des E-Mail-Programmes protokolliert (vgl. [JKA+04], [JKP+05]). Allerdings kann dieses Werkzeug nur die reinen Tätigkeiten des Entwicklers erfassen und kann nicht erkennen, welchem Zweck beispielsweise die aktuell bearbeitete Datei dient.

In der vorliegenden Arbeit wird dagegen der Ansatz vertreten, dass es trotz des beschriebenen notwendigen Mehraufwandes sinnvoller ist, weniger Maße zu erfassen, diese jedoch zusammen mit ihrem Kontext zu betrachten.

7.2 Ausblick

Wie bereits im Kapitel 6 beschrieben, wäre es bei der Erstellung dieser Arbeit sehr fruchtbar gewesen, wenn der hier beschriebene Ansatz und das Framework zur Softwaremessung mit einem externen Industriepartner hätte evaluiert werden können.

Akzeptanz im industriellen Einsatz

Mit dem hier dargestellten funktionsfähigem Prototyp würde ein entsprechender Praxiseinsatz des dargestellten Ansatzes wertvolle Erfahrungen über die Akzeptanz dieses Softwaremessungssystems bei den Anwendern im industriellen Alltag liefern.

Diesbezüglich wäre von besonderem Interesse, ob sich auch Unternehmen, welche bisher wegen des damit verbundenen Aufwandes von Softwaremessungsaktivitäten abgesehen haben, auf Grund der Leichtgewichtigkeit des dargestellten Ansatzes zur Durchführung von Softwaremessung bewegen lassen.

Definition der Erfahrungsdatenbank

Ein weiteres Ziel einer industriellen Evaluierung wäre die explizite Bestimmung derjenigen Softwaremaße, welche grundsätzlich innerhalb der jeweiligen Organisation bei jedem Projekt erhoben und, wie im Kapitel 6 beschrieben, in einer Erfahrungsdatenbank abgelegt werden sollen. Auch die Festlegung, wie eine solche Erfahrungsdatenbank grundsätzlich aufgebaut sein soll, sollte aus derartigen Praxiserfahrungen resultieren. Auch müsste in diesem Zusammenhang geklärt werden, welche Mitarbeiter in der betroffenen Organisation auf welche Daten in dieser Datenbank zugreifen dürfen. Auch wenn innerhalb dieser Arbeit davon abgeraten wird, mittels Softwaremessung die Produktivität einzelner Mitarbeiter beurteilen zu wollen (vgl. S. 89), so würde eine derartige Erfahrungsdatenbank dennoch unweigerlich auch Informationen enthalten, welche trotz eventueller Anonymisierung einzelnen Mitarbeitern zuordenbar sind.

Anwendung auf andere Entwicklungsdomänen

Im Rahmen dieser Arbeit wurde dieser Softwaremessungsansatz nur im Umfeld der Java-Programmierung dargestellt und implementiert. Hinweise auf die Integrationsmöglichkeiten in Großrechner- bzw. SAP-R/3-Umgebungen wurden bereits in Kapitel 6 gegeben. Interessant wäre jedoch auch, die Anwendbarkeit auf aktuelle Web-Frameworks wie *Ruby on Rails* (vgl. [WB06]) zu untersuchen. Derartige Entwicklungsansätze zeichnen sich unter anderem dadurch aus, dass bei der Programmierung nicht strikt zwischen Präsentations- und Datenbankschicht unterschieden werden kann. Stattdessen genügt es beispielsweise, die Spalten einer Tabelle nur in der Datenbank festzulegen. Der zugehörige Code zur Eingabe und Visualisierung der Daten wird dann mittels des Rails-Frameworks generiert (vgl. [WB06], S. 24 f.). Hier sind daher auch andere Ansätze der Projektstrukturierung und des Projektcontrollings erforderlich.

Abschließend kann somit festgestellt werden, dass das im Rahmen dieser Arbeit dargestellte *Framework zur automatisierten Softwaremessung* einen ersten Prototyp einer neuen Generation

- leichtgewichtiger,
- flexibler und
- kontextorientierter

Softwaremessungswerkzeuge darstellt, welcher als Grundlage weiterer Forschungsaktivitäten dienen kann.

A Vorgehensmodelle

A.1 Das V-Modell XT

A.1.1 Hintergrund des V-Modell XT

Das *V-Modell XT* ist ein Vorgehensmodell zum Planen und Durchführen von Projekten und stellt eine Weiterentwicklung des 1997 veröffentlichten *V-Modell 97* dar. Dieses wurde 1997 mit der Veröffentlichung des *Entwicklungsstandards für IT-Systeme des Bundes* (EStdIT) als Regelwerk für die Entwicklung, Pflege und Änderung von IT-Systemen im Bereich der Bundesverwaltung festgelegt.

Das V-Modell 97 wurde nach der Fertigstellung im Jahr 1997 nicht mehr weiter fortgeschrieben und spiegelte den damaligen Stand der Informationstechnologie wider. Neuere Methoden und Technologien wie die komponentenbasierte Entwicklung oder der Test-First-Ansatz werden daher im V-Modell 97 nur bedingt berücksichtigt (vgl. [BMI06], S. 5).

Vor diesem Hintergrund wurde durch das *Bundesministerium des Innern* (BMI) und das *Bundesamt für Informationsmanagement und Informationstechnik der Bundeswehr* (ITAmtBw) das Projekt *Weiterentwicklung des Entwicklungsstandards für IT-Systeme des Bundes auf Basis des V-Modell 97* (WEIT) bei der Technischen Universität München (TUM) und den Partnern IABG, EADS, Siemens AG, 4Soft GmbH und TU Kaiserslautern in Auftrag gegeben, welches im Jahr 2004 in der Veröffentlichung des V-Modell XT mündete. Dessen Einsatz ist inzwischen verbindlich für Behörden der Bundesverwaltung bei neu zu entwickelnden Systemen vorgeschrieben (vgl. [FHKS08]).

Ziele des V-Modell XT

Ziele der Weiterentwicklung waren dabei insbesondere

- die Verbesserung der Anpassbarkeit, Anwendbarkeit, Skalierbarkeit und Modifizier- und Erweiterbarkeit des V-Modells,
- die Berücksichtigung des neuesten Stands der Technologie und Anpassung an aktuelle Vorschriften und Normen,
- die Erweiterung des Anwendungsbereiches auf die Betrachtung des gesamten Systemlebenszyklus im Rahmen von Entwicklungsprojekten und
- die Einführung eines organisationsspezifischen Verbesserungsprozesses für Vorgehensmodelle.

A.1.2 Aufbau des V-Modell XT

Vorgehensbausteine

Beim V-Modell XT dienen sogenannte *Vorgehensbausteine* (vgl. Abbildung A.1) als grundlegende, strukturierende Einheiten, welche jeweils eine Aufgabenstellung repräs-

sentieren, die im Rahmen eines V-Modell-Projekts auftreten kann. Dabei werden diejenigen Produkte, Aktivitäten und Rollen in einem Vorgehensbaustein zusammengefasst, welche für die Erfüllung der zugehörigen Aufgabenstellung relevant sind, wie beispielsweise die Inhalte des Projektmanagements oder der Softwareentwicklung.

Produkte werden im V-Modell XT mit abgerundeten Ecken, Aktivitäten dagegen in Form von Rechtecken dargestellt (vgl. Abbildungen A.1 und A.2).

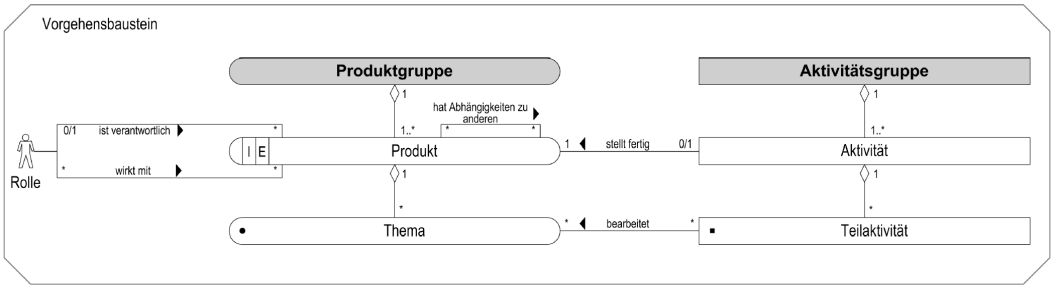


Abbildung A.1: Struktur eines Vorgehensbausteins aus dem V-Modell XT (siehe [BMI06], S. 13)

Als Produkte werden die zu erarbeitenden Ergebnisse und Zwischenergebnisse bezeichnet. Die Gesamtheit aller Produkte wird hierarchisch strukturiert, indem inhaltlich eng zusammengehörende Produkte zu Produktgruppen zusammengefasst werden. Darüber hinaus kann ein komplexes Produkt in mehrere Themen gegliedert sein. Analog werden Aktivitäten zu Aktivitätsgruppen zusammengefasst bzw. können noch weiter in Teilaktivitäten gegliedert werden. Die so gebildeten Vorgehensbausteine sind intern hierarchisch aufgebaut und bilden die modularen Einheiten des V-Modells.

Beispiel eines Vorgehensbausteins

Beispielsweise zeigt Abbildung A.2 den Vorgehensbaustein *Softwareentwicklung*, welcher die Aktivitätsgruppen

- *Systementwurf*,
- *Systemspezifikation* und
- *Systemelemente*

samt der gleichnamigen Produktgruppen enthält. Innerhalb dieses Vorgehensbausteins sind die Rollen *Software-Architekt* und *Software-Entwickler* aktiv, welche für Produkte wie die Software-Spezifikation oder den Datenbankentwurf zuständig sind bzw. die eigentliche Implementierung der Software durchführen. Die Erzeugung von Software-Code erfolgt dabei innerhalb der Aktivität „*Software-Modul realisieren*“. „Ein SW-Modul findet sich auf der untersten Hierarchieebene der Systemelemente und wird im Gegensatz zu allen anderen SW-Elementen durch ein nicht weiter unterstrukturiertes Stück Programmcode konkret realisiert“ (siehe [BMI06], S. 323).

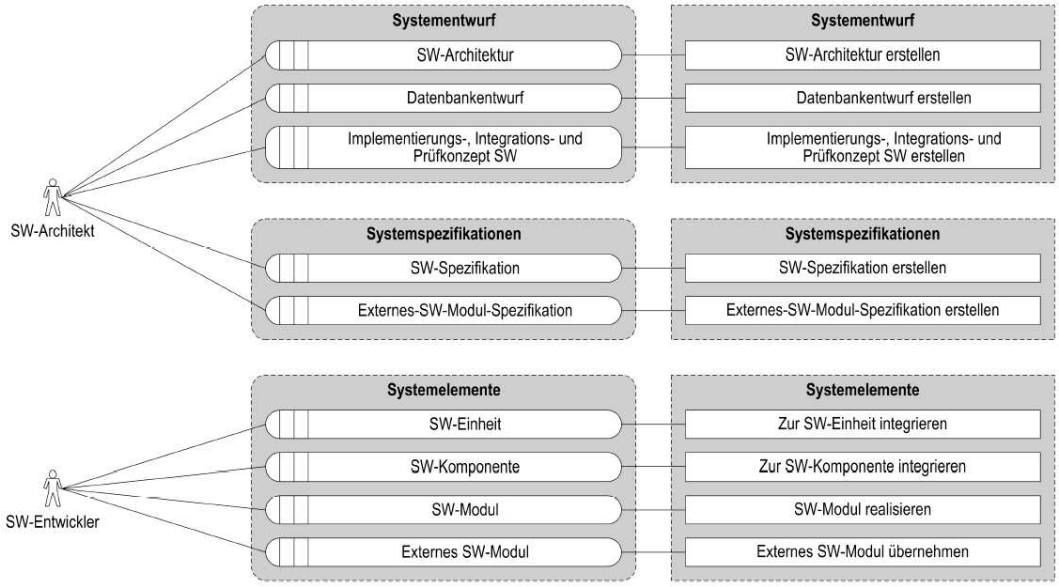


Abbildung A.2: Inhalte des Vorgehensbausteins Softwareentwicklung (siehe [BMI06], S. 109)

Vorgehensbaustein	Sinn und Zweck
Systemerstellung	Definition des Grundgerüsts der Systementwicklung, auf dem weitere Vorgehensbausteine aufbauen
Evaluierung von Fertigprodukten	Marktsichtung und technische Bewertung von potentiell einsetzbaren Fertigprodukten
Logistikkonzeption	Ausgestaltung und logistische Unterstützung der Lebensphasen des ausgelieferten Systems
Software-Entwicklung	Realisierung der innerhalb der Systemarchitektur (⇒ Systemerstellung) identifizierten Softwareeinheiten
Hardware-Entwicklung	Entwurf, Definition, Realisation und Integration der erforderlichen Hardware
Weiterentwicklung und Migration von Altsystemen	Planung und Durchführung von Weiterentwicklungsmaßnahmen bestehender Systeme bzw. der Migration derselben
Benutzbarkeit und Ergonomie	Gestaltung der Schnittstellen zwischen Benutzer und System
Systemsicherheit	Durchführung und Bewertung von Gefährdungs- und Systemsicherheitsanalysen
Vertragsschluss (AN)	Aktivitäten im Rahmen des Ausschreibungs- und Vertragswesens (Auftragsnehmer)
Lieferung und Abnahme (AN)	Erstellung der Lieferung und Auslieferung (Auftragsnehmer)

Tabelle A.1: Mögliche Vorgehensbausteine für ein *Systementwicklungsprojekt* auf Auftragnehmerseite

Tailoring beim V-Modell XT

Das V-Modell XT beinhaltet derzeit (Version 1.2) 21 Vorgehensbausteine, aus denen im Rahmen des Projekt-Tailorings je nach Projekt-Typ die notwendigen und sinnvollen ausgewählt werden. Die Dokumentation zum V-Modell XT schlägt beispielsweise für ein Systementwicklungsprojekt auf Auftragnehmerseite[1], d. h. für die Organisation, welche die eigentliche Implementierung durchführt, die in Tabelle A.1 aufgeführten Vorgehensbausteine vor.

Aktivitätsgruppen

Diese Vorgehensbausteine sind, wie oben erwähnt, in Aktivitätsgruppen und diese wiederum in einzelne Aktivitäten gegliedert. Dabei ist jede Aktivität zwar eindeutig einem Vorgehensbaustein und innerhalb dessen einer Aktivitätsgruppe zugeordnet, jedoch können die Aktivitätsgruppen selbst Bestandteile verschiedener Vorgehensbausteine sein.

So kommt die Aktivitätsgruppe *Systemelemente* in den Vorgehensbausteinen *Systemerstellung*, *Hardwareentwicklung* und *Softwareentwicklung* vor (Abbildung A.3).

Die Aktivitätsgruppen gliedern die Aktivitäten nach inhaltlichen Aspekten und dienen im Wesentlichen dazu, den Überblick über die Vielzahl der Aktivitäten des V-Modell XT zu verbessern. Im V-Modell XT sind dreizehn Aktivitätsgruppen de-

[1] Man beachte, dass der Vorgehensbaustein *Anforderungsfestlegung* ausschließlich der Auftraggeberseite zugeordnet ist.

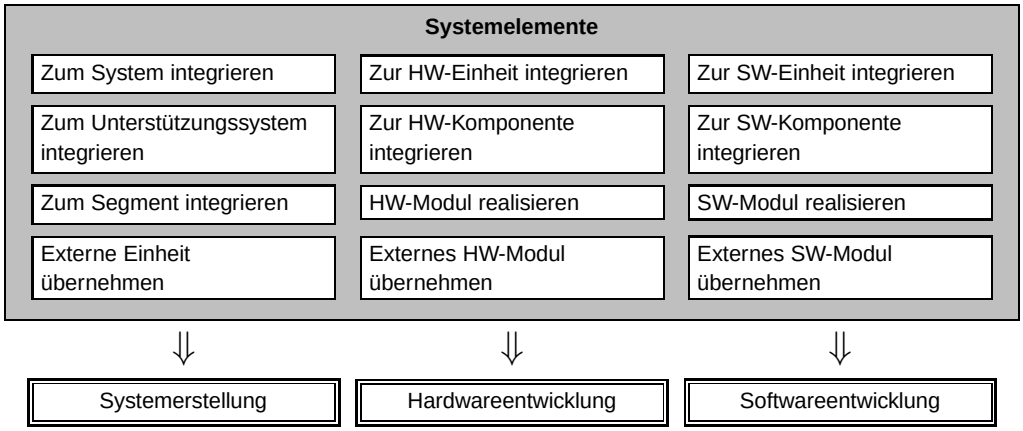


Abbildung A.3: Verteilung der Aktivitäten der Aktivitätsgruppe *Systemelemente* auf drei Vorgehensbausteine

finiert. Diese können zur weiteren Strukturierung den Bereichen

- Projektmanagement,
- Entwicklung und
- Organisation

zugeordnet werden (vgl. [BMI06], S. 409).

Dieser durch die Aktivitätsgruppen bewirkte Katalogisierungsaspekt zeigt sich z. B. anhand der Aktivitätsgruppe *Prüfung* (siehe [BMI06], S. 551 ff.) recht deutlich. Zu dieser Aktivitätsgruppe gehören alle Aktivitäten, welche die Tätigkeiten *Prüfen*, *Testen* oder *Kontrollieren* beinhalten. So gehört zu dieser Aktivitätsgruppe

- die Überprüfung von Spezifikationsdokumenten,
- das Prüfen von Systemelementen oder
- die Kontrolle von Lieferungen.

Beziehungen zwischen Aktivitäten, Aktivitätsgruppen und Vorgehensbausteinen

Wie in Abbildung A.3 dargestellt wird, besteht die Aktivitätsgruppe *Systemelemente* aus zwölf Aktivitäten. Jeweils vier davon sind Bestandteil der Vorgehensbausteine *Systemerstellung*, *Hardwareentwicklung* bzw. *Softwareentwicklung*. Umgekehrt enthält der Vorgehensbaustein *Softwareentwicklung* insgesamt neun Aktivitäten, welche aus drei verschiedenen Aktivitätsgruppen stammen (vgl. Abbildung A.2). Dabei sind die Mengen der Aktivitäten, welche den einzelnen Vorgehensbausteinen zugeordnet sind, disjunkt, d. h. jede Aktivität ist genau einem Vorgehensbaustein zugeordnet.

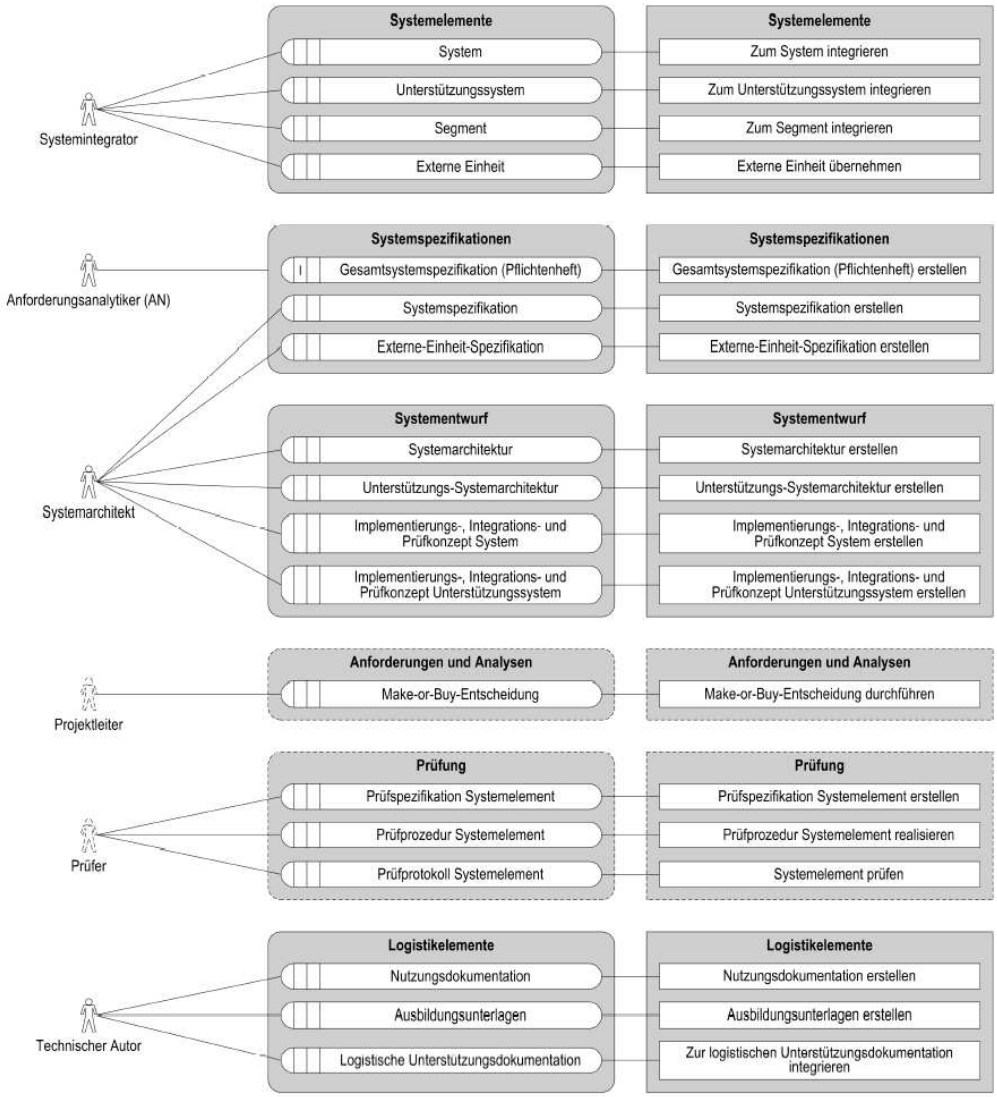


Abbildung A.4: Vorgehensbaustein *Systemerstellung* (aus [BMI06], S. 107)

A.2 Der Rational Unified Process

A.2.1 Der RUP als Produkt

Der *Rational Unified Process* (RUP, [Kru04]) ist ein iteratives Vorgehensmodell zur Softwareentwicklung, welches von der Firma *Rational Software Corporation* entwickelt wurde, welche wiederum inzwischen zum IBM-Konzern gehört. Der RUP wird üblicherweise nicht als feststehender Prozessleitfaden betrachtet, sondern stellt ein Rahmenwerk dar, welches von der anwendenden Organisation entsprechend ihrer Bedürfnisse durch Tailoring angepasst wird.

Der RUP als Softwareprodukt

Darüber hinaus ist der Rational Unified Process auch eine Software-Anwendung, mittels derer die Inhalte des RUP visualisiert werden können. Bei dieser Software handelt es sich im Wesentlichen um eine Web-Anwendung, welche über die Aktivitäten, Produkte, Rollen und Werkzeuge des Software-Prozesses mittels entsprechend verlinkter Web-Dokumente informiert. Beispielsweise gibt es im RUP einen Teilprozess, welcher als *Analysis & Design Workflow* bezeichnet wird. Dieser Workflow besteht aus verschiedenen *Aktivitäten*, welche von Personen, die bestimmte *Rollen* innehaben, unter Verwendung geeigneter *Werkzeuge* ausgeführt werden, wobei bestimmte Produkte (*Artefakte*) verwendet, modifiziert oder erzeugt werden. Der RUP beschreibt nun nicht nur den Ablauf der Aktivitäten und die Anforderungen an die Personen, welche bestimmte Rollen übernehmen, sondern enthält auch explizite Anleitungen (in Form sog. *Tool Mentoren*), wie diese Aktivitäten mit Hilfe ebenfalls von IBM vertriebener Softwareentwicklungswerkzeuge durchgeführt werden können. Auch der Inhalt und die Darstellung dieser Web-Anwendung, welche dann letztlich eine Instanz des RUP repräsentiert, kann mittels entsprechender Software (derzeit als *Rational Method Composer* bezeichnet) modifiziert werden. Im Rahmen dieser Anpassung können beispielsweise auch die Werkzeug-Anleitungen in Bezug auf andere, nicht von IBM stammende Entwicklungswerkzeuge, angepasst werden.

Tool-Mentoren

A.2.2 Die Struktur des RUP

Der RUP ist, wie in Abbildung A.5 dargestellt, hinsichtlich zweier Dimensionen gegliedert. Die horizontale Dimension repräsentiert die dynamische oder zeitliche Struktur des Prozesses. Hier ist der Prozess in Phasen gegliedert, welche wiederum mehrere Iteration enthalten können und jeweils durch einen Meilenstein abgeschlossen werden. Daneben soll in der vertikalen Dimension des RUP dessen statische Struktur in Form verschiedener *Disziplinen* wie *Business Modeling* oder *Project Management* dargestellt werden.

Keine Neuauflage des Wasserfallmodells

Kroll und Kruchten betonen, dass die RUP-Phasen nicht einfach eine zeitgemäße Umbenennung der Phasen des klassischen Wasserfallmodells seien. Die einzelnen Phasen geben nicht wie beim Wasserfallmodell an, welche Aktivitäten ausschließlich innerhalb dieser Zeitabschnitte durchgeführt werden, sondern sie geben den jeweiligen Schwerpunkt der Aktivitäten an. So wird erwartet, dass innerhalb der Inception-Phase der Fokus der Tätigkeiten auf der Ermittlung von Anforderungen liegt, während ein Großteil der eigentlichen Implementierung innerhalb der

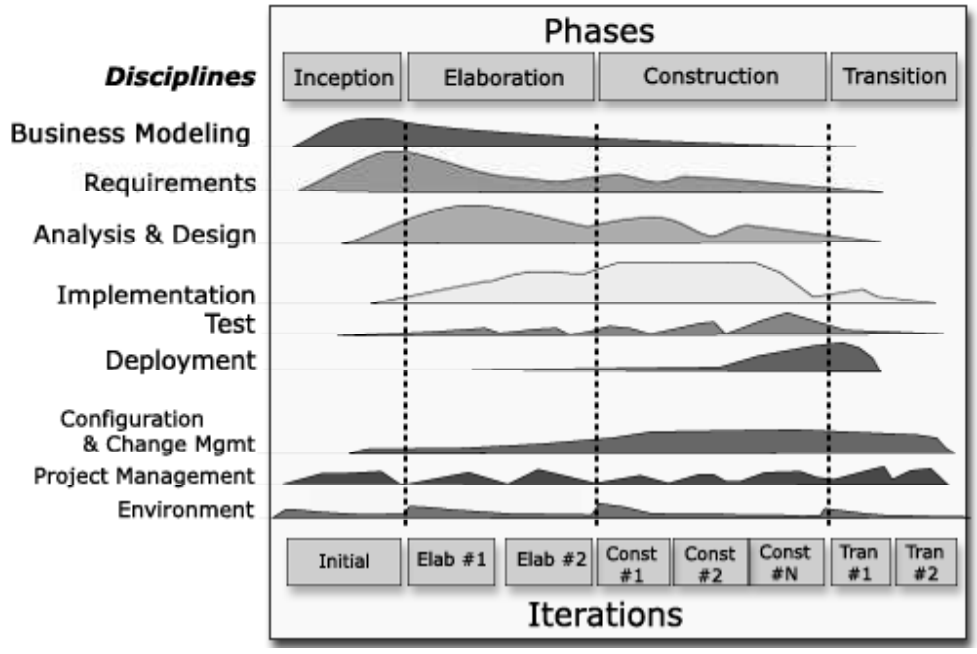


Abbildung A.5: Struktur des *Rational Unified Process* [Kru04]

Construction-Phase stattfinden wird. Jedoch kann, im Gegensatz zum klassischen Wasserfallmodell, grundsätzlich jede Tätigkeitsart in allen Phasen des Projektes auftreten (vgl. [KK05], S. 87 f.).

Disziplinen und Workflows

Die Disziplinen der vertikalen RUP-Dimension dienen zur Gruppierung der Aktivitäten des Prozesses in „logische Container“. Eine weitere, teilweise zu den Disziplinen orthogonal stehende Gruppierungsart stellen die sogenannten *Workflows* dar. Ein Workflow stellt dabei eine Folge von Aktivitäten dar, durch die ein Ergebnis von erkennbaren Wert erzeugt wird. Dabei werden innerhalb des RUP drei Workflow-Arten unterschieden:

Core Workflows: Jeder Disziplin ist ein gleichnamiger *Core Workflow* zugeordnet, welcher den Ablauf und die Abhängigkeiten der zu *Workflow-Details* zusammengefassten Gruppen von Aktivitäten grob beschreibt (vgl. [Kru04], S. 44 ff.). Beispielsweise zeigt Abbildung A.7, dass während der Inception-Phase vom *Analysis & Design Workflow* nur – und auch nur optional – das Workflow-Detail *Perform Architectural Synthesis* durchgeführt wird, dessen Inhalt eine Machbarkeitsstudie ist.

Workflow Details: Die Workflow-Details beschreiben eng zusammengehörige Aktivitäten innerhalb eines Workflows. Beispielsweise werden diese Aktivitäten typischerweise von einer Gruppe Projektbeteiligter gemeinsam durchgeführt oder es wird dabei ein wichtiges Zwischenergebnis erstellt. Jedoch können die einzelnen Aktivitäten auch in mehreren Workflow-Details auftreten. In den

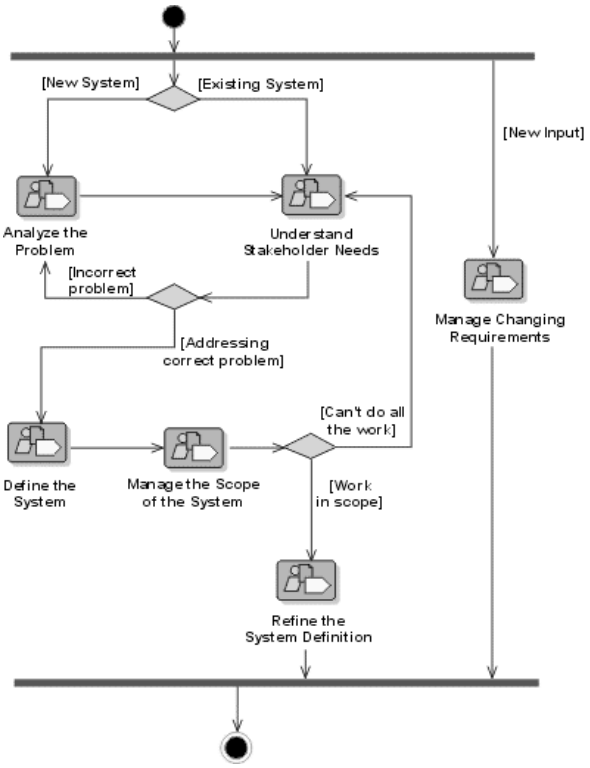


Abbildung A.6: Core-Workflow *Requirements*

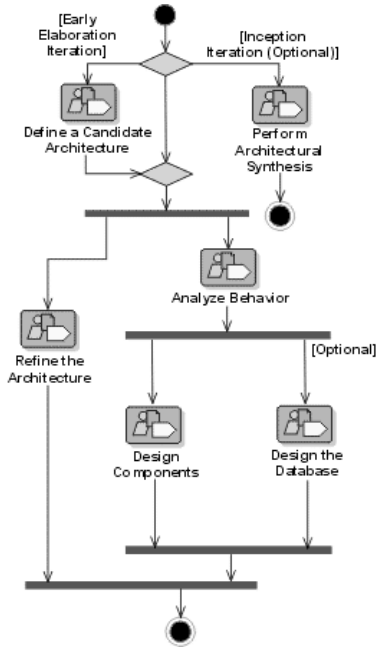


Abbildung A.7: Core-Workflow *Analysis & Design*

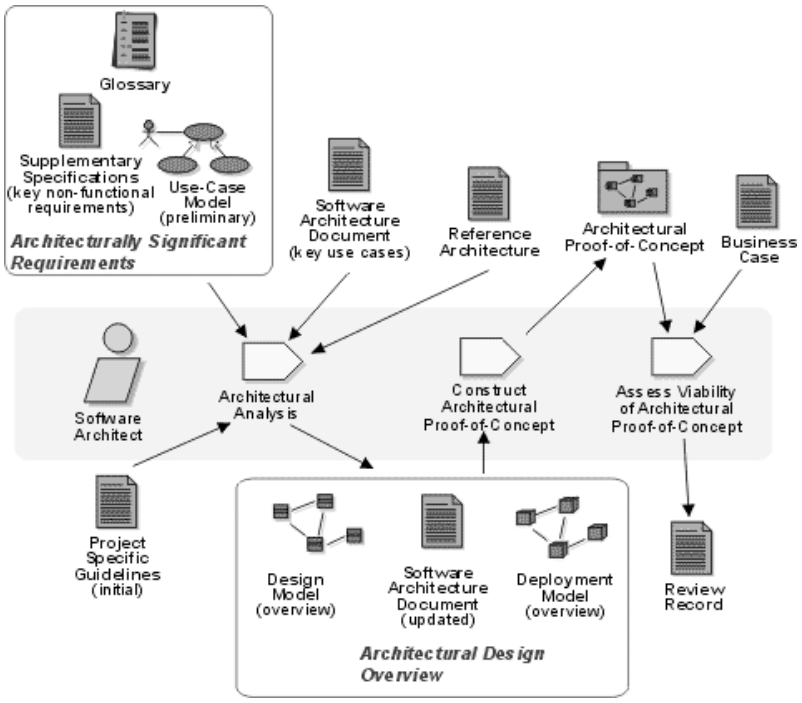


Abbildung A.8: Workflow-Detail *Perform Architectural Synthesis*

Workflow-Details wird auch der Informationsfluss dargestellt, d. h. welche Artefakte als Input bzw. Output für die jeweiligen Aktivitäten fungieren und wie die Aktivitäten über die Artefakte miteinander verbunden sind (vgl. [Kru04], S. 44 ff., [KK05], S. 16 f.).
Abbildung A.8 zeigt als Beispiel, wie im Rahmen des Workflow-Details *Perform Architectural Synthesis* zunächst eine Übersicht über die Architektur der zu erstellenden Software entwickelt wird. Daraufhin wird ein Architektur-Prototyp (*Architectural Proof-of-Concept*) erstellt, welcher dann anschließend beurteilt wird.

Iterationspläne: Iterationspläne beschreiben, welche Aktivitäten innerhalb einer bestimmten Iteration ausgeführt werden. Da die Anzahl der Iterationen für eine bestimmte Phase und deren Ausgestaltung vom jeweiligen konkreten Projekt abhängt, werden Iterationspläne für jedes Projekt und für jede Iteration neu erstellt. Die RUP-Web-Anwendung enthält daher nur einige Beschreibungen für typische Iterationspläne zu Demonstrationszwecken (vgl. [Kru04], S. 47). Für jede Iteration soll dabei zunächst beschrieben werden, was innerhalb dieser Iteration erreicht werden soll und gemäß welcher Kriterien das Erreichte überprüft werden kann. Anschließend werden die dazu notwendigen Aktivitäten und die dadurch betroffenen Artefakte bestimmt und Verantwortlichkeiten festgelegt (vgl. [KK05], S. 236 ff.).

Literaturverzeichnis

[ABBD06] Alain Abran, Manfred Bundschuh, Günter Büren, and Reiner R. Dumke, editors. *Applied Software Measurement – Proceedings of the International Workshop on Software Metrics and DASMA Software Metrik Kongress IWSM/MetriKon 2006*. Shaker, 2006.

[ABCR94] William W. Agresti, Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Measurement. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 762–775. John Wiley & Sons, 1994.

[AC77] Algirdas A. Avižienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *Proceedings of the COMPSAC 1977*. IEEE, 1977.

[ADH+01] Klaus-Dieter Althoff, Björn Decker, Susanne Hartkopf, Andreas Jedlitschka, Markus Nick, and Jörg Rech. Experience Management: The Fraunhofer IESE Experience Factory. In *Industrial Conference on Data Mining 2001*, pages 12–29, 2001.

[ADO+03] A. Abran, J-M. Descharnais, S. Oligny, D. St. Pierre, and C. R. Symons. *COSMIC-FFP Measurement Manual v2.2 – The COSMIC Implementation Guide for ISO/IEC 19761:2003*. Common Software Measurement International Consortium (COSMIC), Montrèal, Canada, 2003.

[ADSJ01] Bente Anda, Hege Dreiem, Dag I. K. Sjøberg, and Magne Jørgensen. Estimating software development effort based on use cases - experiences from industry. In M. Gogolla and C. Kobryn, editors, *4th International Conference on the Unified Modeling Language (UML2001)*, volume 2185 of *LNCS*, pages 487–502, Toronto, Canada, 2001. Springer.

[AG83] Allan J. Albrecht and John E. Gaffney. Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering*, 9(6):639–648, 1983.

[AGB03] Martin Auer, Bernhard Graser, and Stefan Biffl. A survey on the fitness of commercial software metric tools for service in heterogeneous environments: Common pitfalls. In *Proceedings of the Ninth International Software Metrics Symposium (METRICS 2003)*. IEEE Computer Society, 2003.

[Alb79] Allan J. Albrecht. Measuring Application Development Productivity. In *Proceedings of IBM Applications Development joint SHARE/GUIDE Symposium*, pages 83–92, Monterey, CA, 1979.

[AP94] Agnar Aamodt and Enrich Plaza. Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI Communications*, 7(1):39–59, 1994.

[AS06] El Hachemi Alikacem and Houari A. Sahraoui. Generic metric extraction framework. In Alain Abran, Manfred Bundschuh, Günter Büren, and Reiner R. Dumke, editors, *Applied Software Measurement – Proceedings of the International Workshop on Software Metrics and DASMA Software Metrik Kongress IWSM/MetriKon 2006*, pages 383–390. Shaker, 2006.

[Avi95] Algirdas A. Avižienis. The methodology of n-version programming. In Michael R. Lyu, editor, *Software Fault Tolerance*. John Wiley, New York, NY, USA, 1995.

[B+00] Barry Boehm et al. *Software Cost Estimation with COCOMO II*. Prentice-Hall, Englewood Cliffs, New Jersey, 2000.

[Bal98] Helmut Balzert. *Lehrbuch der Software-Technik*, volume 2. Spektrum Akademischer Verlag, 2nd edition, 1998.

[Bas92] Victor R. Basili. Software modeling and measurement: The goal question metric paradigm. Technical report, University of Maryland Computer Science Department, September 1992. Technical Report CS-TR2956.

[BBS03] Marcel Bennicke, Walter Bischofberger, and Frank Simon. Eclipse auf dem Prüfstand: Eine Fallstudie zur statischen Programmanalyse. *OBJEKTspektrum*, 2003(5):22–28, 2003.

[BC99] Kent Beck and Dave Cleal. Optional scope contracts, 1999. http://www.xprogramming.com.

[BCR94] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Experience factory. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 511–519. John Wiley & Sons, 1994.

[BCRS94] Victor R. Basili, Gianluigi Caldiera, H. Dieter Rombach, and Rini van Solingen. Goal Question Metric Appraoch. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 578–583. John Wiley & Sons, 1994.

[Bec04] Kent Beck. *Extreme Programming explained: Embrace Change*. Addison-Wesley, Upper Saddle River, New Jersey, second edition, 2004.

[BF00] Manfred Bundschuh and Axel Fabry. *Aufwandsschätzung von IT-Projekten*. MITP, Bonn, 2000.

[BF01] Kent Beck and Martin Fowler. *Planning eXtreme Programming*. Addison-Wesley, Boston, 2001.

[BKL04] Walter Bischofberger, Jan Kühl, and Silvio Löffler. Sotograph – A Pragmatic Approach to Source Code Architecture Conformance Checking. In *Software Architecture. Proceedings of the First European Workshop, EWSA 2004*, volume 3047 of *LNCS*, pages 1–9, Berlin, 2004. Springer.

[BMI06] V-Modell XT, Version 1.2.1, PDF-Version, 2006. `http://www.v-modell-xt.de`.

[BN95] Richard Bache and Martin Neil. Introducing metrics into industry: A perspective an GQM. In *Software Quality Assurance and Measurement: A Worldwide Perspective*, pages 59–68. Internat. Thomson Computer Press, London [et al.], 1995.

[BNL03] Dirk Bayer, Andreas Noack, and Claus Lewerentz. Simple and Efficient Relational Querying of Software Structures. In *10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 216–225. IEEE, 2003.

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, New Jersey, USA, 1981.

[Bos03] Andy Bosch. O/R-Mapping mit Apache Torque. *JavaSPEKTRUM*, pages 18–21, 6 2003.

[BPKL06] Petra Becker-Pechau, Bettina Karstens, and Carola Lilienthal. Automatisierte Softwareüberprüfung auf der Basis von Architekturregeln. In *Software Engineering 2006 – Fachtagung des GI-Fachbereichs Softwaretechnik*, volume P-79 of *LNI*, pages 157–162. GI, 2006.

[BR88] Victor R. Basili and H. Dieter Rombach. The TAME Project: Towards Improvement-Oriented Software Environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, 1988.

[Brä04] Peter Bräutigam, editor. *IT-Outsourcing – Eine Darstellung aus rechtlicher, technischer, wirtschaftlicher und vertraglicher Sicht*. Schmidt, Berlin, 2004.

[BT05] Barry Boehm and Richard Turner. *Balancing Agility and Discipline*. Addison-Wesley, Boston, 2005.

[Bur02] Manfred Burghardt. *Projektmanagement*. Publicis Corporate Publishing, Erlangen, sixth edition, 2002.

[Car05] Edward R. Carroll. Estimating software based on use case points. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 257–265, New York, NY, USA, 2005. ACM Press.

[CH02] Andy Carmichael and Dyn Haywood. *Better Software Faster*. Prentice Hall, Upper Saddle River, New Jersey, 2002.

[CHM+07] M. Ciolkowski, J. Heidrich, J. Münch, F. Simon, and M. Radicke. Evaluating Software Project Control Centers in Industrial Environments. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM) 2007*, pages 314–323. IEEE, 2007.

[CKS03] Mary B. Chrissis, Mike Konrad, and Sandy Shrum. *CMMI: Guidelines for Process Integration and Product Improvement*. Addison-Wesley, Boston, 2003.

[Cla02] Elizabeth Clark. Tracking Software Progresss. In IFPUG, editor, *IT Measurement: Practical Advice from the Experts*, pages 223–236. Addison-Wesley, Boston, Massachusetts, 2002.

[Cla05] Mike Clark. *Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Applications*. Pragmatic Bookshelf, Raleigh, N.C, 2005.

[CLD99] Peter Coad, Eric Lefebvre, and Jeff De Luca. *Java Modeling in Color with UML*. Prentice Hall, Upper Saddle River (New Jersey, USA), 1999.

[CMM06] CMMI Product Team. CMMI for Development, Version 1.2. Technical report, Software enginering Institute, Carnegie-Mellon University, Pittsburgh, 2006. CMU/SEI-2006-TR-008.

[CMP+07] John Casey, Vincent Massol, Brett Porter, Carlos Sanchez, and Jason van Zyl. *Better Builds with Maven*. DevZuz, August 2007. `http:\www.devzuz.com`, Version 1.3.1.

[Coc01] Alistair Cockburn. *Agile Software Development*. Addison-Wesley, Boston, MA, 2001.

[Dau05] Bernhard Daubner. Empowering software development environments by automatic software measurement. In *Proc. 11th IEEE International Symposium Software Metrics (METRICS 2005)*. IEEE Computer Society, 2005.

[DEA98] S. Dami, J. Estublier, and M. Amiour. APEL: A graphical yet executable formalism for process modeling. *Automated Software Engineering: An International Journal*, 5(1):61–96, January 1998.

[Dek04] Ton Dekkers. Why Cosmic Full Function Points will Beat Classic Methods. *Information Economics Journal*, pages 30–31, December 2004.

[DH03] Bernhard Daubner and Andreas Henrich. Ein Plädoyer für Datenflussdiagramme aus der Sicht der Aufwandsschätzung und der agilen Softwareentwicklung. In Klaus R. Dittrich, Wolfgang König, Andreas Oberweis,

Kai Rannenberg, and Wolfgang Wahlster, editors, *INFORMATIK 2003 - Innovative Informatikanwendungen, Band 1, Beiträge der 33. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 2003 in Frankfurt am Main*, volume P-34 of *LNI*, pages 191–195. GI, 2003.

[DH04] Bernhard Daubner and Andreas Henrich. Integration of software measurement into the software development environment. *METRICS NEWS*, 9(1):19–25, 2004.

[DHW06a] Bernhard Daubner, Andreas Henrich, and Bernhard Westfechtel. Integrierte Softwaremessung durch Verankerung der Softwaremaße an Elementen des Vorgehensmodells. In *Software Engineering 2006 – Fachtagung des GI-Fachbereichs Softwaretechnik*, volume P-79 of *LNI*, pages 157–162. GI, 2006.

[DHW06b] Bernhard Daubner, Andreas Henrich, and Bernhard Westfechtel. A Lightweight Tool Support for Integrated Software Measurement. In *Applied Software Measurement – Proceedings of the International Workshop on Software Metrics and DASMA Software Metrik Kongress IWSM/MetriKon 2006*, pages 67–80. Shaker, 2006.

[DHW06c] Bernhard Daubner, Andreas Henrich, and Bernhard Westfechtel. Towards Anchoring Software Measures on Elements of the Process Modell. In *Proceedings of the First International Conference on Software and Data Technologies (ICSOFT 2006)*, Setubal, Portugal, 2006. INSTICC Press.

[DIN87] Dt. Institut für Normung, Berlin. *DIN 69901 - Projektwirtschaft: Projektmanagement*, 1987.

[Dum96] Reiner Dumke. *Softwarequalität durch Messtools: Assessment, Messung und instrumentierte ISO 9000*. Vieweg, Braunschweig, 1996.

[Dum99] Reiner Dumke. Metrics Tools - An Overview. *Metrics News*, 4(1), July 1999.

[Dum01] Reiner Dumke. *Software-Engineering*. Vieweg, 3rd edition, 2001.

[DW97] Reiner R. Dumke and Achim S. Winkler. Came Tools for an Efficient Software Maintenance. In *1st Euromicro Working Conference on Software Maintenance and Reengineering (CSMR '97)*, page 74, Los Alamitos, CA, USA, 1997. IEEE Computer Society.

[EDBSt05] Christof Ebert, Reiner Dumke, Manfred Bundschuh, and Andreas Schmietendorf. *Best Practices in Software Measurement*. Springer, Berlin, et. al., 2005.

[FC04] William A. Florac and Anita D. Carleton. *Measuring the Software Process – Statistical Process Control for Software Process Improvement*. Addison-Wesley, Boston, München, et. al., 2004.

[Feh05] Thomas Michael Fehlmann. *Six Sigma in der SW-Entwicklung*. Vieweg, 2005.

[FF04] Eric Freeman and Elisabeth Freeman. *Head first design patterns*. O'Reilly, Beijing, 2004.

[FHKS08] J. Friedrich, U. Hammerschall, M. Kuhrmann, and M. Sihling. *Das V-Modell XT*. Springer, 2008. to appear.

[Fie05] Rudolf Fiedler. *Controlling von Projekten*. Vieweg, Wiesbaden, 3rd edition, 2005.

[FJE06] Stephan Frohnhoff, Volker Jung, and Gregor Engels. Use Case Points in der industriellen Praxis. In Alain Abran, Manfred Bundschuh, Günter Büren, and Reiner R. Dumke, editors, *Applied Software Measurement – Proceedings of the International Workshop on Software Metrics and DASMA Software Metrik Kongress IWSM/MetriKon 2006*, pages 511–526. Shaker, 2006.

[FN99] Norman E. Fenton and Martin Neil. Software metrics: successes, failures and new directions. *The Jounal of Systems and Software*, 47:149–157, 1999.

[Fow04] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, 2004.
`http://martinfowler.com/articles/injection.html`.

[FP98] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Company, Boston, Massachusetts, second edition, 1998.

[Fuj05] Fujitsu Siemens Computers, München. *LMS (BS2000/OSD) V3.3 – Library Maintenance System*, Januar 2005. Datenblatt.

[Gar84] David A. Garvin. What does Product Quality really mean? *MIT Sloan Management Review*, 26:25 – 43, 1984.

[GC87] Robert B. Grady and Deborah L. Caswell. *Software Metrics: Establishing a Company-Wide Program*. Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[Gib06] Robert Gibbons. *A Primer in Game Theory*. Prentice Hall, Harlow, München, et. al., 2006.

[Gra92] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[GWI04] Bill Gilbert, Dave West, and Doug T. Ishigaki. IBM Rational Team Unifying Platform: IBM Rational ProjectConsole Sample Measures. Technical report, IBM Corporation, 2004.

[Hal77] Maurice H. Halstead. *Elements of Software Science*. Elsevier, New York, 1977.

[Har60] H. L. Harter. Tables of Ranges and Studentized Range. *The Annals of Mathematical Statistics*, 31:1122–1147, 1960.

[Har06] Holger Hartmann. Fortschrittsmessung bei agilen Softwareentwicklungsprojekten unter Verwendung von Open-Source-Werkzeugen. Bachelor-Arbeit, Universität Bayreuth, 2006.

[Hen02] Andreas Henrich. *Management von Softwareprojekten*. Oldenbourg, München, Wien, 2002.

[Het93] Bill Hetzel. *Making Software Measurement Work*. QED Publishing Group, Boston [et al.], 1993.

[HF94] Tracy Hall and Norman Fenton. Implementing software metrics – the critical success factors. *Software Quality Journal*, 3:195–208, 1994.

[HF97] Tracy Hall and Norman E. Fenton. Implementing Effective Software Metrics Programs. *IEEE Software*, 14(2):55–65, 1997.

[HKV94] Robert Hendrick, David Kistler, and Jon Valett. Software Management Environment (SME) – Components and Algorithms. Technical report, NaSA Goddard Space Flight Center, Greenbelt, Maryland, USA, 1994. Software Engineering Laboratory Series Report SEL-94-001.

[HKVD92] Robert Hendrick, David Kistler, Jon Valett, and William Decker. Software Management Environment (SME) – Concepts and Architecture – Revision 1. Technical report, NaSA Goddard Space Flight Center, Greenbelt, Maryland, USA, 1992. Software Engineering Laboratory Series Report SEL-89-103.

[HL06] Christian Hochberger and Rüdiger Liskowsky, editors. *INFORMATIK 2006 – Beiträge der 36. Jahrestagung der Gesellschaft für Informatik e. V. (GI)*, volume 2. GI, 2006.

[HM07a] Jens Heidrich and Jürgen Münch. Cost-Efficient Customisation of Software Cockpits by Reusing Configurable Control Components. In *Proceedings of the Software Measurement European Forum (SMEF) 2007*, pages 19–32, Rom, 2007.

[HM07b] Geog Herzwurm and Martin Mikusz. Six Sigma in der Softwareentwicklung. In *Software Engineering 2007 – Fachtagung des GI-Fachbereichs Softwaretechnik*, LNI, pages 253–254. GI, 2007.

[HMW06] Jens Heidrich, Jürgen Münch, and Axel Wickenkamp. Usage Scenarios for Measurement-based Project Control. In *Proceedings of the Software Measurement European Forum (SMEF) 2006*, pages 47–59, Rom, 2006.

[HRS04] Peter Hruschka, Chris Rupp, and Gernot Starke. *Agility kompakt*. Spektrum Akademischer Verlag, 2004.

[IBM04] IBM Corporation. *Understanding and Implementing Stakeholder Needs: The Integration of IBM Rational ClearQuest and IBM Rational Requisite Pro*, 2004.

[IFP04] IFPUG, editor. *Function point counting practices manual, Release 4.2*. International Function Point Users Group, Princeton Junction, New Jersey, 2004.

[IJ03] Doug T. Ishigaki and Cheryl Jones. Practical measurement in the rational unified process. *the Rational edge – e-zine for the rational community*, 2003. Online-Publication.

[Ima02] Masaaki Imai. *Kaizen – Der Schlüssel zum Erfolg im Wettbewerb*. Econ, München, 2002.

[Int01] International Organization for Standardization (ISO). *ISO/IEC 9126-1:2001, Software engineering – Product quality – Part 1: Quality model*, 2001. CSNUMBER=22749.

[Ish04] Doug T. Ishigaki. IBM Rational Team Unifying Platform: IBM Rational ProjectConsole for Microsoft Project Reporting & Measurements. Technical report, IBM Corporation, 2004.

[Jac92] Ivar Jacobson. *Object oriented Software Engineering: A Use Case driven Approach*. ACM Press, 1992.

[JKA+04] P. M. Johnson, H. Kou, J. M. Agustin, Q. Zhang, A. Kagawa, and T. Yamashita. Practical Automated Process and Product Metric Collection and Analysis in a Classroom Setting: Lessons learned from Hackystat-UH. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, Los Angeles, California, August 2004.

[JKP+05] P. M. Johnson, H. Kou, M. G. Paulding, Q. Zhang, A. Kagawa, and T. Yamashita. Improving Software Development Management through Software Project Telemetry. *IEEE Software*, 22(4), Juli/August 2005.

[Joh01] Philip M. Johnson. You can't even ask them to push a button: Towards ubiquitous, developer-centric, empirical software engineering. In *The NSF Workshop for New Visions for Software Design and Productivity: Research and Applications*, Nashvile, TN, December 2001.

[Jon96] Capers T. Jones. Software defect-removal efficiency. *IEEE Computer*, 29(4):94–95, 1996.

[Jon00] Capers T. Jones. *Software Assessments, Benchmarks and Best Practices*. Addison-Wesley, Boston, Münchenm, et. al., 2000.

[KA07] Dierk König and Kersten Auel. Gralshüter: Grails-Tutorial I – Installation und erste Anwendung. *iX – Magazin für professionelle Informationstechnik*, 6:144–147, 2007.

[Kan03] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, 2003.

[KB07] Gerd F. Kamiske and Jörg-Peter Brauer. *Qualitätsmanagement von A bis Z*. Hanser, München, 6. edition, 2007.

[KK05] Per Kroll and Philippe Kruchten. *The Rational Unified Process Made Easy*. Addison-Wesley, Boston, 2005.

[Kön07] Dierk König. *Groovy in action*. Manning, New York, 2007.

[Kne06] Ralf Kneuper. *CMMI – Verbesserung von Softwareprozessen mit Capability Maturity Model Integration*. dpunkt-Verlag, Heidelberg, 2nd edition, 2006.

[KO06] Andreas Kowitz and Christian Ofer. Durchführung eines Messprogramms: Ein Erfahrungsbericht. In Alain Abran, Manfred Bundschuh, Günter Büren, and Reiner R. Dumke, editors, *Applied Software Measurement – Proceedings of the International Workshop on Software Metrics and DASMA Software Metrik Kongress IWSM/MetriKon 2006*, pages 455–469. Shaker, 2006.

[KPR01] Seija Komi-Sirvio, Paivi Parviainen, and Jussi Ronkainen. Measurement automation: Methodological background and practical solutions – a multiple case study. In *Proceedings of the Seventh International Software Metrics Symposium (METRICS 2001)*, page 306 ff. IEEE Computer Society, 2001.

[KRSZ00] Ralf Kempkens, Peter Rösch, Louise Scott, and Jörg Zettel. Instrumenting measurement programs with tools. In *PROFES '00: Proceedings of the Second International Conference on Product Focused Software Process Improvement*, pages 353–375, London, 2000. Springer.

[Kru04] Philippe Kruchten. *The Rational Unified Process - An Introduction*. Addison-Wesley, Boston, third edition, 2004.

[Lee06] Clemens Lee. Javancss - A Source Measurement Suite for Java, 2006. http://www.kclee.de/clemens/java/javancss.

[Lot94] Christopher M. Lott. Technology trends survey: Measurement support in software engineering environments. *Int. Journal of Software Engineering and Knowledge Engineering*, 4(3), 1994.

[Lot96] Christopher M. Lott. *Measurement-based Feedback in a Process-centered Software Engineering Environment.* PhD thesis, University of Maryland, 1996.

[Lot97] Christopher M. Lott. Breathing new life into the waterfall model. *IEEE Software*, 14(5):103–105, 1997.

[Mat02] Bernd Matzke. *ABAP – Die Programmiersprache des SAP-Systems R/3.* Addison-Wesley, München, fourth edition, 2002.

[McC76] Thomas McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[MF00] Katrina D. Maxwell and Pekka Forselius. Benchmarking Software Development Productivity. *IEEE Software*, 17(1):80–88, 2000.

[MH04] Jürgen Münch and Jens Heidrich. Software project control centers: Concepts and approaches. *Journal of Systems and Software*, 70:3–19, 2004. Issues 1–2.

[MHS+06] Jürgen Münch, Jens Heidrich, Frank Simon, Claus Lewerentz, Birk Siegmund, Rainer Bloch, Bernhard Krupicz, and Martin Dehn. Softpit: Ganzheitliche Projekt-Leitstände zur ingenieurmäßigen Software-Projektdurchführung. In *Statuskonferenz Forschungsoffensive „Software Engineering 2006“*, Leipzig, 2006. Fraunhofer IESE, Bundesministerium für Bildung und Forschung (BMBF).

[MK00] Mala Murugappan and Gargi Keeni. Quality Improvement – The Six Sigma Way. In *APAQS 2000 – Proceedings of the First Asia-Pacific Conference on Quality Software*, pages 248–257, Hong Kong, China, 2000. IEEE.

[MK03] Mala Murugappan and Gargi Keeni. Blending CMM and Six Sigma to Meet Business Goals. *IEEE Software*, 20(2):42–48, March/April 2003.

[Mül03] Frank Müller. Agile Softwareentwicklung. *iX – Magazin für professionelle Informationstechnik*, 2:109–111, 2003.

[MR99] Thomas J. Mowbray and William A. Ruh. *Inside CORBA. Distributed Object Standards and Applications.* Addison-Wesley, 1999.

[MWD96] Katrina D. Maxwell, Luk Van Wassenhove, and Soumitra Dutta. Software Development Productivity of European Space, Military, and Industrial Applications. *IEEE Trans. Software Eng.*, 22(10):706–718, 1996.

[Nie07] Frank Niemann. Individualität bremst SAP-Systeme aus. *COMPUTERWOCHE*, 2007(50), 2007. `http://www.computerwoche.de/1221861`.

[Oak96] John S. Oakland. *Statistical Process Control*. Elsevier, 3rd edition, 1996.

[Obj05] Object Management Group. Software Process Engineering Metamodel, Version 1.1, 2005. http://www.omg.org.

[OW08] Bernd Oestereich and Christian Weiß. *APM – Agiles Software-Projektmanagement*. dpunkt, Heidelberg, 2008.

[Par97] Behrooz Parhami. Defect, Fault, Error, . . . , of Failure? *IEEE Transactions on Reliability*, 46(4):450–451, December 1997.

[PCCW93] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber. Capability Maturity Model for Software, Version 1.1. Technical report, Software enginering Institute, Carnegie-Mellon University, Pittsburgh, 1993. CMU/SEI-93-TR-024.

[PF02] Stephen R. Palmer and John M. Felsing. *A practical Guide to Feature-Driven Development*. Prentice Hall, Upper Saddle River (New Jersey, USA), 2002.

[PGF96] Robert E. Park, Wolfhart B. Goethert, and William A. Florac. Goal-Driven Software Measurement – A Guidebook. Technical report, Software enginering Institute, Carnegie-Mellon University, Pittsburgh, 1996. CMU/SEI-96-HB-002.

[PM02] Lawrence H. Putnam and Ware Myers. The Core of Software Planning. In IFPUG, editor, *IT Measurement: Practical Advice from the Experts*, pages 53–66. Addison-Wesley, Boston, Massachusetts, 2002.

[PM03] Lawrence H. Putnam and Ware Myers. *Five Core Metrics: The Intelligence behind successful Software Management*. Dorset House Publishing Co., New York, 2003.

[PPM07] ARIS Process Performance Manager. Factsheet, ID-Number: FS-PPM-0207-D, 2007. http://www.ids-scheer.com.

[PR98] Gerold Patzak and Günter Rattay. *Projekt-Management*. Linde, 3. edition, 1998.

[Pro04] Project Management Institute. *A Guide to the Project Management Body of Knowledge: PMBOK Guide*. Project Management Institute, Newtown Square, Pennsylvania, third edition, 2004.

[Put78] Lawrence H. Putnam. A general empirical solution to the macro software sizing and estimating problem. *IEEE Transactions on Software Engineering*, 4(4):345–361, 1978.

[QBe08] Qbench, 2008.
http://www.qbench.de.

[Rat02a] Rational Software Corporation. *Rational Suite AnalystStudio – Getting Started*. Cupertino (CA), 2002.

[Rat02b] Rational Software Corporation. *Rational TestManager – User's Guide*. Cupertino (CA), 2002.

[Ric06] Chris Richardson. *POJOs in Action: Developing Enterprise Applications with Lightweight Frameworks*. Manning, Greenwich, Connecticut, 2006.

[Riz06] Stefano Rizzo. Agiles Projekt- und Anforderungsmanagement: Der Live-Ansatz. *OBJEKTspektrum*, 2:38–44, 2006.

[RRK06] Rainer Radtke, Sabine Reszies, and Wolfang Klose. *Grundlagen Rechnungswesen & DATEV*. Teia Lehrbuch Verlag, 2006. eBook.

[Rus02] Janet Russac. Cheaper, Better, Faster: A Measurement Program That Works. In IFPUG, editor, *IT Measurement: Practical Advice from the Experts*, pages 147–158. Addison-Wesley, Boston, Massachusetts, 2002.

[RW05] Jörg Rech and Sebastian Weber. Werkzeuge zur Ermittlung von Software-Produktmetriken und Qualitätsdefekten – Studie zu Software-Messwerkzeugen 2005. Technical Report IESE 108.05/D, Fraunhofer Institut Experimentelles Software Engineering, 2005.

[SAP03] SAP AG. *Arbeitszeiterfassung mit SAP CATS*. Walldorf, 2003.

[SB99] Rini van Solingen and Egon Berghout. *The Goal/Question/Metric Method – A Practical Guide for Quality Improvement of Software Development*. McGraw-Hill, Berkshire (England), 1999.

[Sei03] Siegfried Seibert. Softwaremessung, quantitative Projektsteuerung und Benchmarking. *projektMANAGEMENT aktuell*, 4:26–34, 2003.

[Sel92] Richard W. Selby. Software Measurement and Experimentation Frameworks, Mechanisms, and Infrastructure. In *Experimental Software Engineering Issues. International Workshop Dagstuhl Castle, Germany, September 14-18, 1992*, volume 706 of *LNCS*, pages 89–106. Springer, 1992.

[Sel05] Richard W. Selby. Measurement-Driven Dashboards Enable Leading Indicators for Requirements and Design of Large-Scale Systems. In *11th IEEE International Symposium on Software Metrics (METRICS 2005)*. IEEE Computer Society, 2005.

[She02] Martin Shepperd. Case-based Reasoning and Software Engineering. Technical report, Empirical Software Engineering Research Group at Bournemouth University, November 2002. Technical Report TR02-08.

[Sie04] Johannes Siedersleben. *Moderne Software-Architektur. Umsichtig planen, robust bauen mit Quasar.* dpunkt, 2004.

[Sim01] Frank Simon. *Meßwertbasierte Qualitätssicherung: Ein generisches Distanzmaß zur Erweiterung bisheriger Softwareproduktmaße.* Dissertation, Technische Universität Cottbus, 2001.

[Sin06] Lorenz Singer. Implementierung der Halstead-Softwaremaße als Maven-Plugin. Bachelor-Arbeit, Universität Bayreuth, 2006.

[SK06] Frank Simon and Christian Koll. Traceability zwischen Metriken und dem strategischen Ziel Wartbarkeit. In Alain Abran, Manfred Bundschuh, Günter Büren, and Reiner R. Dumke, editors, *Applied Software Measurement – Proceedings of the International Workshop on Software Metrics and DASMA Software Metrik Kongress IWSM/MetriKon 2006*, page 41ff. Shaker, 2006.

[SL05] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest.* dpunkt.verlag, Heidelberg, 2005.

[Sne96] Harry M. Sneed. Schätzung der Entwicklungskosten von objektorientierter Software. *Informatik-Spektrum*, 19(3):133–0149, 1996.

[Son07] Maven: The Definitive Guide, 2007. `http:\www.sonatype.com`, Version 1.0 Alpha 2.

[SPSB91] Richard W. Selby, Adam A. Porter, Doug C. Schmidt, and Jim Berney. Metric-driven analysis and feedback systems for enabling empirically guided software development. In *Proceedings of the 13th International Converence on Software Engineering (ICSE 1991).* IEEE Computer Society, 1991.

[SQS05] SQS. QBench-Nomenklatur, 2005. http://www.qbench.de.

[SSM06] Frank Simon, Olaf Seng, and Thomas Mohaupt. *Code-Quality-Management.* dpunkt.verlag, Heidelberg, 2006.

[Sta05] Tim Stapenhurst. *Mastering Statistical Process Control – A Handbook for Performance Improvement Using Cases.* Elsevier, 2005.

[SW01] Geri Schneider and Jason P. Winters. *Applying Use Cases – A practical Guide.* Addison-Wesley, 2nd edition, 2001.

[SZ05] Manfred Schulte-Zurhausen. *Organisation.* Vahlen, 4. edition, 2005.

[THB+06] Dave Thomas, David Hansson, Leon Breedt, Mike Clark, James Duncan Davidson, Justin Gehtland, and Andreas Schwarz. *Agile Web Development with Rails.* Pragmatic Bookshelf, second edition, 2006.

[Töp04] Armin Töpfer, editor. *Six Sigma – Konzeption und Erfolgsbeispiele für praktizierte Null-Fehler-Qualität*. Springer, 3rd edition, 2004.

[TT06] Sylvie Trudel and Pascale Tardif. Successes and challenges experienced in implementing a measurement program in small software organizations. In Alain Abran, Manfred Bundschuh, Günter Büren, and Reiner R. Dumke, editors, *Applied Software Measurement – Proceedings of the International Workshop on Software Metrics and DASMA Software Metrik Kongress IWSM/MetriKon 2006*, pages 229–244. Shaker, 2006.

[TZ97] Roseanne Tesoriero and Marvin Zelkowitz. The Web Measurement Environment (WebME): A Tool for Combining and Modeling Distributed Data. In *22nd Annual Softwrae Engineering Workshop (SEW)*, 1997.

[Ull07] Christian Ullenboom. *Java ist auch eine Insel*. Galileo Computing, sixth edition, 2007.

[VTT99] VTT Electronics, Oulu, Finland. *MetriFlame User Guide*, 1999. http://virtual.vtt.fi/metriflame.

[WB06] Ralf Wirdemann and Thomas Baustert. *Rapid Web Development mit Ruby on Rails*. Hanser, München, 2006.

[Wik] Wikipedia. `http://de.wikipedia.org`.

[Wol03] Björn Wolle. Statische Analyse von Java-Anwendungen: Eignen sich Lines-of-Code-Metrik und Halstead-Länge? *WIRTSCHAFTSINFORMATIK*, 45(1):29–49, 2003.

[Zah05] Christoph Zahrnt. *Richtiges Vorgehen bei Verträgen über IT-Leistungen*. dpunkt.verlag, Heidelberg, 2nd edition, 2005.

[Zül04] Heinz Züllighoven. *Object-Oriented Construction Handbook*. dpunkt, 2004.

[ZM07] Thomas Zehler and Jürgen Münch. Soft-pit: Ganzheitliche Projekt-Leitstände zur ingenieurmäßigen Software-Projektdurchführung. Technical report, Fraunhofer Institut Experimentelles Software Engineering, 2007. `http://www.software-kompetenz.de/?20614`.

[Zus98] Horst Zuse. *A Framework of Software Measurement*. de Gruyter, Berlin, New York, 1998.

[Zus03] Horst Zuse. Kommentar zum Artikel von Björn Wolle: Statische Analyse von Java-Anwendungen: Eignen sich Lines-of-Code-Metrik und Halstead-Länge? *WIRTSCHAFTSINFORMATIK*, 45(6), 2003.

Index