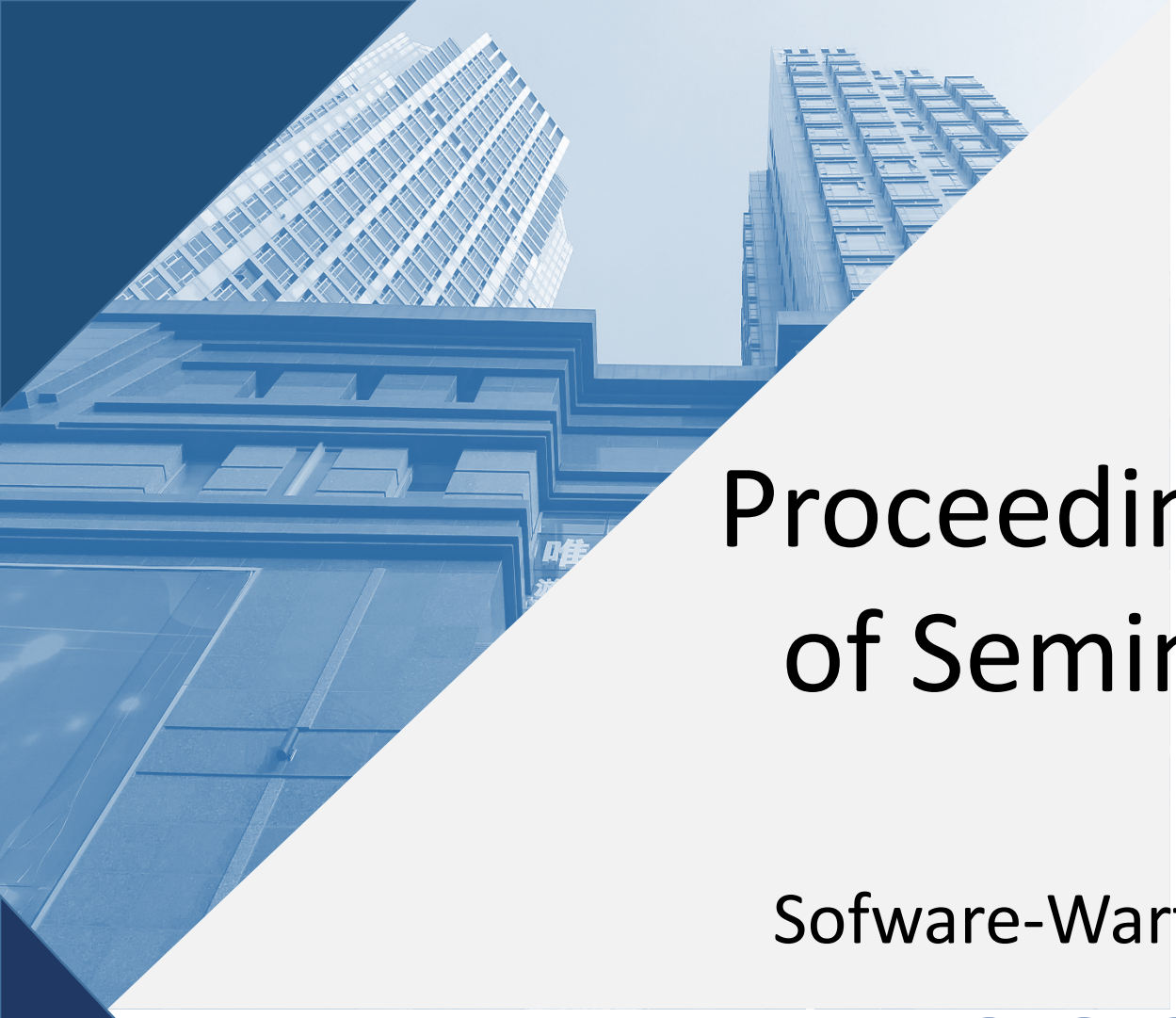


Proceedings of Seminar

Sofware-Wartung

2006

Editors: Horst Lichter
Holger Schackmann

SWC Software Construction | RWTH AACHEN UNIVERSITY

Inhaltsverzeichnis

Kapitel 1

Laws of Software Evolution

Miriam Günther

Inhaltsverzeichnis

1.1 Einleitung

Seit der Software-Krise in den späten 60er-Jahren hat sich viel im Bereich der Softwaretechnik geändert, um die damaligen gravierenden Programmsystemfehler und die Problematik der nicht vorhandenen Lösungsansätze zu überwinden. Seitdem sind neue Programmiersprachen, Methoden, Werkzeuge und Unterstützung (CASE) entwickelt worden, welche aber meistens der Neuentwicklung von Software dienen. Der Wartungsprozess wurde in dieser Hinsicht vernachlässigt, dabei macht er über 60 % des Aufwands der Entwicklung eines Systems aus [Boe81]. Auch heute ist also der Softwareentwicklungsprozess immer noch kompliziert und nicht vollständig durchschaut. Bekannte Beispiele für gescheiterte Software-Projekte sind die Explosion der Ariane 5-Rakete aufgrund nicht vorhandener Ausnahmebehandlung und die fehlgeschlagene Synchronisierung von Fließband und Gepäckwagen beim Gepäcktransportsystem des Flughafens Denver. Dies sind nur zwei Beispiele, die Liste von fehlerhafter Software ist aber lang. Aus diesem Grund wird immer noch nach Verbesserungen des Softwareentwicklungsprozesses gesucht. Lehman versucht dazu mit seinen in dieser Arbeit vorgestellten Studien beizutragen.

Es gibt viele verschiedene Definitionen von Wartung, welche Teilaufgaben dazu gehören und wann sie beginnt. Hier soll unter Wartung der Prozess aller Änderungen eines im Einsatz befindlichen Softwaresystems nach Auslieferung verstanden werden [Som01]. Eine zentrale Fragestellung in der Wartung ist die Dynamik der Evolution von Programmen, welche ständige Veränderungen ausmacht. Zu diesen Änderungen in komplexen Systemen hat Lehman im Rahmen seiner Forschungsarbeit bei IBM und später im universitären Kontext Studien durchgeführt, indem er das Wachstum und die Evolution mehrerer großer Systeme untersucht hat. Das Ergebnis, die Formulierung der Gesetze der Software-Evolution, soll die Gesetzmäßigkeiten bzw. das Verhalten von Systemen bei Veränderung oder Evolution beschreiben. Da diese Gesetze allgemein gültig zu sein scheinen, sollten ihre Aussagen bei der Planung des Wartungsprozesses mitbetrachtet werden. Allgemeines Ziel von Lehmans Forschung ist es, den Prozess der Softwareentwicklung und -wartung zu verbessern, also effektiver, günstiger und schneller gestalten und vor allem besser planen und kontrollieren zu können.

Ein Aspekt der Wartung bezieht sich auf die Pflege des Software-Produkts, d.h. es werden Änderungen oder auch Erweiterungen vorgenommen. Bei jedem Änderungs- oder Erweiterungsvorgang entsteht eine neue Version der Software, auch Release genannt. Die Evolution eines Software-Produkts kann man anhand seiner Versionen erkennen. Entsprechend der Unterteilung in perfektionierende, adaptive und korrektive Wartung werden auch Releases in drei Kategorien unterteilt [Leh97]. „Major Mainstream Releases“ beinhalten relativ viel neuen Quellcode und fügen dem Programm neue Eigenschaften hinzu. „Minor Mainstream Releases“ beziehen sich auf nur kleine Verbesserungen bzw. Anpassungen des Systems. Schließlich werden bei „Error Correction Releases“ nur Fehler korrigiert, es wird keine zusätzliche Erweiterung der Funktionalität vorgenommen. In der Praxis kann man aber meistens bei der Veröffentlichung eines Releases diese Unterteilung nicht so strikt vornehmen.

In dieser Arbeit werden zunächst Lehmans langjährige Forschungsarbeit und ihre bisherigen Ergebnisse vorgestellt. Der Schwerpunkt liegt dabei auf den Gesetzen der Software-Evolution. Dann werden neuere Entwicklungen dieses Forschungsgebiets am Beispiel der 11 hypothetischen Softwaresystem-Evolutionsgesetze, die Calas et al. von den bestehenden TRIZ-Evolutionsgesetzen abgeleitet und untersucht haben, betrachtet. Diese werden schließlich mit Lehmans Gesetzen verglichen und evaluiert.

1.2 Software-Evolution nach Lehman

1.2.1 Lehmans anfängliche Studien zur Software-Evolution

Daten, die Lehman 1968 zu einer neunmonatigen Studie über IBM Software-Prozesse erhielt, führten zur Untersuchung der Evolution des unter anderem untersuchten Systems OS/360. Seitdem forscht Lehman auf dem Gebiet des Softwareprozesses und der Evolution von Software. Während fast 30 Jahren Forschung formulierte Lehman bis 1996 insgesamt acht „Gesetze der Software-Evolution“ (vgl. [Leh96]). Erweitert wurden diese Gesetze durch die FEAST-Studien. Mit seiner Arbeit hat Lehman zum Verständnis der Software-Evolution beigetragen, welches der Verbesserung des Software-Prozesses dienen soll. 2003 erhielt er dafür den Stevens Award. Dieser Preis wird seit 1995 an Personen verliehen, die hervorragende Beiträge zu Literatur oder Methoden im Feld der Softwareentwicklung geleistet haben. Ein weiterer bekannter Preisträger ist Tom DeMarco.
Lehmans Forschungsziel ist die Formulierung einer Theorie der Software-Evolution.

1.2.2 Softwareklassen

Lehman kategorisiert Programme in folgende verschiedene Klassen:

E-Type-Software (Evolutionäre Software): Diese Softwaresysteme lösen Probleme oder realisieren eine Computeranwendung der realen Welt. Auf Programme dieses Typs beziehen sich Lehmans Gesetze. E-Type-Software wird ständig an ihre Domäne angepasst, unterliegt damit also einer Evolution. Diese Anpassungen werden von Entwicklern, Managern und Benutzern initiiert, deshalb basiert diese Software auf Feedback. Jede neue Version der Software basiert auf dem Quellcode der vorherigen Version. Aus diesem Grund muss bei der Erstellung jeder Version darauf geachtet werden, dass die Entwicklung von nachfolgenden Versionen möglich ist.

S-Type-Software: Diese Software löst ein formal beschriebenes Problem. Die Bezeichnung „S“ wurde ausgewählt, um die Bedeutung der Spezifikation hervorzuheben, die die geforderten Eigenschaften des Produkts definiert. Weiterhin kann das „S“ auch für statisch stehen, da Programme dieser Art im Gegensatz zu E-Type-Systemen keiner Evolution unterliegen, um die Zufriedenheit der Benutzer zu erhalten. Sobald S-Type-Software ein Problem nicht mehr löst, muss ein neues Programm zur Problemlösung

entwickelt werden. Evolution lässt sich hier insoweit feststellen, als Erfahrungen der Benutzer mit dem alten Programm in die Entwicklung des neuen mit einfließen. Für jedes neue Problem muss auch ein neues Programm geschrieben werden.

Die Unterscheidung von E- und S-Type-Programmen ist für Lehman bedeutend, da S-Type-Software per Definition stabil ist, sie unterliegt keiner Evolution und keinem Änderungsprozess. Seine Forschung widmet Lehman der Untersuchung der Evolution und der Dynamik bei der Entwicklung von E-Type-Programmen. Sie sind nicht stabil, sondern einer Evolution unterworfen, mit der die Komplexität der Wartung von Programmen dieser Art verbunden ist.

P-Type-Software: Früher [Leh80] unterteilte Lehman Programme noch nach der SPE-Klassifizierung, er führte also noch zusätzlich die Klasse der P-Programme auf. Probleme, die von P-Programmen gelöst werden, können vollständig und präzise formuliert und spezifiziert werden. Auch diese Art von Problemen entstammt der realen Welt, ein Beispiel ist ein Programm zur Wettervorhersage. Da sich die Bedingungen der realen Welt ständig ändern, müssen auch P-Programme ständig geändert werden.
Da Lehman später feststellte, dass sich P-Type-Software entweder dem S- oder dem E-Type zuordnen lässt [Leh01], betrachtete er den P-Type nicht länger.

1.2.3 Lehmans Gesetze der Software-Evolution

Die ersten drei Gesetze der Software-Evolution formulierte Lehman 1974. Sie entstanden aus der Analyse von Daten, die aus einer Studie des IBM Programmierprozesses von 1968 stammten. Zwei weitere Gesetze wurden 1980 in einem Paper vorgestellt, in dem das sechste Gesetz in einer Fußnote vermerkt war. Die beiden folgenden Gesetze wurden in Präsentationen eingeführt und zum ersten Mal in [Leh96] veröffentlicht. Alle Gesetze beziehen sich ausschließlich auf E-Type-Systeme und sind ihrem Entstehungs- bzw. Publikationsdatum nach nummeriert, wobei das Prinzip des achten Gesetzes bereits 1974 aufgefallen war, aber erst 1996 zum Gesetz formalisiert wurde.

Erstes Gesetz - Fortwährende Veränderung: *Ein E-Type-Programm, das in Gebrauch ist, muss ständig angepasst werden, sonst wird seine bereitgestellte Funktionalität von den Benutzern als immer weniger zufrieden stellend empfunden werden und es verliert seinen Nutzen.*

Dieses Gesetz beschreibt die intrinsische Eigenschaft von E-Type-Programmen, nämlich fortwährende Anpassung und Evolution (vgl. Abb. 1.1). Diese werden durch ein Ungleichgewicht zwischen der Software und ihrem Anwendungsbereich hervorgerufen. Softwareentwicklung, -installation und -betrieb ändern die Software und ihren Anwendungsbereich derart, dass Ungleichgewicht zwischen ihnen entsteht. Anders ausgedrückt löst das Softwaresystem durch seine Benutzung Veränderungen in seiner Umwelt -seinem Anwendungsbereich - aus, die dann wiederum Änderungen des Softwaresystems erfordern. Evolution entsteht durch einen von Feedback gesteuerten und kontrollierten Wartungsprozess. Immer wenn Benutzer feststellen, dass das System nicht wie gewünscht bzw. korrekt arbeitet, werden sie eine Behebung dieser

Mängel fordern. Wird dem dauernden Druck nach Evolution, d.h. Anpassung an immer neue Situationen nicht nachgegeben, so nimmt der Grad der Zufriedenheit mit der Ausführung des Systems immer weiter ab.

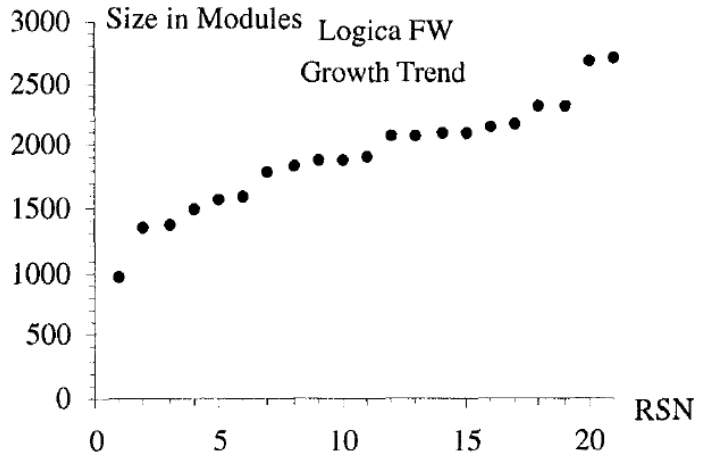


Abbildung 1.1: Wachstum des Systems Logica [Leh97]

Zweites Gesetz - Steigende Komplexität: *Durch die Weiterentwicklung eines Programms steigt seine Komplexität durch ständige Änderungen kontinuierlich an, wenn nicht explizit Gegenmaßnahmen unternommen werden, um die Komplexität zu reduzieren.*

Dieses Gesetz ist an den Zweiten Hauptsatz der Thermodynamik angelehnt, der besagt, dass die Entropie S (Maß für die Unordnung eines abgeschlossenen Systems) niemals abnimmt, eventuell aber zunimmt. Da nach dem ersten Gesetz ein Softwaresystem ständig an die wechselnde Anwendungsumgebung angepasst werden muss, steigt damit auch seine Komplexität. Da die Änderungen nämlich sukzessiv vorgenommen bzw. implementiert werden, steigen die Beziehungen und Abhängigkeiten zwischen den Systemelementen in einem unstrukturierten Muster an und führen so zu einem Anstieg der Systementropie. Wenn dieser Anwachs an Komplexität aber nicht unterbunden wird (anti-regressive Aktivitäten), wird der Aufwand, um das System zufrieden stellend zu halten (progressive Aktivitäten), immer schwieriger. Wenn allerdings mehr Aufwand zur Reduzierung der Komplexität betrieben wird, ist weniger Aufwand für funktionale Erweiterungen verfügbar. Da Ressourcen immer begrenzt sind, fällt die Wachstumsrate des Systems mit der Alterung immer ab. Das Gleichgewicht zwischen progressiven und anti-regressiven Aktivitäten ist von Feedback bestimmt.

Drittes Gesetz - Selbstregulierung: *Globale Prozesse der Programmevolution sind selbstregulierend.*

Das Team, das für die Evolution einer industriell produzierten E-Type-Software verantwortlich ist, arbeitet und handelt in einem größeren organisatorischen Kontext. Das Management dieser Organisation trägt dafür Sorge, dass die organisatorischen Regeln und Ziele jederzeit erfüllt werden. Diese gegenseitige Kontrolle, basierend auf posi-

tiven und negativen Feedback-Kontrollen (s. dazu Bedeutung von Feedback), bildet zusammen mit anderem eine „disziplinierte Dynamik“. Die genauen Wachstumskurven verschiedener Systeme unterscheiden sich zwar, aber die groben Wachstumstrends sind auffallend gleich (vgl. z.B. Abb. 1.2 und Abb. 1.3). Das bedeutet, dass die Attribute von Releases, Anzahl der Fehlermeldungen, Größe und Zeit zwischen ihnen, global gesehen fast gleich und damit zu einem gewissen Grad berechenbar sind. Dazu dient Lehman beispielsweise die Formel des Inverse Square Models, auf das hier aber nicht näher eingegangen werden soll. Die Aussage dieses Gesetzes ist insoweit erstaunlich, als die Größe eines einzelnen Releases hauptsächlich von Entscheidungen des Managements bezüglich des funktionalen Nutzens des Programms abhängt. Die Größe wird dennoch von einigen anderen Wachstumsfaktoren mit bestimmt wie Komplexitätswachstum und wachsende Leistungsfähigkeit. Wie oben schon erwähnt, spielen Feedback-Mechanismen dabei die entscheidende Rolle.

Viertes Gesetz - Bewahrung der organisatorischen Stabilität (unveränderlicher Arbeitsaufwand): *Der durchschnittliche effektive globale Arbeitsaufwand bei der Entstehung eines Systems ist unveränderlich über die Lebensdauer des Produkts betrachtet.*

Dies ist nicht auf den ersten Blick ersichtlich, da man normalerweise davon ausgehen würde, dass der Aufwand für Systemwachstum und -evolution von unternehmerischen Entscheidungen abhängt. Dies mag bis zu einem gewissen Grad zutreffen, wird aber deutlich von äußeren Kräften, zum Beispiel der Verfügbarkeit von qualifizierten Mitarbeitern, Arbeitszeitregelungen und der aktuellen Arbeitsmarktlage eingeschränkt. Auch Feedback spielt hier wieder eine entscheidende Rolle, so dass in der Praxis Stabilität des Aufwands der Produktpflege auf einem ziemlich konstanten Level erreicht wird. Die Größe des Projektteams und die Entwicklungszeit lassen sich also nicht beliebig vergrößern bzw. verkürzen.

Fünftes Gesetz - Bewahrung der Vertrautheit/des bekannten Zustands: *Während des aktiven Betriebs eines sich verändernden Programms ist der Inhalt der aufeinander folgenden Versionen statistisch unveränderlich.*

Die Vertrautheit aller Beteiligten mit den Zielen des Softwareentwicklungsprozesses spielt eine große Rolle. Je mehr Änderungen, Zusätze etc. mit einer Version verbunden sind, desto schwieriger ist für die Beteiligten die gemeinsame Zusammenarbeit und desto riskanter ist die vollständige Ermittlung und vor allem Erfüllung von Anforderungen. Dies bedeutet, dass das Einbauen von vielen neuen Funktionen eine bedeutende Fehlerquelle darstellt. Das Gesetz besagt, dass die Änderungen des Systems mit jedem Release aber global gesehen während der Lebenszeit eines Programms ungefähr konstant sind.

Sechstes Gesetz - Fortwährendes Wachstum: *Der funktionale Inhalt eines Programms muss ständig erweitert werden, um die Zufriedenheit der Benutzer während der Lebensdauer des Programms zu gewährleisten.*

Bei der Entwicklung eines neuen Software-Produkts wird zunächst ein Domänenmodell erstellt, das Anforderungen und Spezifikationen des erforderlichen Systems für einen bestimmten Anwendungsbereich festhält. Aus verschiedenen Gründen können

dort aber nicht alle gewünschten Funktionen oder Attribute festgehalten werden. Führen diese ausgelassenen Attribute irgendwann zu Schwierigkeiten bei der Benutzung des Systems, kommen Forderungen zur Veränderung des Systems auf. Diese sind hierbei gleichbedeutend mit einem Wachstum der Leistungsfähigkeit. Es werden Attribute/Funktionen implementiert, die noch nicht in der ersten Version untergebracht werden konnten. Mit der Zeit wächst ein E-Type- System also immer mehr, wieder gesteuert durch Feedback (s. Abb. 1.1).

Siebtes Gesetz - Sinkende Qualität: *Die Qualität von E-Type-Programmen wird als sinkend wahrgenommen, sofern sie nicht ständig gewartet, an die sich verändernde Anwendungsumgebung angepasst und einer ständigen Qualitätskontrolle unterworfen werden.*

Ein System, das bislang fehlerfrei lief, kann plötzlich unerwartetes und vor allem unvorhersehbares Verhalten zeigen (vgl. dazu [Leh90]). Mit dem Prinzip der Software-Unschärfe nimmt Lehman hier Bezug auf die Heisenbergsche Unschärferelation der Quantenphysik bezüglich Ort und Impuls eines Teilchens. Nach dem siebten Gesetz erhöht sich die Unbestimmtheit bezüglich der Anforderungen der Software mit der Zeit, wenn nicht als Teil der Wartungsaktivitäten erfolgreiche Versuche unternommen werden, Unstimmigkeiten zwischen der E-Type-Anwendung und der realen Anwendungsdomäne zu entdecken und zu beheben. Es existiert eine Lücke zwischen der theoretisch unbegrenzten E-Type-Anwendung, ihrer Anwendungsdomäne der realen Welt und dem endlichen System, das mit endlichen Ressourcen in endlicher Zeit entwickelt wird, um sich einer bestimmten Anwendung einer bestimmten Domäne zu widmen. Da sich die Anwendungsdomäne der realen Welt dauernd ändert und damit auch die Anforderungen an die Software, muss die Funktionalität und das Verhalten des E-Type-Systems ständig angepasst werden, um die Benutzer weiterhin zufrieden zu stellen.
Ein weiterer Punkt ist, dass je mehr die Benutzer mit dem System arbeiten und es verstehen, desto eher entwickeln sie Wünsche nach neuen Anwendungen oder Erweiterungen. Sie werden mit der Zeit immer fordernder und anspruchsvoller bezüglich der Eigenschaften eines Programms werden, weil sich ihre Kriterien bezüglich der Akzeptanz und der Zufriedenheit verändern und eventuell auch alternative Produkte verfügbar sind.

Achtes Gesetz - Feedback-System: *Der Prozess des Programmierens von E-Type-Systemen erzeugt mehrstufige Feedback-Systeme und muss als solcher behandelt werden, um erfolgreich modifiziert oder verbessert zu werden.*

Wie schon bei den einzelnen Gesetzen erwähnt, spielt Feedback bei der Entstehung von E-Type-Software eine komplexe Rolle. In Abb. 1.2 ist das Wachstum des Systems OS/360 mit der Anzahl seiner Versionen (= Sequence No.) dargestellt. Lehman misst Größe bzw. Wachstum eines Systems an der Anzahl der Module, da die Quelltextlänge (LOC) dafür aus verschiedenen Gründen nicht geeignet ist. Mit der Modulgröße ist eine bessere Abschätzung der Systemfunktionalität und -leistung möglich. Auf lange Sicht ist die Wachstumsrate eines Systems selbstregulierend. Der Welleneffekt („ripple effect“) ab Version 20, das Durchbrechen des ansonsten stetigen und gleichmäßigen

Wachstums, ist auf positive und negative Feedback-Schleifen (s. Bedeutung von Feedback) zurückzuführen. Der auffällige Sprung nach oben von Version 19 zu Version 20 zeigt laut Lehman ein von übermäßig positivem Feedback gesteuertes Wachstum.

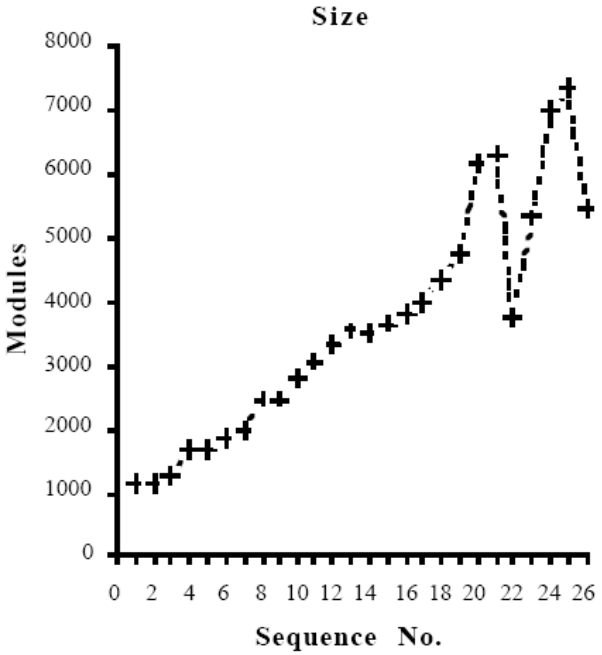


Abbildung 1.2: Wachstum des Systems OS/360 [Leh96]

1.2.4 Bedeutung von Feedback

Nachdem Lehman die Bedeutung von Feedback erkannt hatte (vgl. seine Gesetze, insbesondere das achte Gesetz), sollten nun die FEAST-Projekte das Verständnis der Gesetze erhöhen, sie detaillierter erforschen, ihre Relevanz und praktischen Auswirkungen darstellen und vor allem Wege finden, um die Gesetze ertragbringend und vorteilhaft zu nutzen (vgl. [Leh96], [Leh97]). Dazu wurden in den FEAST-Projekten die Rolle und der Einfluss von Feedback während der Evolution von E-Type-Softwaresystemen untersucht, um so den Softwareprozess verbessern zu können. Die Abkürzung FEAST steht für Feedback, Evolution And Software Technology. Der industrielle Softwareprozess ist ein Feedback-Prozess. Dies wird dadurch deutlich, dass er nicht nur die technische Entwicklung beinhaltet, sondern auch Prozesssteuerung, verschiedenste Arten von Management, Marketing, Anwenderbetreuung etc. Positives Feedback (s. dazu auch achtes Gesetz u. Abb. 1.2) erzeugt oder beschleunigt dabei Wachstum und kann so zu Instabilität führen. Negatives Feedback hingegen wirkt stabilisierend. Die FEAST-Hypothese besagt, dass evolutionäre Softwaresysteme (E-Type-Systeme) Feedback-Systeme sind, womit sie einer Dynamik und einer Tendenz zur Stabilität unterliegen. Die Eigenschaft von Feedback-Systemen liegt darin, dass ihre andauernde Änderung und Anpassung an die Anwendungsdomäne auf den Rückmeldungen und Vorgaben al-

ler Beteiligten - Entwickler, Manager und Benutzer - beruht, also auf deren Feedback (s. dazu auch E-Type-Software). Feedback-Systeme können aus vielen Schleifen und Ebenen bestehen und Feedback-Prozesse auf verschiedenen Ebenen beinhalten. Deshalb können sie sehr komplex sein, weshalb sie schwer zu verbessern sind. Wenn man nun die Mechanismen innerhalb eines solchen Systems, also die Feedback-Phänomene, versteht, dann kann man über diese den Prozess der Systementwicklung verbessern. Die FEAST-Studien versuchen deshalb das Feedback-System, das hinter dem Entwicklungsprozess der Software steht, zu begreifen und damit den Prozess selbst zu beherrschen. Dazu sollen Fragen wie „Welche echten kausalen Zusammenhänge sind innerhalb eines Softwareprojekts nachweisbar?“ und „Haben die Phänomene innerhalb eines Feedback-Systems auch wirklich Auswirkungen auf die Softwareentwicklung?“ beantwortet werden mit dem Ziel, eine wissenschaftliche Grundlage für die Handhabe von evolutionären Systemen zu schaffen.

Bis zu Beginn der FEAST-Studien 1996 gab es laut Lehman keine wirklichen Verbesserungen des Softwareentwicklungsprozesses (vgl. [Leh96]). Fortschritte in Programmiersprachen haben nur lokale Bedeutung für das Codieren selbst, das nur einen kleinen Teil der Gesamtentwicklung ausmacht. Formale Methoden mindern ihre Effektivität bzw. Anwendbarkeit durch die mathematischen Fähigkeiten, die benötigt werden, um sie effektiv zu nutzen. Die Wirksamkeit von CASE wird schließlich dadurch eingeschränkt, dass Firmen Werkzeuge nur Stück für Stück erwerben und dann nachher womöglich feststellen, dass diese zusammen nicht effektiv nutzbar sind.

FEAST/1 (1996-1998) untersucht das Wachstum des damals acht Jahre alten Bankentransaktionssystems Logica plc Fastwire (FW) anhand dessen rund 20 dokumentierten (Sub-)Versionen (s. Abb. 1.3). Wie man sieht, ähnelt diese Wachstumskurve der von OS/360 in Abb. 1.2 und zeigt deutlich, dass der Softwareprozess selbst stabilisierend ist. Diese Daten stützen somit das erste, dritte und sechste Gesetz von Lehman. Obwohl zwischen den Untersuchungen der verschiedenen Systeme OS/360 und Logica 20 Jahre liegen, liefern beide ähnliche Ergebnisse, was die Gesetze der Software-Evolution stützt.

Hier wurde mittels Blackbox-Methoden geforscht, d.h. es wurden nur die qualitativen Daten des Projekts ausgewertet, um Muster in der Evolution zu erkennen. Zur Erforschung der Feedback-Mechanismen wurden aber insgesamt drei Ansätze verwendet [Leh97]:
Beim Whitebox-Ansatz werden Modelle der Systeme geschaffen. Diese dienen dazu, die Dynamik innerhalb eines Systems zu beschreiben und Feedback-Mechanismen widerzuspiegeln, um diese und ihre Eigenschaften zu identifizieren. Der Fokus liegt dabei also auf internen Prozessstrukturen.
Mit Multiagentensystemen (erst nach Abschluss der ersten FEAST-Studie) wird der Whitebox-Ansatz fortgesetzt. Mit ihrer Hilfe können Modelle für ausgewählte Prozesse geschaffen und vorgeschlagene Verbesserungen evaluiert werden.

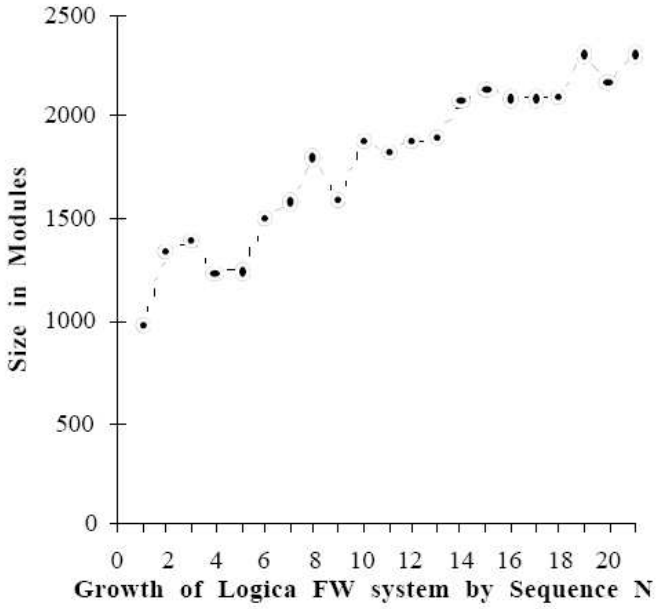


Abbildung 1.3: Wachstum des Systems Logica [Leh96]

1.2.5 Kritik an der Verwendung des Begriffs „Gesetz“

Die Verwendung des Begriffs „Gesetz“ für die Formulierung Lehmanns Forschungsergebnisse stößt nicht auf allgemeine Zustimmung. Vielmehr kommt Kritik auf, dass diese Gesetze eher Beobachtungen oder Hypothesen sind. Die Hauptargumente der Kritiker beziehen sich darauf, dass das beschriebene Verhalten sich nur auf ein bestimmtes Unternehmen, IBM, ein bestimmtes System, OS/360, und die Softwareentwicklungs-Methoden der 70erJahre beziehe, so dass keine für Gesetze notwendige Allgemeingültigkeit gegeben sei. Lehman weist diese Kritik von sich (vgl. [Leh96]) und besteht darauf, dass seine Erkenntnisse aus Sicht der Softwareentwicklung als Gesetze akzeptiert werden müssen. Er hat nicht nur die Betrachtung eines Systems in seine Studien miteinbezogen, vielmehr betrachtete er weitere Daten aus anderen Quellen.
Ein anderer Kritikpunkt besteht darin, dass die Analyse der Daten, aus der die Gesetze entstanden sind, statistisch nicht bedeutend ist. Dazu bemerkt Lehman, dass er niemals behauptet hätte, die Gesetze seien von statistischen Modellen abgeleitet. Dies sei allein deshalb schon unmöglich, da er dazu viel zu wenige Datenpunkte hatte. Die Gesetze repräsentieren eine entstehende Theorie des Software-Prozesses und der Software-Evolution und basieren auf mehreren Inputs, die Realität der Softwareentwicklung eingeschlossen. Lehman räumt aber dennoch ein, dass später noch Belege gesucht werden müssen.

1.3 Aktuelle Entwicklungen in der Forschung zur Software-Evolution

Als aktuelle Entwicklung in der Forschung zur Software-Evolution sollen nun 11 neue hypothetische Gesetze der Softwaresystem-Evolution von Calas et al. [Cal05] vorgestellt werden. Diese Gesetze basieren auf TRIZ und sind ihrem ursprünglichen physikalischen Kontext entnommen und der Softwareentwicklung angepasst worden. Mit Hilfe der Gesetze soll geklärt werden können, wie Software entwickelt werden muss, damit sie in zukünftigen Versionen möglichst wettbewerbsfähig ist.

1.3.1 Die TRIZ-Methode

Da die 11 Gesetze von TRIZ abgeleitet sind, soll diese Theorie hier zunächst vorgestellt werden (s. dazu auch www.triz-online.de). TRIZ ist russisch und steht für Teoriya Resheniya Izobretatelskih Zadach, was soviel wie „Theorie des erfinderischen Problemlösens“ bedeutet. Ziel dieser Theorie ist es, mit Hilfe von strukturiertem Erfinden mit System zu Innovationen zu gelangen. Es handelt sich dabei also um eine Methode, um Probleme kreativ zu lösen und Innovationen systematisch zu entwickeln, unter Anwendung empirischer Grundgesetze der technologischen Evolution und einiger Werkzeuge. Erfinder dieser Theorie ist Genrich Altschuller. Er fing in den 40er Jahren an zu untersuchen, wie man kreative Ideen und Lösungen durch systematische Vorgehensweise provozieren kann. Als Patentoffizier konnte er mehrere Millionen patentierte Erfindungen und Ideen nach Mustern durchsuchen, die die Erfindungszeit drastisch reduzieren würden. Daraus entstand eines der Prinzipien von TRIZ, nämlich das Finden und Lösen von Widersprüchen. TRIZ ist eine komplexe, umfangreiche und teilweise auch komplizierte Theorie, deren vier Säulen Systematik, Wissen, Analogie und Vision sind. In der Praxis werden zur Problembeschreibung und -lösung meistens nur Altschullers 39 technische Parameter oder sogar nur seine 40 innovativen Prinzipien angewandt. Calas et al. [Cal05] haben ihre Softwaresystem-Evolutionsgesetze von den technischen Evolutionsgesetzen von TRIZ abgeleitet. Die originalen Gesetze, eigentlich Prinzipien bzw. Muster, sind in Abb. 1.4 dargestellt. Sie sind eher allgemein gehalten und wurden zum Teil von Calas et al. dem Software-Kontext entsprechend angepasst. Frühere Studien über die Anwendbarkeit von TRIZ in der Softwareentwicklung deuten darauf hin, dass zumindest 19 der 40 innovativen Prinzipien direkt in der Softwareentwicklung anwendbar sind und die TRIZ Evolutionsgesetze sinnvoll für Prognosen zu Produktlinien eingesetzt werden können. Wissenschaftlich bewertet wurde bis zu der hier vorgestellten Studie aber noch nichts in dieser Hinsicht.

1.3.2 Gesetze der Softwaresystem-Evolution

(i) *Stufenweise Evolution:* Alle Systeme entwickeln sich nach S-Kurven, die die Systemleistung als eine Zeitfunktion beschreiben und verdeutlichen, welche Phasen ein System durchläuft. Abb. 1.5 zeigt S-Kurven von drei Software-Generationen.

i	Evolution in stages [5].
ii	Evolution towards increased ideality [5].
iii	Non-uniform development of system elements [5].
iv	Evolution toward increased dynamism and controllability [5].
v	Evolution towards increased complexity, then simplification through reduction [5].
vi	Evolution with matching and mismatching components [5].
vii	Evolution towards decreased human involvement [5].
viii	Law of transition to micro-level [6].
ix	Law of increasing flexibility [6].
x	Law of transition to a higher-level system [6].
xi	Law of shortening of energy flow path [6].

Abbildung 1.4: Originale technische TRIZ-Systemevolutionsgesetze [Cal05]

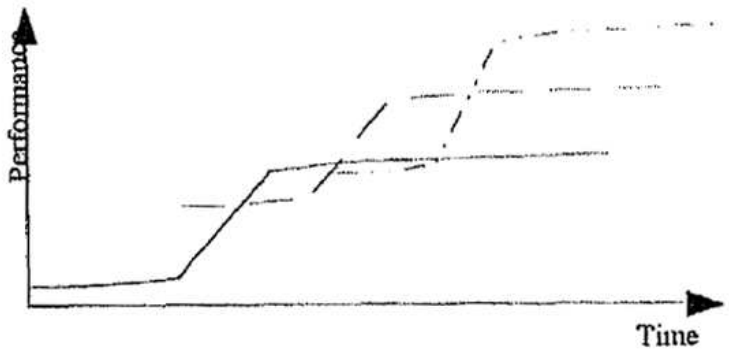


Abbildung 1.5: S-Kurven für drei Software-Generationen [Cal05]

(ii) *Evolution zu vergrößerter Idealität:* Alle Systeme besitzen Funktionen, die nützliche und schädliche Effekte haben. Im Allgemeinen entwickeln sich Systeme in Richtung eines höheren Grades an Idealität. Idealität ist hier der Quotient aus allen nützlichen Effekten und allen schädlichen Effekten, d.h. die Anzahl der nützlichen Faktoren erhöht sich in Bezug auf die Anzahl der schädlichen.

(iii) *Uneinheitliche Entwicklung der Systemteile:* Jede Komponente eines Software-Systems entwickelt sich nach seiner eigenen S-Kurve und damit unterschiedlich im Vergleich zum Rest des Systems. Wenn also eine Komponente, unabhängig von den anderen, ihre Stufe der Abnahme zu erreichen, hält sie damit die Evolution des gesamten Software-Systems zurück.

(iv) *Evolution zu erhöhter Dynamik und Steuerung:* Frühe Systemversionen beinhalten keine oder nur wenige Konfigurationsoptionen. Im Laufe der Evolution werden spätere Software-Versionen immer mehr Möglichkeiten beinhalten, um das System über Parameter zu kontrollieren. Damit erreichen die Systeme eine höhere Dynamisierung und Steuerbarkeit.

(v) *Evolution über erhöhte Komplexität zum Einfachen:* Systeme entwickeln zunächst mehr Komplexität, weil Funktionen vervielfältigt und verbessert werden und Qualität erhöht wird. Während der Evolution wird aus einem Monosystem zunächst ein Bi- oder Polysystem, um später wieder in eine neue Version zurück integriert zu werden, indem durch Vereinfachung die gleiche Funktionalität durch weniger komplexe Teile angestrebt wird (s. Abb. 1.6).

Abb. 5: Evolution über Mono- zu Polysystemen [Cal05]

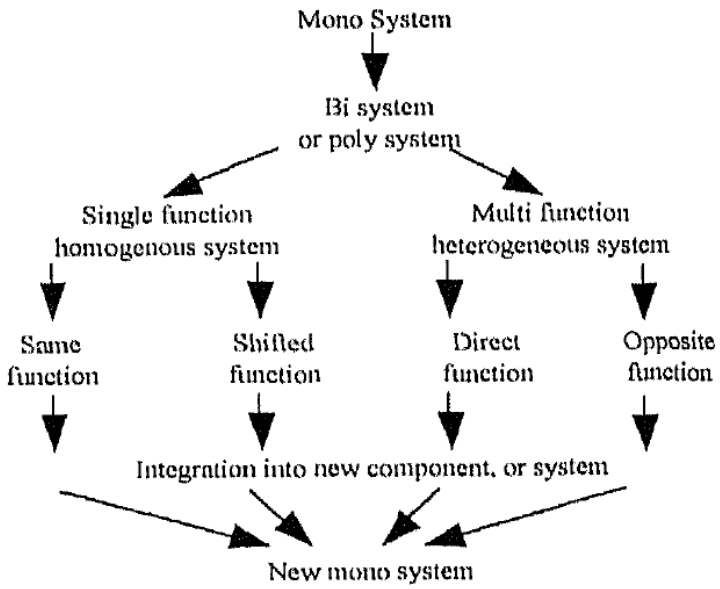


Abbildung 1.6: Evolution über Mono- zu Polysystemen [Cal05]

(vi) *Evolution mit passenden und nicht passenden Elementen:* Ein System und seine Teile entwickeln sich entweder in Richtung erhöhter Symmetrie oder in Richtung erhöhter Asymmetrie. Dies hängt davon ab, was am besten zu den Einsatzbedingungen passt. Da Calas et al. als alternative Formulierung „Evolution entweder zu erhöhter oder verringerter System-Entropie“ vorschlagen, könnte Symmetrie bzw. Asymmetrie mit der Systemkomplexität zusammenhängen.

(vii) *Evolution zu geringerer menschlicher Interaktion:* Während der Entwicklung reduzieren Systeme routinemäßige Benutzeraufgaben, so dass die Menschen sich mehr intellektuell stimulierender Arbeit widmen können. Indem die Systeme diese eigentlich menschlichen Routineaufgaben übernehmen, wird menschlicher Einsatz verringert.

(viii) *Gesetz des Übergangs zur Miniaturisierung/Mikroebene:* Systeme haben die Tendenz, sich von Makrolevel-Funktionen (z.B. sehr große Objekte oder Klassen) zu besser organisierten objektorientierten Mikrolevel-Lösungen zu entwickeln. Die Software wird also in kleinere Objekte aufgeteilt, sodass die Granularität und Modularität des Systems steigt.

(ix) *Gesetz der steigenden Flexibilität:* Systeme entwickeln immer flexiblere Strukturen und können sich so besser an Änderungen der Bedingungen oder Anforderungen anpassen. Dieses Gesetz ist in mancher Hinsicht mit Gesetz (iv) Evolution zu erhöhter Dynamik und Steuerung vergleichbar.

(x) *Gesetz des Übergangs zu einem System auf höherer Ebene:* Wenn sich Bi- oder Polysysteme zu neuen Monosystemen entwickeln (s. (v)) und die Anzahl der Hilfselemente reduziert wird, erhöht sich die Idealität, weil gemeinsame Ressourcen reduziert werden können. In der TRIZ-Methode wird unter einem Bisystem ein System verstanden, das aus zwei Objekten bzw. Systemen entstanden ist, ein Polysystem ist analog dazu aus vielen Objekten bzw. Systemen entstanden. Einige Systeme können völlig reduziert werden, wenn die Funktionen in einem Supersystem verschmelzen.

(xi) *Gesetz der Verringerung von Datenflüssen und Transformationspfaden:* Dieses Gesetz besagt, dass im Zuge von Optimierungen Datenflüsse vereinfacht werden können. Die Effekte dieses Gesetzes lassen sich bei der Evolution von Compilern beobachten, wenn ehemalige Mehrphasen-Compiler zu Einphasen-Compilern geworden sind, die effizienter sind.

Ziel der Untersuchung von Calas et al. war es nun, herauszufinden, inwieweit die vorgestellten Gesetze für die Software-Evolution gültig sind. Um diese Gesetze zu verifizieren, führten Calas et al. eine qualitative retrospektive Fallstudie durch. Dabei nutzten sie Daten aus über 25 Jahren (1975-2001) von zwei Systemen, SCADA (Ericsson’s Supervision Control and Data Acquisition software platform architecture) und die dazugehörigen Software-Produktlinien. Diese Systeme entsprechen alle Lehmans Definition von E-Type-Systemen. Jedes Gesetz wurde auf speziell dafür entwickelte Kriterien zur Unterstützung und zur Falsifizierung hin untersucht, die auf der Erfahrung in der Softwareentwicklung beruhten. Außerdem wurden die Gesetze mit denen von Lehman verglichen.

Calas et al. kommen zu dem Ergebnis, dass die 11 hypothetischen Softwaresystem-Evolutionsgesetze, die von den TRIZ Evolutionsgesetzen abgeleitet sind, gültig und auf die untersuchte Softwareentwicklungs-Domäne, die Entwicklung von Telekommunikationsmanagement-Systemen und deren Produktlinien, und auf jegliche andere Systeme ähnlicher Komplexität anwendbar sind. Sieben der elf Gesetze sind vollkommen gültig ohne eine einzige falsifizierende Beobachtung, bei vier Gesetzen ((iii), (viii), (x) und (xi)) gab es geringe Probleme, sie wurden teilweise falsifiziert, aber auch sie können dennoch angewendet werden. Bei keinem Evolutionsgesetz lag der Anteil der Fälle, die die Gesetze stützten, unter 75 %.

1.3.3 Bezug zu Lehmans Gesetzen

Lehmans quantitative Untersuchungen beziehen sich darauf, was entwickelt wird bzw. entsteht. Die Gesetze vermitteln Klarheit darüber, was in einem alternden Software-System passiert und welche Wartungsaktivitäten notwendig sind, um das System vor dem Veralten und dem Abbau zu schützen. Lehman betrachtet außerdem organisatorische Aspekte, die zu Feedback-Schleifen führen. Seine Gesetze basieren auf Ergebnissen zunächst der Studien des IBM Betriebssystems OS/360. Aus den daraus gewonnenen und weiteren Daten und Beobachtungen formulierte er sukzessiv seine Gesetze und die FEAST-Hypothesen.
Im Gegensatz dazu basieren die Evolutionsgesetze von Calas et al. auf den schon lange existierenden TRIZ-Evolutionsgesetzen und sind zum Teil zur Softwareentwicklung passend umformuliert worden. Um die Gültigkeit ihrer Gesetze zu zeigen, führen Calas et al. qualitative Studien durch und versuchen eine Antwort auf die Frage zu finden, wie man Softwareentwicklung möglichst ertragbringend beeinflussen kann. Der Fokus liegt hier eher auf der Entwicklung und der Änderung von mehreren aufeinander folgenden und neu entwickelten Versionen.

Lehmans Gesetze Fortwährende Veränderung und Fortwährendes Wachstum beschreiben einige Aspekte des Gesetzes (i) Stufenweise Evolution. Fortwährendes Wachstum hat außerdem insofern einen Bezug zu (ii) Evolution zu erhöhter Idealität, als die von Lehman untersuchte Software einem Evolutionsverlauf mit festem Kostenparameter folgte. Zu dem Gesetz (iii) Uneinheitliche Entwicklung der Systemteile existieren Gemeinsamkeiten mit Lehmans Gesetzen Fortwährende Veränderung, Steigende Komplexität und Feedback-System. Steigende Komplexität passt zu (v) Evolution über erhöhte Komplexität zum Einfachen mit dem Unterschied, dass Letzteres auch einen Übergang von Poly- zu neuen Monosystemen vorhersagt. Das Gesetz (vi) Evolution mit passenden und nicht passenden Elementen kann auch als „Evolution entweder zur Erhöhung oder zur Abnahme der System-Entropie“ formuliert werden. So ähnelt es Lehmans Gesetz Steigende Komplexität, welches für eine Abnahme der System-Entropie steht. Die TRIZ-Gesetze (i), (ii), (iii), (v) und (vi) stehen also in Einklang mit Lehmans acht Gesetzen, wobei Gesetz (iii) Uneinheitliche Entwicklung der Systemteile neue Aspekte von Lehmans zweitem Gesetz Steigende Komplexität beleuchtet. Insgesamt stellen die neuen Gesetze laut Calas et al. eine mögliche Erweiterung der Hypothesen des Forschungsgebiets der Software-Evolution dar. Es gibt Studien [Sca04],

die die Anwendbarkeit von Lehmans Gesetzen auf Open Source-Software kritisch betrachten, da die von Lehman untersuchte Software zum Teil vertraulich, also nicht der öffentlichen Inspektion zugänglich ist. Dazu schlagen Calas et al. weitere Untersuchungen der Validität ihrer Ergebnisse auch mit Open Source-Entwicklungsprojekten vor.

1.4 Fazit

Theorien über Software-Evolution haben hohes Potenzial, da sie zumindest teilweise den Bedarf nach detaillierteren Prognosen zum Langzeit-Verhalten bzw. der Langzeit-Entwicklung stillen. Außerdem würde eine solche Theorie die Möglichkeit einer proaktiven Software-Evolution bieten, indem neue Lösungen vorhergesagt und erfunden werden und technologische Evolution und Entwicklung zukünftig mit reduziertem Investitionsrisiko vorangetrieben werden.
Zu Gesetzen der Software-Evolution gibt es außer den bekannten von Lehman aber nicht viele Studien. Lehmans Daten stammen aus Untersuchungen von großen Software-Systemen [Leh03] mit um die 100K LOC oder mehr. Aufgrund der Größe der untersuchten Systeme nutzte Lehman recht banale Metriken wie die Anzahl der enthaltenen Module. Die Gesetze identifizierte er einzeln jeweils aus den erhaltenen Daten. Über die Jahre verfeinerte und erweiterte Lehman seine Gesetze immer wieder aufgrund von zusätzlich erhaltenen Daten [Leh03]. Die grundlegenden Aussagen seiner Forschungsergebnisse bleiben dabei aber immer gleich. Zum einen ist die Evolution von E-Type-Software bedeutend von Feedback bestimmt. Außerdem entwickelt sich diese Art von Software mit den Eigenschaften, dass Größe und Komplexität anwachsen und die Qualität von den Benutzern als sinkend empfunden wird. Um dieser Problematik entgegenzuwirken, ist eine ständige Änderung der Software notwendig.
Eine neuere Studie zu Gesetzen der Software-Evolution stammt von Calas et al. Sie beziehen die 11 TRIZ-Evolutionsgesetze auf Software und führten zur Verifizierung eine qualitative retrospektive Fallstudie durch. Die Ergebnisse dieser Studie deuten an, dass die Software-System Evolutionsgesetze gültig für die Langzeitentwicklung sind und in der zukünftigen Forschung zur Softwareentwicklung genutzt werden könnten. Allerdings deuten sich mit den „geringeren Problemen“ bei einigen Softwaresystem-Evolutionsgesetzen, insbesondere dem Gesetz (xi) Verringerung von Datenflüssen und Transformationspfaden, Zweifel an der vollständigen Anwendbarkeit der TRIZ-Evolutionsgesetze auf die Software-Evolution an. Hier scheint ein Überdenken notwendig, ob es sinnvoll ist, wirklich alle 11 Gesetze auf Software anwenden zu wollen, auch wenn die Aussage einiger Gesetze (z. B. (iii), (x) und (xi)) nicht ganz zu Software-Evolution zu passen scheint. Dies wird auch daran deutlich, dass Calas et al. diese Gesetze nicht näher in Bezug auf Software erläutern. Mit Sicherheit stimmen die Aussagen einiger Gesetze ((i), (ii), (v) und (vi)) insbesondere deshalb, weil sie sich größtenteils mit Lehmans Gesetzen decken. Zusammenfassend lässt sich sagen, dass die Studie von Calas et al. ein interessanter Ansatz ist, letztlich aber keine überragenden neuen Erkenntnisse zu Gesetzen der Software-Evolution bringt.
Seit fasst 40 Jahren forscht Lehman nun auf dem Gebiet der Evolution von Software. Obwohl er bislang noch nicht die erhoffte Theorie der Software-Evolution formulie-

ren bzw. vollenden konnte, werden seine Erkenntnisse dennoch weitläufig angewendet. Beispielsweise dienen die Gesetze einigen Prozessmodellen als Basis [Smi02]. Vor allem im Bereich der Open Source-Software sind aber noch Studien notwendig, da Lehman solche Systeme bisher nicht selber untersucht hat. Sicherlich werden auch diese Untersuchungen das Evolutionsphänomen bestätigen, dennoch werden seine Ergebnisse mit neuen Daten wieder modifiziert und erweitert werden müssen

Literaturverzeichnis

[Boe81] B. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[Cal05] Mankefors-Christiernin S. & Boklund A. Calas, G. *A Case Study Evaluation of 11 Hypothetical Software System Evolution Laws*. IASTED Conf. on Software Engineering, 2005.

[Leh80] M. M. Lehman. *Programs, Life Cycles and Laws of Software Evolution*. Proceedings of the IEEE, 1980.

[Leh90] M. M. Lehman. *Software Uncertainty and the Role of CASE in its Minimisation and Control*. IEEE Computer Aided Software Engineering, 1990.

[Leh96] M. M. Lehman. *Laws of Software Evolution Revisited*. Springer-Verlag, Berlin, 1996.

[Leh97] Ramil J. F. Wernick P. D. Perry D. E. & Turski W. M. Lehman, M. M. *Metrics and Laws of Software Evolution - The Nineties View*. IEEE Computer Society Press, 1997.

[Leh01] J. F. Lehman, M. M. & Ramil. *Rules and Tools for Software Evolution Planning and Management*. Annals of Software Engineering, 2001.

[Leh03] J. F. Lehman, M. M. & Ramil. *Software evolution — Background, theory, practice*. Inf. Process. Lett, 2003.

[Sca04] W. Scacchi. *Understanding Open Source Software Evolution*. http://www.ics.uci.edu/w̃scacchi/Papers/New/Understanding-OSS-Evolution.pdf, 2004.

[Smi02] Ramil J. F. Smith, N. *Qualitative Simulation of Software Evolution Processes*. WESS, 2002.

[Som01] I. Sommerville. *Software Engineering*. Pearson Studium, 2001.

Kapitel 2

Reverse Engineering

Tobias Campmann

Inhaltsverzeichnis

2.1 Einführung

In der heutigen Softwareenwicklung nehmen Problemanalyse und Entwurf der Software ein gewichtige Rolle im Entwicklungsprozess ein. Die hieraus resultierenden Anforderungsspezifikationen, UML-Diagramme, Entwicklungspläne, usw. (*hier:* Dokumentationen) liefern eine Modell-Sicht auf das zu implementierende System. Was aber, falls diese Dokumente unvollständig oder veraltet sind, oder ganz fehlen? Genau hier setzt Reverse Engineering an und erzeugt entgegen der üblichen Entwicklungsrichtung Modellierungen aus dem fertigen Produkt. Dies kann wichtig sein, falls das System im Laufe des Entwicklungsprozesses degeneriert ist, oder ein altes System erweitert oder überarbeitet werden soll.

Für Software-Projekte einer gewissen Größe ist die Erstellung guter Dokumentationen unumgänglich, gerade wenn sie über einen längeren Zeitraum mit einer Gruppe von Entwicklern laufen. Deswegen wird üblicherweise vor der Implementierung eine Planungsphase durchgeführt, die die Anforderungen des Kunden an das System spezifiziert, Entwurfsentscheidungen trifft und die Software-Architektur entwirft. Hieraus entstehen dann Aufzeichnungen über die Ziele des Projektes, das Vorgehen bei der Entwicklung und die innere Funktionsweise der Software, eventuell auch über die Handhabung des Produkts. Sind diese Dokumente nicht mehr oder nur noch teilweise vorhanden, wird es sehr schwierig das System nur noch anhand des Sourcecodes zu warten oder weiter zu entwickeln. Reverse Engineering versucht nun aus dem vorliegenden Source- bzw. Bytecode Dokumentationen, wie Anforderungsdefinitionen, Designentscheidungen wie Software-Patterns oder Entwurfsspezifikationen, z.B. UML-Diagramme, zurückzugewinnen.

Zuerst gibt es einen Überblick über Reverse Engineering. Im Folgenden werden im Detail mögliche Gründe und Szenarien sowie einige Techniken und Ansätze für Reverse Engineering vorgestellt. Hierbei liegt ein besonderer Fokus auf Reverse Engineering objektorientierten Codes. Daraufhin wird Analyse des Codes betrachtet, in Abschnitt 2.3 wird dann die Auswertung der gewonnenen Daten diskutiert. Desweiteren wird ein Ausblick in die Praxis gegeben und einige Tools präsentiert. Schließlich gibt es noch einen Ausblick in die Praxis und ein Fazit.
Das Dekompilieren von Bytecode wird hier nicht betrachtet.

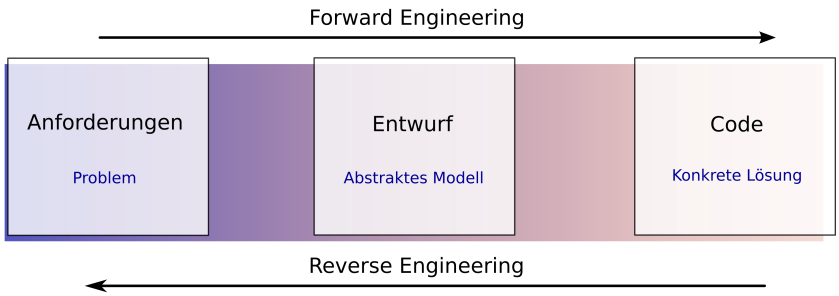


Abbildung 2.1: Übersicht Forward/Reverse Engineering

2.1.1 Reverse Engineering

Der Begriff Reverse Engineering stammt eigentlich aus den Ingenieurwissenschaften. Es ist der Teil der Entwicklung, der entgegen der normalen Vorgehensweise verläuft, meint im Maschinenbau z.B. das Zerlegen eines Bauteils, Analysieren der Funktionen und Rückerstellung eines Bauplans. Die traditionelle Entwicklungsmethode Forward Engineering ist folgendermaßen definiert:

> *Forward Engineering* erstellt aus der Analyse der zu lösenden Probleme abstrakte Modelle und überführt diese stückweise in eine konkrete Lösung. [Lan03]

Reverse Engineering bezeichnet dann genau den umgekehrten Prozess, also die Gewinnung abstrakter Modelle aus einer konkreten Lösung (Abb. 2.1). Dies deckt sich auch mit der Definition aus dem SWEBOK [Soc]:

> Reverse engineering is the process of analyzing software to identify the softwares components and their interrelationships and to create representations of the software in another form or at higher levels of abstraction

Software Reverse Engineering gliedert sich somit in den Prozess der Softwarewartung/-erweiterung ein. Durch die Analyse bereits vorliegenden Codes (bzw. Bytecodes) entsteht dabei ein besseres Verständnis der Software-Architektur; Visualisierungen liefern eine bessere Übersicht über das System, als das reine Lesen des Sourcecodes es könnte. Hierbei werden die wichtigen Komponenten (*hier*: Software-Artefakte) identifiziert und ihre Beziehungen zueinander gefunden
Ein Aspekt, der häufig zum Reverse Engineering hinzugezählt wird, ist das Dekompilieren von Bytecode bzw. assembliertem Code. Dekompiliertem Code fehlt üblicherweise auch jegliche Dokumentation, womit Dekompilieren als eine Art einleitende Maßnahme des Reverse Engineering gesehen werden kann. Im Folgenden liegt nun ausdrücklich der Fokus auf der Rückgewinnung von Entwürfen und Anforderungen; Dekompilierung ist eher eine Disziplin des Compilerbaus.

2.1.2 Anwendungsgebiete

Warum also Reverse Engineering?
Falls alle Software sauber entworfen, dokumentiert und gepflegt würde, wäre Reverse Engineering vermutlich unnötig. Die Erfahrung aus der Praxis zeigt aber, dass dies nahezu niemals der Fall ist. Häufig fehlen z.B. beim Warten eines Legacy-Systems die Anforderungsspezifikationen, die Dokumentationen und Programmentwürfe sind nicht mehr aktuell oder fehlerhaft. Dies kann mehrere Gründe haben:

- Oft ist es innerhalb der Laufzeit der Projektes nicht mehr möglich die Spezifikationen sauber zu erstellen, bzw. auf dem neuesten Stand zu halten.
- Aus finanziellen Gründen werden diese gar nicht erst/nicht ausreichend erstellt.
- Das System ist auf Grund sich ändernder Anforderungen degeneriert.
- Erweiterungen und Bugfixes haben den Code gegenüber den Dokumentationen inkonsistent gemacht.

Reverse Engineering versucht nun diese Dokumente wiederherzustellen, bzw. äquivalente Visualisierungen zu erzeugen. Es gibt viele Szenarien, in denen eine solche Aufbereitung sehr gewinnbringend ist. [EJC90]

- Im Re-Engineering-Prozess spielt das Reverse Engineering eine entscheidende Rolle. Siehe dazu Abschnitt 2.1.3.
- Beim Arbeiten mit Legacy-Systemen ist es notwendig, das System erst einmal zu verstehen, bevor man in der Lage ist Änderungen durchzuführen. Ältere Programme unterscheiden sich häufig dramatisch von heute entwickeltem (objektorientiertem) Code, besonders in der Programmstruktur (z.B. Spaghetti-Code).
- Die eigentlichen Entwickler des Systems sind möglicherweise nicht mehr zu erreichen, und man versteht das System nicht in seiner Gänze. Möglicherweise wurden überholte Programmiersprachen und -tools benutzt.
- Das System wurde schlecht gewartet: Bugfixes lenken das Programm immer weiter weg vom Modell, Workarounds resultieren in mehrfach kopiertem Code. Oft wird der Code von schlecht eingewiesenen Programmierern gewartet, die die Architektur nicht ganz überschauen.
- Man möchte die Software auf eine andere Hardwarearchitektur portieren. Hierfür ist die Analyse des alten Codes vorteilhaft zur Erstellung und Bewertung des neuen.
- Es soll fremde Software geknackt werden (der vorliegende Code ist vermutlich durch Dekompilierung entstanden). Dieser Ansatz ist eigentlich ein illegaler Versuch der Industriespionage und wird eingesetzt um eine erfolgreiche Software günstig zu imitieren, ohne sich dabei nur auf das (legale) Blackbox-Testen zu verlassen. Dies ist auch der Grund, warum Reverse Engineering etwas in Verruf geraten ist, da oft nicht zwischen legitimen und illegitimen Methoden unterschieden wird.
- Malware und Viren sollen bekämpft werden. Auch hier ist der Sourcecode nicht der eigene. Reverse Engineering kann hier wichtige Hinweise zur inneren Struktur und Organisation solcher Malware geben, um schneller heraus zu finden, wie sie welche Schwachstellen angreift.

Man könnte sich nun fragen, ob es bei solchen Voraussetzungen nicht einfacher wäre, das System von Grund auf neu zu implementieren. Ein Punkt der dagegen spricht ist, dass *das vorhandene System funktioniert*. Eine Reimplementation ist häufig teurer, da dies viel Zeit in Anspruch nimmt. Es müsste hierfür das ganze System neu aufgezogen werden: Das Sammeln von Anforderungen und Benutzerwünschen, Entwerfen der Architektur und Implementierung. Dabei werden dieselben Entwicklungsschritte erneut durchlaufen, was auch bedeutet, dass dieselben Fehler nochmal gemacht werden können. Teilweise sind domänenspezifische Programmentscheidungen fest in den Code eingebaut (hardcoded), und die Anwender müssten in dei neue Benutzeroberfläche eingewiesen werden. Daher versucht man oft, den bestehenden Code wieder zu verwenden.

2.1.3 Reverse Engineering im Software Lifecycle

Reverse Engineering ist ein Teil des Reengineering Prozesses. Üblicherweise wird vorliegende Software analysiert, anschließend reorganisiert (Refactoring) und erweitert. Darüber hinaus sollte Reverse Engineering auch in einer normaleren Softwareevolution genutzt werden. Wartungen, Erweiterungen, Workarounds, etc. sorgen für ständige Inkonsistenz zwischen Entwurf und System. Durch stetes Reverse Engineering können (auch kleinere) Änderungen wieder ins Modell übernommen werden, dadurch bleibt das Modell konsistent und bietet so den neuen Ausgangspunkt für Weiterentwicklung oder Wiederverwendung, bzw. dient als stimmige Dokumentation des Systems. Daraus ergibt sich die Idee des sog. *Round-Trip-Engineering* (s. Abb. 2.2). Hierbei wird Forward Engineering betrieben und nach teilweiser Implementierung Reverse Engineering benutzt, um danach wieder konsistent vorwärts zu entwickeln. [Lan99]
Auch während der Fehlersuche und Wartung kann Reverse Engineering eine große Hilfe sein. Zum Beispiel ist es möglich während des Debugprozesses den ausgeführten Code zu verfolgen (Tracing) und so den Fehlercode zu finden. Außerdem kann man Software-Metriken anwenden und so z.B. toten Code finden oder besonders Fehler anfällige Codefragmente identifizieren.

2.1.4 Objekt-Orientierung und Reverse-Engineering

Der heute weit verbreitete Ansatz der objektorientierten Programmierung hat auch Auswirkungen aufs Reverse Engineering. Die Legacy-Systeme der Zukunft werden nicht mehr prozedural aufgebaut sein, sondern objektorientiert. Dies liefert neue Chancen, aber auch Probleme. Üblicherweise ist OO-Code besser strukturiert, macht durch Mechanismen wie Polymorphie das Erstellen von doppeltem (kopiertem) Code unnötig und liefert durch die Vererbungsstruktur bereits eine Hierarchisierung des Codes. Ab einer gewissen Größe verschließt sich aber der Code dem einfachen Verständnis: Unzählige Querverweise, tiefe Vererbungsstrukturen (Verantwortlichkeiten können durch die Klassenstruktur weitergereicht werden) und lange Methodenaufrufsketten sorgen für eine hohe Komplexität. Polymorphie macht es sogar oft schwer den tatsächlich ausgeführten Code zu finden, weil die Laufzeitinformationen des dynamischen Bin-

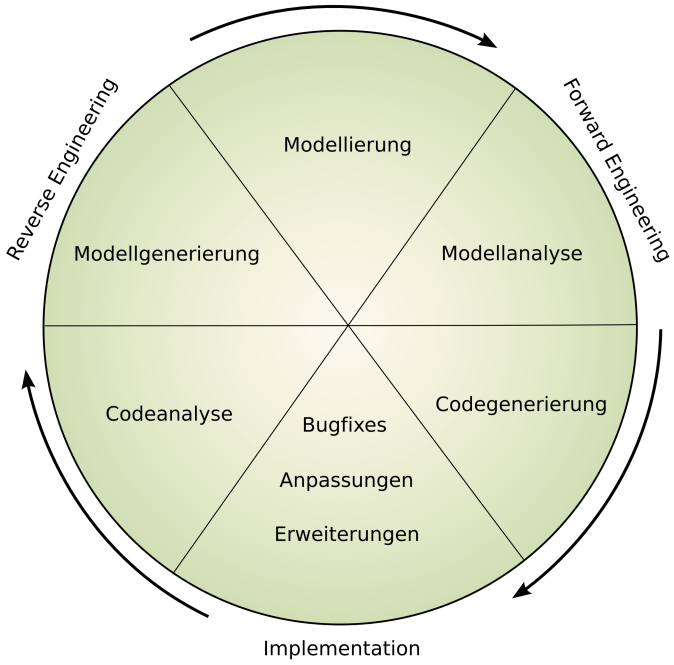


Abbildung 2.2: Round-Trip Engineering

dens fehlen. Darüber hinaus ist es durchaus notwendig zu wissen, wann zur Laufzeit ein Objekt erstellt, zugewiesen oder von der Garbage Collection eingesammelt wird. Für Reverse Engineering bedeutet das eine größere Herausforderung, da auf die Eigenheiten von objektorientiertem Code eingegangen werden muss. Es sollten Design-Entscheidungen wiederentdeckt und Use-Case-, Sequenz- und Klassen-Diagramme wiedergewonnen werden. Außerdem muss die Polymorphie aufgelöst bzw. überwacht werden, um die Analyse vollständig zu machen, also ist Tracen des wirklich ausgeführten Codes und der beteiligten Klassen, Methoden und Objekte nötig. [Lan03]

2.2 Analyse und Datenerfassung

Im Folgenden werden etwas konkretere Ansätze aus dem Bereich des Reverse Engineering diskutiert. Zuerst soll es um das Finden und Erkennen der auszuwertenden Daten gehen; in Abschnitt 2.3 wird dann die Auswertung der Daten diskutiert. Hierbei ist zuerst generell zwischen *statischer* und *dynamischer* Modellierung zu unterscheiden.

2.2.1 Statische Modellierung

Statische Modellierung ist die klassische Reverse Engineering Methode, die nur den vorliegende Quellcode analysiert. Hierzu können im wesentlichen syntaxbasierte Par-

ser benutzt werden, die die vorhandenen Software-Artefakte erkennen. Software-Artefakte können hier Funktionen und Methoden, Module und Subsysteme, Variablen und Typen sein. In objektorientierten Programmiersprachen kommen noch Klassen, Packages und Interfaces hinzu. Außerdem können Beziehung zwischen dieses Artefakten gewonnen werden, also Variablenzugriffe, Funktionsaufrufe, Vererbungen, usw.
Hierdurch kann schon ein sehr guter Einblick in die Software Architektur entstehen, falls die Software entsprechend strukturiert ist. Auf dieser Ebene lassen sich Variablenzugriffe verfolgen und so Zuständigkeiten oder Fehler finden. Im OO-Fall zum Beispiel lassen sich bereits Klassendiagramme einzelner Subsysteme erstellen, Vererbungen und Zugriffe können in Beziehung gesetzt, verstanden und auf Richtigkeit geprüft werden. Software-Patterns sind hieraus auch erkennbar, so können Designentscheidungen direkt ins Modell zurück überführt werden.
Jedoch ergeben sich üblicherweise schon bei statischer Modellierung eine Unmenge von Artefakten, daher ist es nötig bei der Auswertung der Daten über Teile zu abstrahieren, bzw. unwichtige Teile auszublenden, um ein aussagekräftiges Diagramm zu erhalten. Softwaremetriken und Komplexitätsmaße lassen sich auf dem Quellcode auch bereits anwenden, so lässt sich das System auf Eigenschaften wie Kohäsion und Kopplung prüfen; doppelter oder toter Code kann gefunden werden.
Gerade im Hinblick auf objektorientierte Programmierung ist statische Information oft nicht ausreichend, da diese generelle ein dynamische Struktur hat: Objekterstellung, Garbage Collection und Polymorphie werden von der statischen Analyse nicht erfasst. So werden zum Beispiel auch Aufrufe des Default Constructors nicht unbedingt ausgewertet, da nicht explizit im Code erwähnt. [Sys00]

2.2.2 Dynamische Modellierung

Dynamische Modellierung untersucht nun das Laufzeitverhalten des Systems. Softwareartefakte sind hier Objekte (inkl. deren Erstellung und Dekonstruktion), Aufrufketten (event trace information), etc. Hieraus können Informationen über Codeüberdeckung, Speicherausnutzung und -löcher, sowie Sequenzinformationen (Gleichzeitigkeiten und Abläufe) gewonnen werden. Benutzt werden kann dafür neben Debuggern, Profilern oder Event Recordern auch eine Auswertung der Instruktionen der Virtual Machine (falls vorhanden). Das Tracing der Softwareartefakte ermöglicht das Auflösen der dynamischen Bindung zur Laufzeit, um genau zu sagen, welcher Code ausgeführt wurde, was durch die Polymorphie nur anhand des Quelltextes nicht möglich ist. Durch das Überwachen der Objekterstellungen und Dekonstruktionen können wichtige Informationen zum Beispiel über interne Kommunikation oder Datenverwaltung gesammelt werden.
Es resultieren einige Methoden, die im weitesten Sinne zum Reverse Engineering gehören, die sich dynamisch verhalten:

- Tracing von Input – Verfolgen der gemachten Eingabe durch den Quellcode. Wo wird wie damit verfahren? Im wesentlichen ein Teil des Debug-Prozesses kann dies auch entscheidende Aussagen über das System treffen. So kann z.B. ein Sequenzdiagramms erstellt werden, das genau die einzelnen Aufrufe und Pa-

rameterübergaben aufzeigt. Ein einzelner Datensatz kann also durch das ganze System verfolgt werden.

- Es können White-Box-Tests durchgeführt werden, wodurch zum Beispiel Codeüberdeckungsinformationen gesammelt werden können. Hierdurch können fehleranfällige Hotspots identifiziert werden.

- Aufspüren von Memory Leaks – Falls sog. Speicherlöcher gefunden werden, können diese direkt auf gezielte Stellen im Quellcode zurückgeführt werden (z.B. zur Objektkonstruktion).

Auch dynamisch kann so eine relativ große Menge Information erfasst werden, was hohe Ansprüche an eine entsprechende Auswertung, bzw. Visualisierung verlangt.

2.2.3 Hybride Ansätze

Natürlich ist es auch möglich statische und dynamische Modelle zu mischen. Dies ist auch durchaus sinnvoll, da beide Typen andere Informationen aus dem System extrahieren. Dynamische Abstraktionen sind typischerweise Verhaltensmuster oder Anwendungsfälle, während statische Abstraktionen eher Subsysteme spezifizieren, so [Sys99]. Dies liefert eine sinnvolle Übersicht über allgemeine Zusammenhänge, es können z.B. statische Diagramme mit dynamischer Information verfeinert werden, oder umgekehrt. Bei Sequenzdiagrammen könnten Objekte anhand der Vererbungsstruktur zusammengefasst werden; in Klassendiagramme könnten Information aus den Softwaremetriken, wie die Anzahl der Methodenaufrufe, eingewoben werden.
So sinnvoll dies auch erscheint, macht dies den Auswertungsprozess noch ein ganzes Stück schwerer, da noch mehr Information visualisiert werden soll, ohne dabei die Übersicht zu zerstören.

2.3 Datenauswertung und Visualisierung

Die extrahierten Daten bestehen oft aus einer immensen Größe detaillierter Information über Software Artefakte aus niedrigster Ebene. Ein Reverse Engineering von FUJABA resultierte beispielsweise in 25854 Knoten. [Sys99] Die Frage ist nun, wie solche Datenmengen effizient visualisiert bzw. ausgewertet werden können. Hierbei stellt sich auch die Frage, ob alle erfassten Daten auf einmal aufgearbeitet werden, also eine einzige Sicht erzeugt werden soll, oder mehrere Sichten erzeugt werden sollen. Selbst mit sehr starken Abstraktionen und Filterungen wird ein einziger View sehr schnell unübersichtlich, kann aber Zusammenhänge (gerade zwischen dynamischen und statischen Artefakten) verdeutlichen. Mehrere Sichten sind allerdings auch bei der (Vorwärts-) Entwicklung üblich, um unterschiedliche Aspekte voneinander getrennt behandeln zu können.

2.3.1 Filterungen & Abstraktionen

Die Resultierenden Graphen sind meist sehr groß und komplex, daher sind Formen der Filterung bzw. Abstraktionen nötig, um eine Übersicht zu bekommen. Es gibt viele Möglichkeiten diese Information zu abstrahieren bzw. (momentan) unwichtige Informationen zu Filtern.

- Die simpelste Möglichkeit ist das einfache Ausblenden/Filtern einiger Artefakte oder Teilsysteme. Dadurch wird ein Fokus auf einen bestimmten Ausschnitt des Systems gelegt. Alle Beziehungen, nach außen (also außerhalb des Scopes), werden entweder abgeschnitten oder zusammengefasst. So lassen sich z.B. nur die Interna eines Packages und den internen Beziehungen betrachten.
- *Horizontale Abstraktion* bedeutet das Zusammenführen einiger Softwareartefakte auf gleicher Ebene. So können z.B. mehrere Objekte einer Klasse nur als Klassenrepräsentation visualisiert werden, falls eher die Interaktion zwischen den Klassen, als den Objekten in Betracht gezogen werden muss.
- *Vertikale Abstraktion* bedeutet das 'Zusammenklappen' einiger Softwareartefakte auf unterschiedlichen Ebenen zu abstrakteren Oberknoten. Z.B. Klassen zu einer Superklasse, Methoden zu einer abstrakten Methode,... Auch können Methodenaufrufe innerhalb eine Klasse 'verschwiegen' werden, falls nur die Kommunikation zwischen Klassen interessant ist.
- Beim Tracing kann das Setzen von Breakpoints helfen die Datenmenge zu verringern. So kann das Tracing praktisch zu bestimmten Zeiten (an Breakpoints) gestartet bzw. gestoppt werden. So können nur Teilausschnitte der Event Traces betrachtet werden, also nur die Zeit vor/nach/zwischen den Breakpoints. Dies erlaubt das Erstellen mehrerer Szenarien, als Grundlage für Sequenz- oder Use-Case-Diagramme.
- Es ist auch möglich nur bestimmte Exceptions zu tracen. So können bestimmte Fehler genauer analysiert werden.

2.3.2 Diagramme und Graphsichten

Die entstehenden Daten sind im wesentlichen Entitäten und Relationen, weswegen sich eine Graph-Visualisierung anbietet. Ziel ist im Allgemeinen die Zurückführung auf UML-Diagramme, von denen die meisten (z.B. Klassen-, Aktivitäts- oder Use-Case-Diagramme) ebenfalls eine Graph-Struktur aufweisen. Dies ist auch genau die Idee, die Reverse Engineering so komfortabel macht: UML-Diagramme sind ebenfalls der Standard bei der initialen Entwicklung und Modellierung (Forward Engineering), d.h. das Resultat ist in der selben Sprache, wie der Ausgangspunkt spezifiziert. Daraus entstehen zwei Abbildungen (Modell und Code), die (jederzeit) ineinander überführt werden können.
Im Folgenden werden einige Diagrammarten und ihre Möglichkeiten gezeigt.

- Klassendiagramme: Die typischste und in der Praxis am besten erforschte Visualisierung. Klassendiagramme können durch einfaches parsen der Vererbungs- und Enthaltenseinsbeziehungen erzeugt werden, geben schnell eine Übersicht und liefern eine Grundlage zum Refactoring. In der entstehenden Sicht können Klassen abstrahiert, also visuell zusammengefasst werden (was eventuell eine Grundlage für neue Oberklassen bietet) oder Klassen in Packages angeordnet werden. Auch kann man bereits definierte Software Patterns wiederfinden und als Stereotypen den Klassen zuordnen, also konkrete Designentscheidungen mit abstrakten Mustern identifizieren.

- Sequenzdiagramme: Als Auswertung von Tracing Information bestimmter Methoden oder Klassen bietet sich ein Sequenzdiagramm an. Hier können Methodenaufrufe, Objekterstellungen und -dekonstruktionen in einen sequentiellen (zeitlichen) Rahmen gestellt werden.

- Use-Case Diagramme: Für (per Hand) identifizierte Use-Cases und deren Akteure (Klassen) können die Kollaborationen zwischen den Methoden analysiert werden.

- Weitere Diagramme: Objektdiagramme, Informationsflüsse, Aktivitätsdiagramme...

Darüber hinaus existiert die Möglichkeit zusätzlich erhobene Daten in die Standarddiagramme einzubauen, z.B. Daten aus Softwaremetriken oder Laufzeitinformationen. Dafür werden die einzelnen Knoten skaliert eingefärbt bzw. der Größe und Position eine zusätzliche Semantik zugeordnet, um noch mehr Informationen in das Diagramm einzuflechten, falls z.B. die Anzahl der Methodenaufrufe in einem Klassendiagramm angezeigt werden soll. [Lan99]

2.3.3 Andere Ansätze

Nun noch kurz zu einigen anderen (eher exotischen) Ansätzen der Datenauswertung:

- [ERSS02] versucht die Rückgewinnung von Anforderungsspezifikationen durch Zusammenfassen häufig auftretender Pfade durch das System. Hierbei wird anhand von Interactive Pattern Mining nach wiederauftretenden Mustern in der Programmausführung gesucht, über die Pfade abstrahiert und Rückschlüsse auf die Intentionen der einzelnen Programmteile gemacht. Grundidee ist die Annahme, dass oft verwendete Abläufe in Legacy-Systemen deren Anforderungen reflektieren.

- Auswertung als Entscheidungstabelle: Hierbei werden insbesondere große Auswahl- und Entscheidungsblöcke nach zu Use-Cases getrennt zu Entscheidungstabellen ausgewertet. Dies ist insbesondere für betriebswirtschaftliche Anwendungen interessant, da hier nicht schwierige Algorithmen, sondern komplexe

Fallunterscheidungen auftreten. Szenarien in einer Entscheidungstabelle sind hier vor allem für den nichtinformatischen Experten besser zu lesen. [Sch04]

- Erfassen der Information als logische Daten, z.B. zu Regeln in Prolog. Hierbei werden Meta-Modelle für Softwareartefakte und deren Beziehungen gebildet. Die interne Datenstruktur wird dann anhand von Queries befragt und so ausgewertet. Durch den Anschluß an ein Grafik-Framework werden daraus letztendlich Diagramme generiert. [RD99]

Der üblichste Ansatz ist wie schon erwähnt die Visualisierung als graphähnliches Diagramm, so werden gerade in kommerziellen Tools bisher oft nur statische Informationen ausgewertet. Doch auch die Auswertung mit anderen als nur graphentheoretischen Mitteln scheint einige vielversprechende Ergebnisse zu bringen, gerade da hinter dem Sourcecode ja meist ein weit größeres Meta-Modell mit komplexerer Semantik steckt.

2.4 Reverse Engineering in der Praxis

Wie bereits angesprochen gibt es zahlreiche Szenarien um ein Reverse Engineering durchzuführen. In der Industrie, wie auch in gewissem Maße der Forschung sind effiziente Methoden des Reverse Engineering daher dringend benötigt. Da das Feld noch ein relativ junges ist, gibt es eine nicht unerhebliche Diskrepanz zwischen dem Stand der Forschung und dem Stand der funktionierenden Tools. Wie präsentiert, gibt es mehrere Ansätze und Methoden, von denen sich bisher jedoch fast nur statische durchgesetzt haben.
In der Forschung werden zu theoretischen Zwecken oft eigenimplementierte Prototypen eingesetzt, die die Kernfunktionalitäten der neuen Ansätze testen sollen. Diese Tools sind im industriellen Gebrauch nur schwer einsetzbar, da ihnen oft ein angemessenes Benutzerinterface fehlt, sie oft veraltet sind und Bugs beinhalten. Jedoch liefern sie eine Einsicht auch in dynamische Vorgehensweisen. `RiGI` zum Beispiel ermöglicht das Visualisieren und Browsen statischer und dynamischer Software-Artefakte in einer baumähnlichen Ansicht. Der Nachfolger Creole lässt sich sogar in Eclipse einbinden (Abb. 2.3). Es können Teile ausgeblendet oder zu einem Knoten zusammengezogen werden. Aus diesen gefilterten Sichten heraus lassen dynamische Analysen starten, z.B. eine Exception tracen. So entsteht eine kombinierte Sicht aus statischer und dynamischer Information.

Weitere Tools, die (hauptsächlich an Universitäten entstanden sind und zu Forschungszwecken eingesetzt werden sind Ovation, Scene, CodeCrawler, JInsight und FUJABA. Letzteres erfreut sich hoher Beliebtheit gerade bei privaten oder kleineren Projekte, da es kostenfrei ist, viele Funktionalitäten fürs Forward und Reverse Engineering bereitstellt und auch in Punkto Bedienerfreundlichkeit relativ ausgereift ist.

Es gibt neben diesen auch noch eine Reihe kommerzieller Tools, die Reverse Engineering Operationen anbieten. Die meisten beschränken sich dabei auf das erstellen

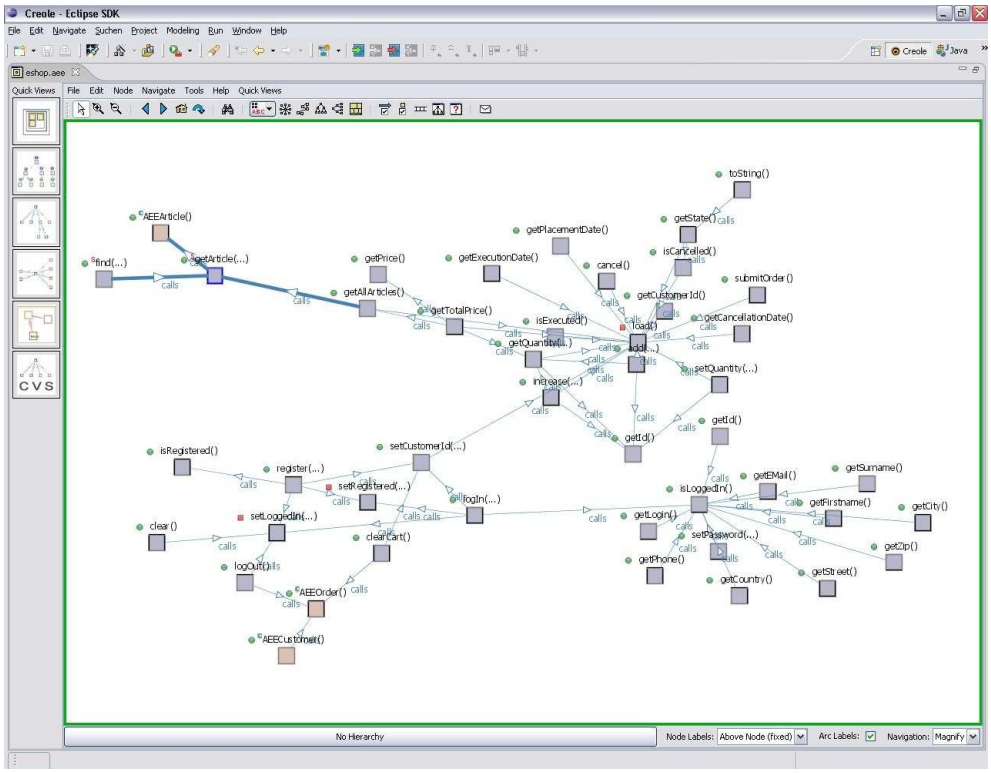


Abbildung 2.3: Screenshot: Call-Graph in `Creole`

von UML-Klassen Diagrammen aus dem Quellcode, also statischen Methoden. Programme wie Microsoft `Visio`, Rational `Rose` oder Altova `UModel` schaffen dies dabei recht effizient und schnell und liefert gerade bei objektorientierter Software bereits eine erhebliche Einsicht in den Quellcode. Viele dieser Programme lassen sich als Plug-in ins `Eclipse` Framework einbauen, im Bereich des Java Programmierens fast die standard Entwicklungsumgebung. `Visual Paradigm` oder Omondo `EclipseUML` profitieren dabei unter anderem von dessen Refactoring Funktionen, so ist es einfach möglich auf grafischer Ebene Klassen und Methoden zu verschieben, oder umzubenennen. Außerdem können hier auch Software-Metriken eingebunden werden, und so das System statistisch analysiert werden. Eingebettet in eine Entwicklungsumgebung ist dadurch auch Round-Trip-Engineering möglich, da der reverse engineerte Code nun am Modell bearbeitet werden kann, anschließend die Änderungen in den Code übernommen werden, usw.

Borlands `Together` liefert schon die Möglichkeit die generierten Klassendiagramme zu analysieren, so können Softwarepatterns, wie das Singleton- oder Visitor-Muster, automatisch gefunden und den Klassen als Stereotypen zugewiesen werden. Dadurch werden bewusste Designentscheidungen wiedergewonnen. Together beherrscht sogar dynamische RE-Methoden: Es können Sequenzdiagramme erstellt werden, also in einer Art Tracing Aufrufe zwischen Methoden erkennt und darstellt (Abb. 2.4). Durch die Aufruftiefe parametrisiert lassen sich dann noch einzelne Klassen aus dem Dia-

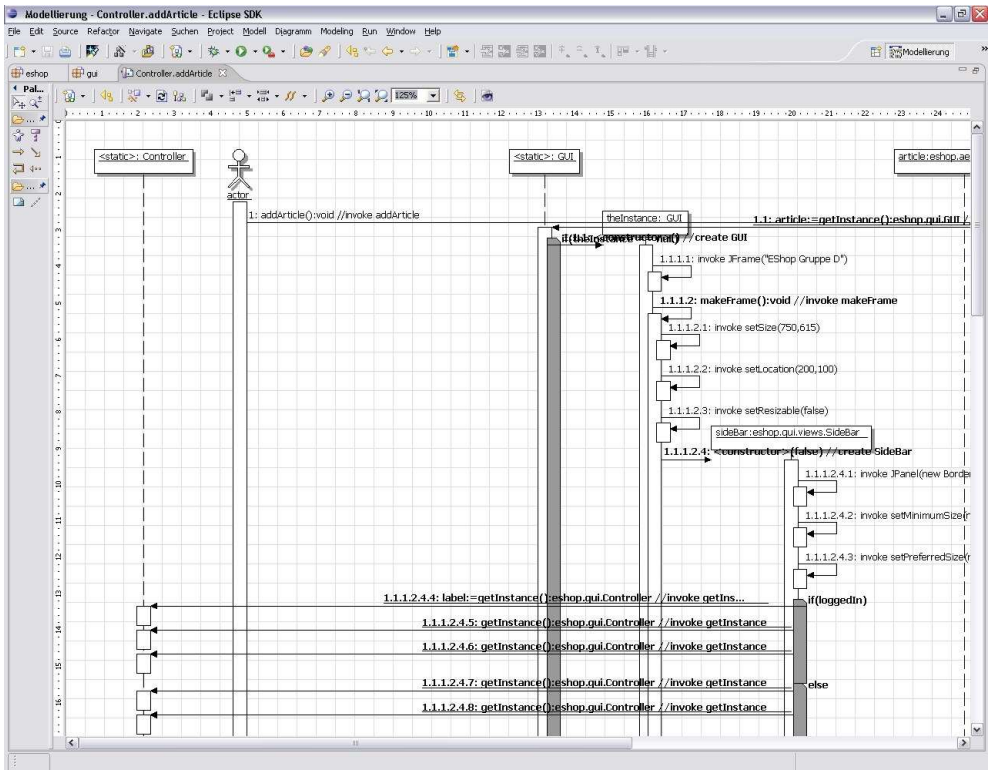


Abbildung 2.4: Screenshot: Sequenzdiagramm in `Together`

gramm ausschließen.

Ein weiteres Tool, das hier Erwähnung finden soll ist IBM `Structural Analysis`. Dieses Tool liefert eine sehr dynamische Graphsicht auf die einzelnen Klassen, Methoden und Pakete des Systems. Ein sehr komfortables Browsen und Ein- wie Ausblenden von Teilpaketen ist möglich (Abb. 2.5). Die Vertiefungsebene der Analyse kann gewählt werden, und Beziehungen wie Benutzen, Erben oder Aufrufe können einzeln angezeigt werden. Außerdem kann der Aufbau des Systems kann auf Schwachstellen und zentrale Verknüpfungspunkte untersucht werden. Patterns wie Butterfly oder Hub machen es möglich die Programmstruktur auf Kohäsion und Kopplung, und somit Stabilität, zu prüfen. Der Butterfly ist eine Struktur in der viele Objekte von einem abhängen, so wie zum Beispiel bei zentralen Interfaces. Das Hub hängt zusätzlich selbst noch von vielen anderen Objekten ab, eine instabile, unerwünschte Struktur. Durch Heuristiken wie das Skeleton lässt sich die insgesamte Struktur des Systems prüfen. Das Skeleton stellt die Programmarchitektur als physikalische, bauklotzartige Grafik dar. Gut designte Systeme sollten dann nicht in sich zusammenfallen.

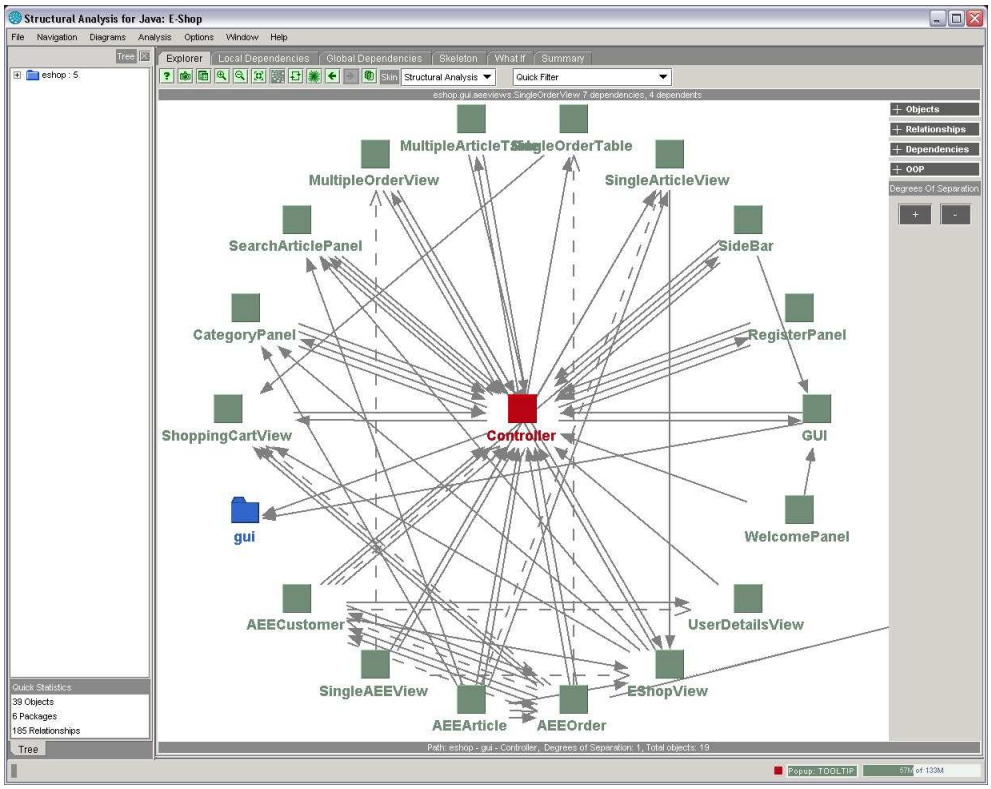


Abbildung 2.5: Screenshot: Klassenstruktur Übersicht in `Structural Analysis`

2.5 Fazit

Reverse Engineering ist ein sehr interessanter und nützlicher Teil des Reengineering Prozesses. Das Zurückgewinnen der Modelle, Konzepte und Anforderungen aus bereits erstelltem Code ist dabei sowohl bei Legacy-Systemen, wie auch während dem normalen Entwicklungsprozess von Nutzen. Unterschieden wird im Reverse Engineering zwischen statischer Modellierung, die sich nur am Quellcode orientiert, und dynamischer Modellierung, die Informationen zur Laufzeit erfasst und auswertet. Dies ist insbesondere bei objektorientierter Software von Vorteil. Häufig entstehen beim Reverse Engineering sehr große Datenmengen, die erst noch gefiltert werden müssen, um übersichtlich visualisiert werden zu können.

Heutige RE-Tools sind in der Lage Code zu analysieren und insbesondere statische UML-Diagramme daraus zu erstellen. Dadurch wird der Reengineering Prozess teilweise automatisiert und ein Round-Trip durch Einbettung in die üblichen Entwicklungsumgebungen ermöglicht. Eine vollständige Automatisierung der Modellgeneration ist sicher utopisch, da die Datenmenge zu groß und komplex strukturiert sind, aber das Erstellen durch wenige Mausklicks ist bereits eine erhebliche Erleichterung. Dennoch gibt es einige Ansätze, vornehmlich dynamische, die in der Praxis bisher kaum Anwendung gefunden haben.

Literaturverzeichnis

[EJC90] James H. Cross II Elliot J. Chikofsky. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–18, 1990.

[ERSS02] Mohammad El-Ramly, Eleni Stroulia, and Paul Sorenson. Recovering software requirements from system-user interaction traces. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 447–454, New York, NY, USA, 2002. ACM Press.

[Lan99] Michele Lanza. Combining metrics and graphs for object oriented reverse engineering. Master's thesis, 1999.

[Lan03] Michele Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, May 2003.

[RD99] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In Hongji Yang and Lee White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 13–22. IEEE, 1999.

[Sch04] Rainer Schmidberger. Use-case-bezogenes reverse-engineering von entscheidungstabellen. In *EMISA*, pages 72–83, 2004.

[Soc] IEEE Computer Society. Swebok. In *http://www.swebok.org/*.

[Sys99] T. Systä. on the relationships between static and dynamic models in reverse engineering java software, 1999.

[Sys00] Tarja Systä. Static and dynamic reverse engineering techniques for java software systems, 2000.

Kapitel 3

Change Impact Analysis

Michael Dreher

Inhaltsverzeichnis

3.1 Einführung

Nehmen Sie an, Sie sind ein Programmierer, und haben gerade eine Änderung am Quelltext eines Programms gemacht. *Was müssen Sie sonst noch ändern, damit das Programm wieder funktioniert?*

Diese Seminararbeit beschäftigt sich mit der Analyse des Einflusses, den eine Änderung eines Teils eines Programms auf den Rest des Programms hat. Wenn eine Entität in einem Programm geändert wird, kann diese Änderung einen Einfluss auf die anderen Entitäten des Programms haben. Es gibt unterschiedliche Herangehensweisen um den Einfluss einer Änderung zu untersuchen, und jede hat ihre Vor- und Nachteile.

Die Analyse des Einflusses einer Änderung in einem Software-System ist nicht nur von theoretischem Interesse, sondern hat ein klares Ziel: *Die Minimierung der Kosten der Software-Wartung*.

Es gibt Studien die behaupten, dass mindestens 50 Prozent der Zeit und des Budgets von großen Software-Systemen für die Wartung verbraucht werden [Zve83]. Durch die Analyse des Einflusses, den eine Änderung einer Entität auf das gesamte Programm hat, kann das Risiko von Seiteneffekten der Änderung auf andere Teile des Systems verringert werden. Damit könnten auch die Kosten der Wartung deutlich gesenkt werden.

Da die Analyse selbst natürlich Zeit und Geld kostet, muss sie Zeit und Geld bei der Wartung einsparen, um einen praktischen Nutzen zu haben. Durch die Wartung entstandene, unbemerkte Fehler in Programmen können sehr viel Geld kosten.

Ein besonders prominentes Beispiel hierfür ist der Absturz der Ariane 5, bei dem ein Schaden von ca. 1,7 Milliarden DM entstanden ist [Gie02]. Durch eine Analyse des Einflusses von Änderungen hätte dieser Absturz möglicherweise verhindert werden können. Durch solche Erfahrungen werden die Anstrengungen getrieben, die man unternimmt, um den Einfluss von Änderungen möglichst automatisiert zu analysieren und zu erkennen.

Fast jedes Programm wird mehr oder weniger häufig verändert. Änderungen werden beispielsweise gemacht, um neue Eigenschaften zum Programm hinzuzufügen, um Fehler zu beheben, oder um die Leistung des Systems zu verbessern. Die Entwickler, die das Programm modifizieren, müssen sicherstellen, dass abhängige Entitäten aktualisiert werden, damit das Programm weiterhin seiner Spezifikation genügt.

3.2 Change Impact Analyse

Wie identifiziert man die abhängigen Entitäten? Manche Abhängigkeiten sind leicht zu identifizieren. Zum Beispiel zieht die Änderung der Signatur einer privaten Methode einer Klasse die Änderung aller klasseninternen Aufrufe dieser Methode nach sich.

Sollte der Entwickler dabei einen Aufruf vergessen, so zeigt spätestens der Compiler diesen Fehler an.

Viele Abhängigkeiten sind aber nicht so leicht, und allein durch den Quelltext des Programms möglicherweise gar nicht zu finden. Gerade bei objektorientierten Programmen gibt es nicht-lokale Effekte, die schwierig zu finden sein können. Und je größer das Programm wird, und je mehr Entwickler daran arbeiten, desto schwieriger wird es alle Abhängigkeiten im System zu identifizieren.

In objektorientierten Sprachen versucht man durch Konzepte wie *information hiding* oder *design by contract* eine möglichst lose Kopplung zwischen den Entitäten eines Programms zu erreichen. Doch auch mit solchen Konzepten lassen sich Abhängigkeiten nicht ganz vermeiden, und bei Änderungen am Programm besteht immer die Gefahr, dass Abhängigkeiten übersehen werden, und dies zu Fehlern führt.

Bei der Analyse des Einflusses von Änderungen (Change Impact) versucht man nun also, den Entwickler mit automatischen Werkzeugen zu unterstützen, die möglichst genau die abhängigen Entitäten einer Änderung identifizieren sollen. Die verschiedenen Ansätze zur Identifizierung der abhängigen Entitäten werden in den folgenden Abschnitten vorgestellt.

Das Ziel der Analyse ist also, dem Entwickler Abhängigkeiten im Programm aufzuzeigen, die er dann auf notwendige Modifikationen prüfen kann. Damit soll bei einer Änderung am Programm das Übersehen von Abhängigkeiten möglichst vermieden werden, um die Kosten und den Zeitaufwand von Änderungen zu minimieren.

3.2.1 Propagierung von Änderungen

Wie wirkt sich die Änderung einer Entität eines Programms auf die anderen Entitäten aus? Die *Propagierung* einer Änderung bezeichnet alle Änderungen die nötig sind, um die Konsistenz des Programms mit seiner Spezifikation sicherzustellen, nachdem eine bestimmte Entität geändert wurde.

Beispielsweise können nach der Änderung einer Methode einer Klasse, Änderungen in anderen Klassen nötig sein, die die geänderte Methode aufrufen. Die Propagierung ist beendet, wenn alle Änderungen abgeschlossen sind, sodass das Programm wieder seiner Spezifikation genügt.

Die Änderung einer Entität hat üblicherweise Nachwirkungen, die sich wie eine Welle durch das ganze Programm erstrecken können. Idealerweise sollte, insbesondere in objektorientierten Systemen, eine Änderung einer Entität von der geänderten Entität vollständig absorbiert werden, und keine Auswirkungen auf andere Entitäten haben. Dieses Ideal wird in der Praxis jedoch nicht erreicht.

Der typische Ablauf einer Propagierung sieht so aus: Durch eine neue Anforderung, Verbesserung oder Fehlerbehebung identifiziert der Entwickler die erste Entität, die geändert werden muss. Nach der Änderung an dieser Entität bestimmt er weitere En-

titäten, die geändert werden müssen, und ändert diese ebenfalls. Findet er selbst keine weitere abhängige Entität mehr, bittet er um Hilfe.

Die Hilfe kann von einem erfahrenen Entwickler, einem Hilfswerkzeug oder einer Reihe von Tests kommen. Möglicherweise werden noch weitere Entitäten gefunden, die zu ändern sind. Der Entwickler ändert sie, und sucht wieder nach abhängigen Entitäten. Dieser Prozess wird so lange durchlaufen, bis keine abhängigen Entitäten mehr gefunden werden. Im Idealfall ist das Programm jetzt wieder konsistent mit seiner Spezifikation.

Das Ziel der Analyse des Einflusses von Änderungen ist es, einen möglichst gutes Werkzeug zu entwickeln, um den Entwickler zu unterstützen. Je automatischer und schneller abhängige Entitäten von diesem Werkzeug gefunden werden können, desto mehr Zeit und Kosten spart man bei der Wartung.

Nicht korrekt oder nicht vollständig propagierte Änderungen führen oft schwerwiegende und schwer auffindbare Fehler im Programm ein. Die Kosten und die Zeit, diese Fehler dann zu finden und zu beheben, könnte man einsparen, wenn man diese Fehler von vorne herein vermeiden könnte.

Es reicht jedoch nicht aus, nur alle abhängigen Entitäten zu identifizieren. Die abhängigen Entitäten müssen dann auch korrekt modifiziert werden, damit das Programm wieder seiner Spezifikation genügt. Werkzeuge, die den Einfluss von Änderungen untersuchen, identifizieren zwar abhängige Entitäten, können aber die korrekte Anpassung der identifizierten Entitäten nicht selbst übernehmen.

3.3 Programmanalyse

Der intuitivste Ansatz ist die direkte Analyse des Quelltextes des Programms. Bei diesem Ansatz betrachtet man statische Abhängigkeiten zwischen den Entitäten eines Programms, die aus dem Quelltext oder einer Laufzeitanalyse des Programms hervorgehen.

3.3.1 Call-Graphen

Eine einfache Methode die Abhängigkeiten in einem Programm darzustellen ist der *call-Graph*. Die Abhängigkeiten werden als Graph notiert, mit den Entitäten als Knoten, und den Abhängigkeiten als Kanten. Dieser Graph setzt Entitäten in Beziehung, wenn eine Entität Leistungen bereitstellt, die von einer anderen Entität für die korrekte Funktion benötigt werden.

Dies kann der Aufruf einer Funktion von einer anderen Funktion aus sein, oder eine Methode einer Klasse, die eine Methode einer anderen Klasse aufruft. Basierend auf diesen Daten kann von einer geänderten Entität im Programm auf andere Entitäten

geschlossen werden, die möglicherweise von der Änderung beeinflusst werden.

Ein call-Graph lässt sich relativ leicht erzeugen, indem zum Beispiel jeder Aufruf einer Funktion aus einer Funktion eine Kante zwischen den Funktionen erzeugt. Weitere Relationen können gebildet werden, indem das Benutzen einer Variable ebenfalls eine Kante zwischen Funktion und Variable erzeugt, ebenso für die Definition einer Variable. Dieser Graph repräsentiert dann die Struktur des Programms, bezogen auf die Abhängigkeiten zwischen den Entitäten.

Ist es bei imperativen Programmen noch leicht möglich, einen call-Graph zu erstellen, so wird es bei objektorientierten Programmen kompliziert. In objektorientierten Sprachen gibt es Konzepte wie *dynamisches Binden* und *Polymorphismus*.

Diese Konzepte machen die Identifizierung von Abhängigkeiten, allein mit einer statischen Analyse des Quelltextes, unmöglich. Ein klares Beispiel für dieses Problem ist das dynamische Binden, bei dem erst zur Laufzeit entschieden werden kann, zu welcher Klasse eine aufgerufene Methode gehört. Diese Abhängigkeit ist allein auf Grund des Quelltextes nicht identifizierbar.

Die Analyse eines call-Graphen ist mit einem *transitiven Abschluss* leicht auszuführen, bringt aber schlechte Ergebnisse. Einerseits werden alle irgendwie abhängigen Entitäten als beeinflusst markiert, andererseits wird nur der Aufruf einer Entität betrachtet, und alle anderen Aspekte des Aufrufs ignoriert. Als Folge sind die Ergebnisse des transitiven Abschlusses auf einem call-Graphen zwar zahlreich, aber sehr ungenau.

3.3.2 Program Slicing

Beim *Slicing* (schneiden) eines Programms betrachtet man generell einen Ausschnitt eines Programms, und die Zustände die dieser Teil des Programms durchläuft. Für eine Variable oder eine Menge von Variablen v, und eine Anweisung n beinhaltet der *Slice* alle Anweisungen, die zum Wert von v beigetragen haben, bis zu dem Punkt, an dem n aufgerufen wird. Ein Slice eines Programms ist also eine Teilmenge der Anweisungen des Programms.

Beim *statischen* Slicing werden die Teile des Programms betrachtet, die relevant für die Berechnung *sein könnten*. Dies wird gemacht, indem die statisch vorhandene Information, also der Quelltext, dazu benutzt wird, einen call-Graphen zu erstellen. Von der Anweisung, die betrachtet wird, werden alle Anweisungen zurückverfolgt, die zur Berechnung der untersuchten Anweisung oder Variablen beigetragen haben könnten.

Statisches Slicing ist *sicher* in dem Sinne, dass es alle möglichen Eingabewerte und Auswirkungen im Programm betrachtet [LR03]. Bei Programmen, die sicherheitskritisch sind, ist dieser konservative Ansatz wichtig. Dennoch wird möglicherweise eine große Menge von abhängigen Entitäten bestimmt, die nur noch wenig Nutzen für einen Entwickler hat. Die Berechnung von statischen Slices ist sehr speicheraufwendig und für angemessen lange Programmdurchläufe nicht mehr in akzeptabler Zeit durchführ-

bar.

Das *dynamische* Slicing extrahiert dagegen genau die Anweisungen des Programms, die für eine gegebene Menge von Eingabewerten relevant *sind*. Dies wird gemacht, indem das Programm einmal ausgeführt wird. Dann können die Abhängigkeiten zwischen den Entitäten des Programms mit Hilfe der Ablaufverfolgung identifiziert werden.

Diese Methode ist in akzeptabler Zeit berechenbar, aber auf Kosten der Sicherheit. Es werden nicht mehr alle möglichen Auswirkungen betrachtet, die auftreten können, sondern nur noch die, die für die gegebenen Eingabewerte aufgetreten sind.

Ein dynamischer Slice ist also vollkommen abhängig von den Eingabewerten, und damit auch nur so gut wie die gewählten Eingabewerte. Zhang und Gupta haben eine effiziente Methode für dynamisches Slicing entwickelt [ZG04]. Mit dieser Methode ist das dynamische Slicing auch praktisch einsetzbar.

Mit statischem oder dynamischen Slicing kann man also entweder *sichere* Resultate bekommen, oder *schnelle* Resultate, nicht beides. Zwischen diesen beiden Extremen muss man in jedem Fall einen Kompromiss eingehen.

3.3.3 Chianti

Eine Analyse basierend auf Slicing und Tests ist in einem Prototyp mit dem Namen Chianti umgesetzt [RT01, RST+04]. Zur Analyse der Abhängigkeiten müssen dabei Änderungen am Quelltext in eine Menge von atomaren Änderungen transformiert werden. Diese atomaren Änderungen sind beispielsweise das Ändern eines Methodenrumpfes, das Löschen einer Klasse oder das Hinzufügen eines Feldes. Eine zwingende Voraussetzung ist das Vorhandensein von Tests.

Chianti generiert für jeden vorhandenen Testfall dynamisch einen Kontrollflußgraphen [RST+04]. Man betrachtet zwei Versionen des Programms. Eine Version vor der Änderung, und eine Version nach der Änderung. Für Tests, die Teile der Programme ausführen, wird dann eine Menge von Tests bestimmt, deren Verhalten sich geändert haben kann. Für jeden dieser betroffenen Tests wird dann eine Menge von Änderungen bestimmt, die dazu geführt haben, dass sich das Verhalten des Tests geändert hat.

Es werden also nicht die Entitäten gesucht, die von der Änderung einer Entität beeinflusst worden sein könnten, sondern die Änderungen die einen Test beeinflusst haben könnten.

Die Umsetzung von Chianti ist für die Programmiersprache Java vollständig, liefert also bei Fehlschlagen eines Tests alle verantwortlichen Änderungen zurück [RST+04]. Klar ist auch, dass dieser Ansatz nur so gut sein kann, wie die Tests, die ihm zugrunde liegen. Wenn eine Änderung keinen Test beeinflusst, weil kein Test diesen Teil des Quelltextes abdeckt, dann könnte man in trügerische Sicherheit verfallen, dass die gerade gemachte Änderung keinen Einfluss auf das Programm hat, auch wenn sie es in

Wirklichkeit hat. Das ist ein generelles Problem beim Verwenden von Tests, und nicht spezifisch für diesen Ansatz.

Dieser Ansatz ist vor allem deshalb interessant, weil er durchaus praktisch einsetzbar ist. Die Berechnungszeiten von Chianti sind gut genug, um es in die Entwicklung mit einzubeziehen, und die Ergebnisse können zur Vermeidung von Fehlern beitragen.

Andererseits ist dieser Ansatz nicht sicher in dem Sinne, dass nicht alle möglichen Verhaltensweisen des Programms untersucht werden, sondern nur die, die von Tests abgedeckt sind. Eine gute Abdeckung mit Tests würde aber zu einer hilfreichen, und gleichzeitig praktisch gut einsetzbaren Methode führen.

3.3.4 Empirische Ergebnisse

In der Studie von Hassan und Holt schneidet die Analyse des Quelltextes am schlechtesten ab, was zum Teil auch auf die Einfachheit der Identifizierung der Abhängigkeiten zurückzuführen sein könnte [HH04]. Eine Abhängigkeit wird hier nur erstellt, wenn eine Methode eine andere Methode aufruft, eine Variable deklariert, oder eine Variable benutzt. Aufgrund der schlechten Resultate verfolgen die Autoren diesen Ansatz nicht weiter.

Die Analyse der, in diesem Paper CUD genannten, Heuristik liefert nur 42% der nötigen Änderungen zurück, und das ohne die Ergebnisse zu filtern. Andere Methoden erzielen hier 83% oder 87%. Auch die Präzision der zurückgelieferten Abhängigkeiten ist schlecht. Liefern andere Ansätze ohne Filterung der Ergebnisse immerhin noch 6% bis 12% korrekte Abhängigkeiten zurück, so stimmen von den vorgeschlagenen Änderungen der CUD Heuristik nur 2%.

Eine detaillierte Beschreibung eines Algorithmus für dynamisches Slicing beschreiben Zhang und Gupta [ZG04]. Diese Version ist im Vergleich zu älteren Slicing-Algorithmen wesentlich leistungsfähiger, wird OPT genannt, und mit den Algorithmen FP, NP und LP verglichen.

Der Algorithmus FP konstruiert aus der Ablaufverfolgung des Programms den gesamten Call-Graphen, bevor ein Slice berechnet werden kann. Diese Methode braucht enorm viel Speicherplatz um den Graphen abzulegen, und wenig Zeit zur Berechnung. Der Algorithmus NP berechnet nichts, bevor ein Slice berechnet wird. Diese Methode braucht wenig Speicher, aber enorm viel Zeit für die Berechnung. Der Algorithmus LP ist ein Kompromiß aus beiden Ansätzen, indem vor der Berechnung eines Slices eine Zusammenfassung des Call-Graphen erzeugt wird, um bei der eigentlichen Berechnung die relevanten Stellen im Graph schnell zu finden. Diese Methode braucht nicht viel Speicher, ist aber trotzdem schnell.

Der Algorithmus LP kommt wie OPT für realistische Programmdurchläufe mit dem Hauptspeicher aus, braucht aber für die Berechnung der Abhängigkeiten zwischen 4.69 und 25.21 Minuten in den durchgeführten Experimenten. OPT hingegen benötigt

für den selben Slice nur 1.74 bis 36.35 Sekunden. Die optimierte Berechnungszeit beträgt also nur etwa 0,5% bis 2,5% der Zeit, die LP benötigt. Der FP-Algorithmus verbraucht zu viel Speicher für realistische Programmanalysen, braucht aber trotzdem noch etwa doppelt so lange wie OPT.

Die Leistungsfähigkeit von Chianti wird empirisch an einer Software namens Daikon untersucht [RST+04]. Ein Schwachpunkt wird hier deutlich sichtbar, nämlich die Abdeckung des Quelltextes durch Tests. Diese ist bei Daikon nur ca. 21%, und damit relativ niedrig. Bessere Abdeckung durch Tests würde auch die Leistungsfähigkeit von Chianti steigern.

Chianti betrachtet nicht die Änderung einer einzigen Entität, sondern eine zusammenhängende Änderung aller Entitäten bis zur Wiederherstellung der Konsistenz des Programms. Dabei ist die Menge von beeinflussten Tests im Durchschnitt 52% aller Tests. Für jeden beeinflussten Test ist die Anzahl der dafür verantwortlichen Änderungen sehr gering. Nur ca. 4% der atomaren Änderungen dieser Modifikation des Programms werden zur Überprüfung vorgeschlagen. Damit zeigt Chianti sehr genau die verantwortlichen Änderungen für beeinflusste Tests auf.

3.3.5 Vor- und Nachteile

Was sind die Vor- und Nachteile der Untersuchung der strukturellen Abhängigkeiten innerhalb eines Programms? Jedes Programm besteht aus Quelltext, also lassen sich auch für jedes Programm diese strukturellen Abhängigkeiten untersuchen.

Strukturelle Abhängigkeiten können auch dann identifiziert werden, wenn sie vorher noch nie aufgetreten sind, was ein klarer Vorteil zu den historischen Abhängigkeiten ist, siehe Abschnitt 3.4. Die Kosten der Bereitstellung der Daten für die strukturelle Analyse sind gleich null, da der Quelltext in jedem Fall vorhanden und untersuchbar ist.

Diese Methode hat aber auch große Nachteile. Je größer das Programm ist, desto aufwendiger ist die Analyse und stößt bei allen statischen Ansätzen schnell an die Grenzen der Berechenbarkeit. Das dynamische Slicing bietet eine berechenbare Alternative zur Untersuchung der Abhängigkeiten eines Programms, aber auf Kosten der Sicherheit der Ergebnisse. Die praktische Einsetzbarkeit des dynamischen Slicings haben Zhang und Gupta gezeigt [ZG04].

Die Analyse der statischen Abhängigkeiten der einzelnen Entitäten des Programms kann auch nur statische Abhängigkeiten entdecken. Vielen Änderungen geht aber eine funktionale Überlegung voraus, die Änderungen in aus statischer Sicht völlig unabhängigen Teilen des Programms bewirken können. Solche Abhängigkeiten zu entdecken ist mit der Programmanalyse nur schwer möglich. Theoretisch ist die Entdeckung einer Abhängigkeit zwischen einer Datenbanktabelle und der zugreifenden Methode zwar möglich, praktisch aber sehr aufwendig.

Ein Nachteil von Chianti ist, dass die Änderungen zuerst implementiert werden müssen, bevor sie untersucht werden können. Eine Analyse der Auswirkungen von Änderungen bevor man die Änderungen macht, also zum Beispiel um Vorteile und Nachteile verschiedener Änderungsvarianten zu prüfen, ist damit ausgeschlossen.

3.3.6 Zusammenfassung

Generell muss man bei diesem Ansatz immer einen Kompromiss zwischen der Anzahl und der Genauigkeit der Ergebnisse, sowie der Sicherheit und der Schnelligkeit der Ergebnisse eingehen. Je nachdem in welchem Kontext man die Analyse anwendet, muss man also abwägen, welche Ziele die Analyse des Einflusses von Änderungen haben soll.

In einer Situation, in der die Sicherheit am wichtigsten ist, muss man Abstriche in der Schnelligkeit und der Genauigkeit hinnehmen und statisches slicing einsetzen. In Situationen, in denen man möglichst schnelle, genaue Ergebnisse haben will, muss man hinnehmen, dass man nur wenige Ergebnisse bekommt.

3.4 Analyse der Versionshistorie

Ein vielversprechender Ansatz ist die Analyse der Versionshistorie eines Softwareprojektes. Hierbei basieren die Abhängigkeiten im Programm nicht auf dem Quelltext, sondern auf Daten aus der Versionshistorie. Entitäten sind dann abhängig voneinander, wenn sich in der Vergangenheit schon zusammen geändert wurden.

Die Frage ist in diesem Fall weniger die Frage *ob*, sondern *wie stark* die Entitäten voneinander abhängig sind.

Bei Amazon[1] gibt es eine Einblendung auf der Internetseite, die eine Empfehlung

> Kunden, die dieses Buch gekauft haben, haben auch diese Bücher gekauft...

ausgibt. Die Liste der Empfehlungen enthält dann Bücher, die typischerweise mit dem ausgewählten Buch gekauft wurden. Den gleichen Ansatz verfolgt die Analyse der Versionshistorie, nach dem Motto

> Programmierer, die diese Funktion geändert haben, haben auch folgende Funktionen geändert...

[1] www.amazon.de

3.4.1 Datenquellen

Zur Analyse der Versionshistorie muss diese natürlich erst einmal existieren. Nahezu jedes größere Softwareprojekt hat ein Versionshistorie, und in allen hier bearbeiteten Untersuchungen war dies ein CVS System. Das große Problem bei der Analyse von Versionshistorien ist, dass die Verbindung zwischen den einzelnen Einträgen beim Speichern verloren geht.

Eine logische Änderung teilt sich also in viele einzelne individuelle Änderungen an Dateien auf, zwischen denen die logische Verbindung verloren geht. Diese logische Verbindung muss erst wieder hergestellt werden, was aber das Thema einer anderen Ausarbeitung ist. Für diese Ausarbeitung kann also vorausgesetzt werden, dass die logischen Verbindungen zum größten Teil wiederhergestellt sind.

3.4.2 Analyse der Daten

Zur Untersuchung der Abhängigkeiten werden *Support* und *Confidence* definiert [ZDZ03, ZWDZ04]. Der *Support* einer Abhängigkeit zwischen zwei Entitäten beschreibt, wie oft diese beiden Entitäten schonmal zusammen geändert wurden. Insbesondere gibt der support einer Entität mit sich selbst die absolute Anzahl der Änderungen an dieser Entität an.

Die *Confidence* beschreibt die Stärke der Abhängigkeit, also die Wahrscheinlichkeit einer Änderung an einer Entität, wenn eine andere Entität geändert wird. Mit diesen beiden Werten kann man beschreiben, wie sicher eine Änderung an einer Entität eine Änderung an einer anderen Entität hervorruft. Somit können die Ergebnisse der Analyse nach Aussagekraft sortiert werden.

3.4.3 ROSE

ROSE ist ein Werkzeug zur Analyse der Versionshistorie, das Abhängigkeiten zwischen Entitäten, wie Funktionen oder Variablen erkennt, also eine detaillierte Analyse der Abhängigkeiten durchführt [ZDZ03, ZWDZ04].

Ausserdem benutzt es Techniken des *Data Mining* für die Bestimmung der Entitäten aus den Abhängigkeiten in der Versionshistorie. Auf diese Techniken wird hier nicht näher eingegangen.

Wenn man als Entwickler an einer Entität wie beispielsweise einem Array eine Änderung macht, gibt ROSE eine Liste von abhängigen Entitäten aus, die bei der Änderung dieses Arrays in der Vergangenheit ebenfalls geändert wurden. Die Vorschläge, die ROSE ausgibt, sind nach *Confidence* absteigend sortiert, so dass als oberster Vorschlag die sicherste Abhängigkeit steht. Die Liste enthält nur Einträge, bei denen support und confidence entsprechend hoch sind.

Ein beeindruckendes Ergebnis ist die Geschwindigkeit, mit der ROSE die Ergebnisse produziert. Bei der umfangreichen Versionshistorie von GCC kommt ROSE auf einem aktuellen PC durchschnittlich in einer halben Sekunde zum Ergebnis. ROSE ist damit definitiv praktisch einsetzbar, was auch dadurch verstärkt wird, dass es als Eclipse-Plugin verfügbar ist.

Mit einer empirischen Untersuchung der Versionshistorien von acht verschiedenen, großen Softwareprojekten wie Eclipse, GCC oder KOFFICE, wurde die Leistungsfähigkeit von ROSE getestet. Die Wahrscheinlichkeit, dass ROSE bei den ersten drei Vorschlägen eine korrekte Abhängigkeit erkennt, liegt bei durchschnittlich 64%, bei GCC sogar bei 89%.

Bei der Untersuchung ob ROSE vergessene Abhängigkeiten erkennen kann, sind die Ergebnisse nicht so gut. Im Schnitt erkennt ROSE eine ausgelassenen Entität nur in 4% der Fälle. Wenn es eine solche ausgelassene Entität erkennt, ist dieser Vorschlag aber zu 50% korrekt. Auch hier schneidet GCC mit 20% und 81% wieder überdurchschnittlich gut ab.

Bei der Untersuchung der Fehlalarme, also der Fälle, in denen ROSE eine vergessene Entität fälschlicherweise anmahnt, ist das Ergebnis mit 2% sehr niedrig und damit sehr gut. Hier kann ROSE also sehr hiflreich sein, und eine Warnung über eine vergessene Entität sollte ernst genommen werden.

Insgesamt lassen die Ergebnisse von ROSE bei den verschiedenen Projekten auch vermuten, dass die Analyse der Versionshistorie bessere Ergebnisse liefert, je ausgereifter und stabiler ein Projekt ist. Hierbei stechen vor allem die Ergebnisse mit GCC heraus, die mit dem guten Abschneiden von GCC in anderen Untersuchungen ähnlich sind [HH04, ZDZ03]. Zimmermann u.a. kommen ja auch zu dem Ergebnis, dass die Entwicklungsgeschichte das Design von GCC rechtfertigt [ZDZ03].

3.4.4 Vor- und Nachteile

Um Ergebnisse zu erzielen braucht die Analyse der Versionshistorie allein nicht einmal auf den Quelltext des Programms zuzugreifen. Diese Analysemethode kann sogar Abhängigkeiten zwischen Entitäten entdecken, die nicht mal Teil des Programms sind, so wie zum Beispiel zwischen einer Klasse und ihrer Dokumentationsdatei, oder einem Datenbankschema und der zugreifenden Methode. Diese Abhängigkeiten sind mit einer Programmanalyse nicht zu identifizieren.

Die Analyse der Versionshistorie kann auch dazu verwendet werden, um die Architektur des untersuchten Programms zu prüfen [ZDZ03]. Unstimmigkeiten zwischen historischen Abhängigkeiten und geplanten Abhängigkeiten können auf Schwachstellen im Design des Systems hinweisen, und damit Stellen identifizieren an denen eine Restrukturierung der Entitäten möglich wäre.

Die empirischen Ergebnisse der Analyse der Versionshistorie sind vielversprechend.

Bei Hassan und Holt liefert diese Methode das beste Ergebnis bei der Identifizierung der Abhängigkeiten, und das zweitbeste Ergebnis bei der Präzision der Vorschläge [HH04]. Ebenfalls zu sehr guten Ergebnissen kommen [ZDZ03] und [ZWDZ04]. Das entwickelte Werkzeug ROSE gibt durchschnittlich in 64% der untersuchten Fälle mit den ersten drei Vorschlägen eine korrekten Vorschlag an. Noch deutlicher ist das Ergebnis, dass ROSE nur in 2% der Fälle, in denen es eine vergessene Entität anmahnt, unrecht hat. Weitere Vorteile liegen in der schnellen Berechenbarkeit der Abhängigkeiten, im Gegensatz zur Analyse des Quelltextes.

Ein Nachteil ist das nötige Vorhandensein einer Versionshistorie, was allerdings bei fast jedem größeren Softwareprojekt der Fall ist. Ausserdem macht die Anwendung erst Sinn, wenn das Softwareprojekt schon eine Historie hinter sich hat, auf dessen Basis Abhängigkeiten gefunden werden können. Ein gerade neu angefangenes Projekt liefert keine sinnvollen Daten bei der Analyse der Versionshistorie.

Je mehr es aus der Historie zu lernen gibt, desto besser sind die Ergebnisse der Analyse, was man auch erwarten würde. Umgekehrt erkennt dieser Ansatz natürlich keine Abhängigkeiten, die noch nie aufgetreten sind.

Ein weiterer Nachteil ist, dass die logische Verbindung zwischen den verschiedenen Einträgen in der Versionshistorie verloren geht. Sie kann zwar wiederhergestellt werden, diese Wiederherstellung kann aber durchaus fehlerhafte Resultate hervorbringen. Für exakte Daten wäre ein Versionssystem nötig, das auch die logischen Verbindungen speichert.

Ausserdem ist die Wiederherstellung der logischen Verbindungen der Versionshistorie und die Vorbereitung der Daten für die Analyse mit Kosten verbunden, steht also nicht automatisch bereit, wie der Quelltext des Programms.

3.4.5 Zusammenfassung

Insgesamt stellt sich die Analyse der Versionshistorie als sehr gutes Mittel heraus, um die Auswirkungen von Änderungen schnell zu bestimmen. Vor allem die empirischen Ergebnisse und der praktische Nutzen des Werkzeuges ROSE sind gut. Die klare Stärke liegt in der Unabhängigkeit von der Programmiersprache, des Quelltextes und der Möglichkeit auch Abhängigkeiten zu entdecken, die mit einer Programmanalyse nicht identifizierbar sind.

Aber auch bei der Analyse der Versionshistorie muss man einen Kompromiss zwischen der Anzahl und der Genauigkeit der Ergebnisse eingehen. Der Kontext ist also auch hier entscheidend für den Einsatz. Die empirischen Ergebnisse von ROSE unterstreichen den Wert, den die Analyse der Versionshistorie haben kann.

3.5 Analyse von Use Case Maps

Ein weiterer Ansatz ist die Analyse von Dokumenten, die das Design eines Programms beschreiben. Eine Möglichkeit ist die Analyse des Einflusses von Änderungen auf der Basis von *Use Case Maps* [JH05]. Dabei wird nicht die Änderung an einer Entität des Programms untersucht, sondern die Änderung an einer Anforderung der Spezifikation des Programms.

Use Case Maps beschreiben die funktionalen Bestandteile eines Programms aus der Anwendersicht. Es gibt Akteure wie Benutzer oder Datebanken, die mit dem System interagieren, und es gibt *Use Cases*, die eine bestimmte Funktionalität des Systems beschreiben. Eine Use Case Map beschreibt die Zustände, und die Übergänge zwischen den Zuständen eines Systems mit einer Struktur, die es erlaubt sie automatisch zu untersuchen. In diesem Ansatz wird zur Untersuchung der Anforderungen eine Art Slicing verwendet.

Auch dieser Ansatz hat Vorteile und Nachteile. Die Voraussetzung der Analyse von Use Case Maps ist natürlich ihr Vorhandensein. Zur automatischen Anwendung der Analysealgorithmen ist auch die genaue Einhaltung der Syntax dieser Use Case Maps wichtig.

Von Nachteil ist auch, dass die Spezifikation in Form der Use Case Maps mit dem Programm konsistent sein, und während der Entwicklung konsistent gehalten werden muss. Nur so erhält man auch in späteren Phasen noch sinnvolle Ergebnisse aus der Analyse. Diese Konsistenz ist in der Praxis wahrscheinlich gar nicht zu finden.

Die Ergebnisse der Analyse sind auch nicht genau auf bestimmte Teile des Quelltextes bezogen, wie in anderen Ansätzen. Die Use Case Maps geben die funktionale Sicht des Programms wieder, und sind deshalb ungenauer als bespielsweise ein statisches Slicing. Es ist auch fraglich ob diese ungenauen Informationen während der Entwicklung des Programms noch nützlich sind.

Ein Vorteil ist, dass wie bei der Analyse der Versionshistorie auch Abhängigkeiten zwischen Entitäten gefunden werden können, die für die Programmanalyse nicht auffindbar sind. Da den Use Case Maps eine funktionale Sicht auf das System zugrunde liegt, sind funktional bedingte Abhängigkeiten mit diesem Ansatz leicht zu finden.

Ein Alleinstellungsmerkmal gegenüber allen anderen Ansätzen ist die Einsetzbarkeit schon bevor auch nur eine einzige Zeile Quelltext geschrieben wurde, also in sehr frühen Stadien eines Projektes. Gerade zur Evaluierung von antizipierten Änderungen während der Evolution des Programms können so verschienene Designvarianten verglichen werden. Dieser Ansatz dürfte seine Stärken also vor allem in der Planungsphase eines Softwareprojektes haben.

Empirische Ergebnisse gibt es zu diesem Ansatz leider nicht, insgesamt ist dieser Ansatz auch bei Weitem noch nicht so detailliert erforscht wie die Ansätze zur Programmanalyse oder das Untersuchen der Versionshistorie. Um klarere Ergebnisse ableiten

zu können, müssten noch weitere Untersuchungen in diesem Gebiet gemacht werden.

3.6 Hybride Analyse

Bei der hybriden Analyse werden verschiedene Ansätze miteinander kombiniert.

Hassan und Holt bestimmen als die beiden besten Heuristiken die HIS-Heuristik, die auf historischen Abhängigkeiten basiert, und die FIL-Heuristik, die auf den Abhängigkeiten des Layouts im Quelltext basiert [HH04].

Bei der FIL-Heuristik werden Entitäten als abhängig voneinander betrachtet, wenn sie in der gleichen Datei sind. Dieser Ansatz alleine ist für seine Einfachheit erstaunlich effektiv. Zu beachten ist allerdings, dass alle untersuchten Systeme dieser Studie in C programmiert sind, und die Ergebnisse der FIL-Heuristik in objektorientierten Programmiersprachen eher nutzlos sein dürften. Es würde nämlich immer die aktuell bearbeitete Klasse zur Überprüfung vorgeschlagen.

Allerdings muss auch klar sein, dass diese Feststellung ihre Tücken hat. Würde man diese Heuristik auf ein Programm anwenden, das nur aus einer einzigen Datei besteht, so würde FIL immer alle Entitäten erkennen, die auch wirklich von der Änderung betroffen sind, nämlich die einzige Datei. Diese Information wäre aber wertlos, denn man müsste die gesamte Datei, also das gesamte Programm, untersuchen.

Auf groberer Ebene, wenn also nicht Entitäten, sondern zum Beispiel nur Dateien als Ergebnis zurückgegeben würden, wäre FIL sogar eine Heuristik, die viele *und* genaue Ergebnisse liefert. Es würden nicht nur immer alle betroffenen Dateien, also die einzige Datei, korrekt erkannt, die Präzision des Ergebnisses läge auch noch bei 100%, da die Datei immer die richtige ist. Der Wert dieses Ergebnisses für den Programmierer wäre aber gleich null.

Die Autoren kombinieren HIS und FIL mit unterschiedlichen Parametern der Gewichtung, und kommen zu dem Ergebnis, dass auch die Kombination dieser Techniken entweder viele Ergebnisse oder genaue Ergebnisse liefern kann, und nicht beides. Dies liegt daran, dass man nur in eine Richtung optimieren kann. Man kann entweder möglichst viele wirklich betroffene Entitäten finden, und findet dabei auch viele Entitäten die nicht betroffen sind, oder man beschränkt sich auf die wirklich sicher betroffenen Entitäten, und übergeht dabei andere, ebenfalls betroffene Entitäten.

Auch ein hybrider Ansatz kann also den Kompromiss zwischen Qualität und Quantität der Ergebnisse nicht auslöschen. Die praktische Einsetzbarkeit von hybriden Techniken ist nicht speziell eingeschränkt, sondern hängt von der praktischen Einsetzbarkeit der kombinierten Techniken ab.

Die Kombination verschiedener Techniken, wie zum Beispiel der Analyse der Programmstruktur und der Versionshistorie, kann aber durchaus nützlich sein. In frühen Stadien der Entwicklung eines Programms kann die Versionshistorie noch keine sinn-

vollen Ergebnisse liefern. Zu diesem Zeitpunkt würde die Programmanalyse die besten Ergebnisse erzielen, und würde damit den größten Nutzen bei der Wartung des Programms generieren.

Eine Stärke der Analyse der Versionshistorie ist die Aufdeckung von Abhängigkeiten, die die Programmanalyse nicht erkennen kann. Die Analyse der Versionshistorie könnte also zusätzliche Abhängigkeiten zum Ergebnis der Programmanalyse beitragen. Dies wäre genau dann möglich, wenn beide Analysetechniken getrennt ihre Ergebnisse sammeln, und diese dann kombinieren.

In späteren Stadien liefert die Versionshistorie möglicherweise generell bessere Ergebnisse als die Programmanalyse. Aber auch dann wären immer noch Abhängigkeiten von der Programmanalyse auffindbar, die die Versionshistorie nicht entdecken kann. Nämlich dann, wenn diese Abhängigkeiten in der Entwicklungsgeschichte des Projektes noch nicht aufgetreten sind, sich aber aus der Struktur des Programms ergeben.

Es wäre also durchaus lohnend verschiedene Techniken miteinander zu kombinieren. Vor allem eine Kombination der beiden Werkzeuge Chianti und ROSE könnte interessante Ergebnisse liefern, da beide Techniken Schwächen haben, die die andere Technik ausgleichen kann. Klar ist aber auch, dass eine Kombination zwar dazu führen kann, dass Schwächen einer Analyse durch die Andere aufgehoben werden, wahrscheinlich aber nicht dazu führen kann, dass man viele Ergebnisse *und* genaue Ergebnisse erzielt. Dieser Kompromiss gilt generell auch für hybride Ansätze.

3.7 Fazit

Die Analyse des Einflusses von Änderungen in einem Programm kann dazu beitragen, die Kosten der Wartung des Programms zu verringern. Es gibt generell zwei Kompromisse, die man bei der Analyse eingehen muss.

Zum Einen kann man entweder eine sichere, aber aufwendige Analyse machen, oder eine Schnelle, aber Unsichere. Zum Anderen kann man entweder viele mögliche Abhängigkeiten als Ergebnis der Analyse bekommen, oder genaue Abhängigkeiten. Kein Ansatz kann zeigen, dass er sicher *und* schnell ist, oder dass er alle Abhängigkeiten identifiziert die es gibt, *und* die gefundenen Abhängigkeiten genau die Richtigen sind.

Die Programmanalyse hat ihre Stärken in der Quantität der Ergebnisse, und der Möglichkeit, zumindest theoretisch eine sichere Analyse machen zu können, in dem Sinne, dass keine statisch abhängige Entität übersehen wurde. Diese Stärken sind nützlich in Systemen, die sicherheitskritisch sind, wie zum Beispiel in medizinischen Systeme, bei denen eine Fehlfunktion möglicherweise tödliche Folgen haben könnte.

Die Schwächen der Programmanalyse liegen in der aufwendigen Berechnung, und der auf statische Verbindungen eingeschränkten Sicht auf das Programm. Abhängigkeiten die nicht aus der Statik des Programms hervorgehen, kann keine Programmanalyse identifizieren. Dies ist auch bei der Anwendung in sicherheitskritischen Systemen zu

beachten.

Die Analyse der Versionshistorie hat ihre Stärken in der Qualität der Ergebnisse, der praktischen Einsetzbarkeit in gewachsenen Softwareprojekten, und der Schnelligkeit der Analyse. Eine weitere große Stärke ist die Möglichkeit mit dieser Technik Abhängigkeiten zwischen Entitäten zu finden, die nicht mal Teil des Programms sind, wie beispielsweise einer Klasse und ihrer Dokumentationsdatei.

Die Schwächen der Analyse der Versionshistorie liegen in der Abhängigkeit von schon gemachten Änderungen am Programm. Es können eben nur Abhängigkeiten betrachtet werden, die schonmal aufgetreten sind, nicht weil bestimmte Anweisungen die Abhängigkeit erzwingen. Kommt die Programmanalyse noch weitgehend ohne Voraussetzungen aus, ist für die Analyse der Versionhistorie die Existenz der Historie zwingende Voraussetzung. Dazu kommt, dass diese Technik bei jungen Projekten keine sinnvollen Ergebnisse liefert, da die Historie erst eine gewisse Menge von Änderungen gespeichert haben muss, bevor die Analyse der Versionshistorie Sinn macht.

Ein besonderer Fall ist die Analyse der Use Case Maps. Sie ist vor allem sinnvoll im Planungsstadium eines Projektes, und kann dort zur Verbesserung des Designs beitragen. Da eine Konsistenz zwischen Spezifikationsdokumenten und dem Programm in der Praxis wahrscheinlich nicht zu finden ist, macht die Anwendung in späteren Stadien wahrscheinlich keinen Sinn. Dieser Ansatz ist aber noch weitgehend unerforscht und empirisch nicht untersucht.

Eine generelle Schwäche aller betrachteten Untersuchungen liegt darin, dass keine Erfahrungen mit einem tatsächlichen praktischen Einsatz existieren. Jede Studie hat ihren Ansatz mit Hilfe von historischen Informationen verifiziert, und nicht im praktischen Einsatz. Deswegen gibt es auch keine Erkenntnisse darüber, ob die Vorschläge der verschiedenen Analysemethoden in der Praxis überhaupt nützlich sind, oder nicht. Die wichtigste Frage in diesem Zusammenhang ist, ob die erkannten Entitäten sowieso offensichtlich zu ändern waren, oder ob durch das eingesetzte Werkzeug wirklich unerkannte Abhängigkeiten aufgedeckt wurden.

Alles in allem bleibt die Propagierung von Änderungen ein Problem. Jeder Ansatz muss einen Kompromiss zwischen Schnelligkeit und Sicherheit der Berechnung, und zwischen Quantität und Qualität der Ergebnisse machen.

Literaturverzeichnis

[Gie02] Ingolf Giese. Softwarezuverlässigkeit gestern, heute und morgen, 2002. `http://www-aix.gsi.de/ giese/swr/ariane5.html`.

[HH04] A. Hassan and R. Holt. Predicting change propagation in software systems, 2004.

[JH05] J. Hewitt J. Hassine, J. Rilling. Change impact analysis for requirement evolution using use case maps, 2005.

[LR03] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 308–318, Washington, DC, USA, 2003. IEEE Computer Society.

[RST+04] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of Java programs. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 432–448, Vancouver, BC, Canada, October 26–28, 2004.

[RT01] Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. pages 46–53, 2001.

[ZDZ03] Thomas Zimmermann, Stephan Diehl, and Andreas Zeller. How history justifies system architecture (or not). In *Proc. International Workshop on Principles of Software Evolution (IWPSE 2003)*, Helsinki, Finland, September 2003. IEEE, IEEE Press.

[ZG04] X. Zhang and R. Gupta. Cost effective dynamic program slicing, 2004.

[Zve83] N. Zvegintzov. Nanotrends, August 1983.

[ZWDZ04] T. Zimmermann, P. Weigerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes, 2004.

Kapitel 4

Regressionstest

Mark Lehmacher

Inhaltsverzeichnis

4.1 Einleitung

Regressionstests dienen der Validierung modifizierter Software. Durch sie soll festgestellt werden, ob bereits getesteter Code als Resultat von Modifikationen neue Fehler enthält. Außerdem sollen Regressionstest Sicherheit dafür liefern, dass die durchgeführten Änderungen korrekt sind. Da mit der Spezifizierung und Durchführung von Regressionstests erhebliche Kosten verbunden sind, wurden unter anderem gewisse Auswahltechniken vorgeschlagen, um zumindest die Durchführungskosten der notwendigen Tests zu reduzieren. Diese Auswahltechniken wählen, basierend auf speziellen Kriterien, eine Teilmenge aller vorhandenen Testfälle zur Ausführung aus. Gegenstand der Betrachtung im Folgenden sind sowohl die verschiedenen allgemeinen Auswahltechniken, als auch eine spezielle Implementierung einer dieser Techniken. Außerdem werden zwei Faktoren, die Einfluss auf die Effektivität und Kosten der Auswahltechniken nehmen, exemplarisch vorgestellt. Abschließend wird das Testen in der Praxis in Hinsicht auf die Automatisierung von Testausführung und -generierung betrachtet.

4.2 Grundlagen

Softwaretests sind ein wichtiger Bestandteil der Softwareentwicklung. Sie werden eingesetzt zur teilweisen[1] Verifizierung und Validierung von Softwareelementen. Unterschieden wird auf der einen Seite zwischen Tests, die regelmäßig von Entwicklern während der Kodierung eines Softwareprodukts ausgeführt werden, wie beispielsweise Unittests. Dem gegenüber stehen Tests, die Bestandteil anderer Phasen des Softwareentwicklungsprozesses sind, wie beispielsweise Integrationstests. Im Folgenden sollen Regressionstests und Einflussfaktoren auf die Effizienz und Kosten dieser Tests genauer vorgestellt werden.

4.2.1 Regressionstests

Es gibt mehrere Definitionen und Auffassungen bezüglich Regressionstests. Im Allgemeinen versteht man unter einem Regressionstest in der Softwaretechnik die Wiederholung aller Testfälle oder einer Teilmenge davon, um Nebenwirkungen von Modifikationen auf bereits getestete Teile der Software aufzuspüren [Bea04]. Modifikationen sind all diejenigen Tätigkeiten, die mit der Pflege, Änderung und Korrektur von Softwareelementen zu tun haben. Da solche Modifikationen üblicherweise regelmäßig anfallen, müssen auch die Regressionstests ständig angewendet werden. Aufgrund dieser Tatsache ist es häufig zweckdienlich, Regressionstests automatisch durchzuführen. Ungewünschte Nebenwirkungen des Programms, die durch Regressionstests aufgedeckt wurden, nennt man Regressionsfehler.

[1] Tests können nie die vollständige Korrektheit einer Softwarekomponente garantieren (Stichwort Halteproblem)

Diese Aussage wird noch um die Beobachtung ergänzt, dass diese Nebenwirkungen oftmals Fehler betreffen, die bereits während der Entwicklung des Softwareprodukts aufgetreten und beseitigt worden sind und welche dann in Folge von Modifikationen erneut auftreten. Deswegen ist es sinnvoll, für jeden Fehler, der gefunden und beseitigt wird, einen Testfall zu schreiben, der eben diesen Fehler aufdeckt, um so spätere Regressionen zu vermeiden oder zumindest deren Entdeckung zu erleichtern.

In der Praxis stellen Regressionstests in aller Regel eine Menge von Testfällen dar, die funktionale Anforderungen auf Klassen- oder Modulebene abdecken, also nichts anderes sind als Unittests. Die Testfälle selbst müssen mit Hilfe anderer Techniken spezifiziert und mit einem Soll-Ergebnis versehen werden, das dann im Anschluss mit dem Ist-Ergebnis eines Testfalls verglichen wird. Diese Soll-Ergebnisse resultieren üblicherweise aus der Spezifikation. Ein direkter Bezug auf die Ergebnisse eines vorherigen Testdurchlaufs findet damit nicht statt.

Im Gegensatz dazu ordnet Liggesmeyer [Lig02] Regressionstests der Gruppe der diversifizierenden Tests zu. Hierbei handelt es sich um spezielle dynamische Tests während der Programmausführung, die auf Ergebnisvergleichen der Tests verschiedener Softwareversionen basieren. Dadurch wird im Unterschied zu funktionsorientierten Testtechniken die Korrektheit der Testergebnisse nicht anhand der Spezifikation entschieden, sondern durch Vergleich der Resultate der aktuellen Version des Tests mit denen des Vorgängers. Ein Testfall gilt bei diesem Verständis von Regressionstests als erfolgreich absolviert, wenn die Resultate identisch sind.

Besonderen Stellenwert genießen Softwaretests im agilen Softwareentwicklungsansatz des Extreme Programmings. Dort wird auf eine formale Softwarespezifikation verzichtet und stattdessen auf rigoroses Unittesting gesetzt. Testsuites werden intuitiv entwickelt und bilden die einzige beständige Dokumentation der Spezifikation. Aufgrund dieses Ansatzes sind die Testfälle aber auch anfällig für Unvollständigkeiten. Regressionstests können deswegen im Kontext des Extreme Programmings nicht die erhoffte Korrektheit garantieren. Stattdessen bedarf es einer speziellen Methodik, um einen gewissen Level der Zuverlässigkeit zu gewährleisten. Diese Methodik sieht vor, nach Änderungen an Softwarekomponenten diejenigen Testfälle, die die geänderten Komponenten betreffen, komplett neuzugenerieren. Eine tiefergehende Begründung und eine formale Beschreibung der angesprochenen Methodik liefert Simons in seinem Aufsatz zu diesem Thema [Sim05].

Regressionstests sind nicht zwangsläufig Entwicklertests. Auch System- oder Integrationstests können den Charakter eines Regressionstests haben. Im Weiteren wird das Hauptaugenmerk allerdings auf Regressionstests in Form von Unittests gelegt. Das sind Tests, welche bereits während der Kodierungsphase der Software entstehen (beispielsweise bei Anwendung von testgetriebener Entwicklung) und Sofwarekomponenten auf Modul- oder Klassenebene testen. Im Zuge der Wartung der Software und Wiederausführung der Tests als Reaktion auf Modifikationen, erhalten diese Tests dann Regressionstestcharakter.

Im Folgenden werden Regressionstests losgelöst von speziellen Entwicklungsparadig-

men wie etwa des Extreme Programmings betrachtet und Strategien zur Verbesserung der Effektivität dieser Tests vorgestellt. Am Anschluss daran folgt eine beispielhafte Betrachtung zweier Rahmenwerke für Unittests, an denen die Praxis des automatisierten Testens verdeutlicht wird.

4.2.2 Regressionstest-Auswahltechniken

Wie bereits erwähnt, werden in der Praxis nach Modifikationen an einer Softwarekomponente alle oder eine Teilmenge aller Testfälle ausgewählt und durchgeführt. Oftmals ist es allerdings nicht effizient, alle Testfälle auszuführen, weil dies zu lange dauern würde, oder weil ein Großteil der Testfälle nicht die modifizierte Komponente abdeckt. Mit Regressionstests verbundene Kosten (Erstellung, Durchführung und Auswertung) machen Abschätzungen zufolge bis zur Hälfte der gesamten Wartungskosten eines Softwareentwicklungsprojekts aus [RUCH01]. Auch deswegen kann es zweckmäßig sein, nur eine Teilmenge aller vorhandenen Testfälle auszuwählen, um so die Kosten, die bei der Ausführung von Testfällen auftreten, zu reduzieren. Formal hat ein Regressionstest dann folgenden Ablauf:

Sei P ein Programm und P' eine modifizierte Version von P. T sei Testsuite für P.

1. Wähle eine Menge von Testfällen $T' \subseteq T$ zum Ausführen auf P'.
2. Teste P' mit T', um die Korrektheit von P' in Bezug auf T' festzustellen.
3. Falls notwendig, erstelle eine neue Menge Testfälle T'' für P'.
4. Teste P' mit T'', um die Korrektheit von P' in Bezug auf T'' festzustellen.
5. Erstelle eine neue Testsuite T''' für P' aus T, T' und T'' und dokumentiere durchgeführte Änderungen.

Jeder dieser Schritte behandelt wichtige Probleme. Schritt 1 betrifft das Auswahlproblem einer Teilmenge von Testfällen T' aus T zum Testen von P'. Schritt 3 formuliert das Abdeckungsproblem: das Problem der Identifikation der Teile von P' oder der Spezifikation von P', die weitere und genauere Tests benötigen. Den Schritten 2 und 4 liegt das Ausführungsproblem zu Grunde, d.h. Testsuites effizient auszuführen und ihre Resultate auf Korrektheit zu prüfen. Schritt 5 betrifft das Problem der Wartung der Testsuites, insbesondere die Aktualisierung und Sicherung von Testinformationen und der Dokumentation dieser Änderungen. Im Weiteren werden verschiedene Auswahltechniken vorgestellt. Damit liegt der Schwerpunkt der folgenden Betrachtungen also auf Schritt 1.

Alle vorgestellten Auswahltechniken funktionieren sowohl in den Fällen veränderter als auch unveränderter Spezifikation. Im ersten Fall müssen alle Testfälle aus T entfernt werden, die durch die Änderung für P' obsolet geworden sind. Daraufhin können

die Regressionstests wie üblich durchgeführt werden. Allerdings muss man grundsätzlich in der Lage sein, nicht mehr benötigte Testfälle zu identifizieren, andernfalls kann man nicht sicher die Korrektheit von Testfällen beurteilen [GHK+01].

Generell gibt es drei Gruppen von Auswahltechniken und außerdem zwei zusätzliche Ansätze, welche gerade in der Praxis häufig eingesetzt werden.

Generell gibt es die folgenden fünf Typen von Auswahltechniken, wobei der Begriff der sicheren Technik etwas aus dem Rahmen fällt, weil deren Definition auf einer Eigenschaft einer Technik und keinem technischen Ansatz beruht.

4.2.2.1 Minimierungstechniken

Minimierungstechniken versuchen eine minimale Teilmenge von Tests aus T zu wählen, so dass alle modifizierten oder von Modifikationen betroffenen Stellen von P durch diese Testfälle abgedeckt werden. Zielkriterium ist hier also eine maximale Anweisungsüberdeckung der modifizierten Programmteile bei möglichst geringer Anzahl von Testfällen. Bezogen auf P' bedeutet dies, dass diese Technik mindestens einen Testfall auswählt, der alle Programmanweisungen, die hinzugefügt oder modifiziert wurden, überprüft.

4.2.2.2 Datenflusstechniken

Datenflusstechniken selektieren Testfälle, welche Dateninteraktionen testen, die von Modifikationen betroffen sind. Dateninteraktionen sind Muster von Datenbenutzungen in einem System, welche in drei Klassifikationen eingeteilt werden können:

- die Definition einer Variablen oder der Zuweisung eines Wertes an eine Variable
- die Benutzung einer Variablen in einem Ausdruck, wie einer Berechnung oder einem Funktionsrückgabewert
- die Benutzung einer Variablen in einer Bedingung und somit Einfluss der Variablen auf den Kontrollfluss

Basierend auf diesen drei Klassifikationen gibt es verschiedene mögliche Testabdeckungen, die durch Testfälle realisiert werden können. Je nach angestrebter Abdeckung wählt eine Datenflusstechnik die Testfälle aus, die unter dem gewählten Kriterium in Bezug auf den Datenfluss, die von Modifikationen betroffenen Programmteile überprüft.

4.2.2.3 Sichere Techniken

Die meisten Regressionstest-Auswahltechniken (inkl. Minimierungs- und Datenflusstechniken) sind nicht garantiert sicher. Solche problematischen Techniken können nicht gewährleisten, dass alle Testfälle, die mindestens einen Fehler erkennen, ausgewählt werden. Anders ausgedrückt heißt dies, dass eine unsichere Technik einen Testfall, der einen Fehler erkennt, nicht notwendigerweise auswählt. Im Gegensatz dazu garantiert eine sichere Technik, dass die Teilmenge T' mindestens all diejenigen Testfälle von T enthält, welche Fehler auf P' aufdecken. In Abschnitt 4.2.4) wird exemplarisch die sichere Technik DejaVu vorgestellt.

4.2.2.4 Retest-All Ansatz

Bei der Anwendung dieses Ansatzes werden einfach alle vorhanden Testfälle aus T ausgewählt, um die Korrektheit von P' zu testen. Offensichtlich ist der Retest-All Ansatz damit insbesondere eine sichere Technik.

Aufgrund seiner Einfachheit ist der Retest-All Ansatz gerade in der Praxis besonders verbreitet.

4.2.2.5 Ad-hoc/Zufällige Auswahl

Falls sich ein Einsatz von Retest-All verbietet, beispielsweise aufgrund von Zeiteinschränkungen, und sich keine andere Auswahltechnik anbietet, wählen Entwickler Testfälle häufig ”aus dem Bauch heraus” aus. Alternativ basieren Entwickler ihre Auswahlentscheidungen auf Grundlage ihres Wissens über (lose) Zusammenhänge zwischen Testfällen und Funktionalitäten. Dies sind dann sogenannte Ad-hoc Auswahlen von Testfällen. Auch dieser Ansatz wird häufig in der Praxis eingesetzt.

Ein anderer ganz einfacher Ansatz ist die zufällige Wahl einer bestimmten Menge von Testfällen aus T, z.B. anhand eines fixen Prozentsatzes, nach dem Testfälle aus T ausgewählt werden.

4.2.3 Kosten- und Nutzenmodell

Um den Einsatz verschiedener Techniken vergleichbar zu machen, bedarf es eines Modells. Leung und White haben ein einfaches Modell zur Kosten- und Nutzenabwägung von Auswahltechniken entwickelt [LW91]. Kosten sind unterteilt in direkte und indirekte Kosten. Indirekte Kosten sind Kosten, die mit Management Overhead, Datenbankwartung und der Entwicklung unterstützender Werkzeuge verbunden sind. Direkte Kosten sind Kosten, die mit dem Testprozess an sich zusammenhängen, wie beispielsweise die Auswahl der Testfälle, deren Ausführung und die Analyse der Er-

gebnisse der durchgeführten Testfälle. Einsparungen sind die Kosten, die durch nicht ausgewählte und damit nicht ausgeführte Testfälle vermieden werden.

Formal sieht der Sachverhalt dann wie folgt aus: T' ist eine Teilmenge der gesamten Testfälle T, die von einer Technik M zum Testen des Programms P ausgewählt wird. $|T'|$ ist die Anzahl der Testfälle in T'. s sind die durchschnittlichen Kosten pro Testfall, die die Anwendung von M auf P zu Ermittlung von T' verursacht. r sind die durchschnittlichen Kosten pro Testfall, die die Ausführung von P auf einem Testfall von T, inklusive der Auswertung des Resultats dieses Testlaufs, verursacht. Damit eine Auswahltechnik effektiv ist, muss folgende Ungleichung gelten: $s|T'| < r(|T| - |T'|)$. Die notwendige Analyse von P zur Bestimmung von T' muss günstiger sein als die Ausführung und Analyse der nicht ausgewählten Testfälle $|T| - |T'|$. Beachten muss man bei der Anwendung dieses Modells allerdings, dass Fehler, die zwar von T aber nicht von T' erkannt werden, nicht als Kosten erfasst werden. Da der primäre Zweck des Testens allerdings darin besteht, Fehler zu entdecken, muss genau abgewogen werden, in welchem Maße man Einbußen in der Fehlererkennungseffizienz durch den Einsatz von (unsicheren) Auswahltechniken in Kauf nimmt.

4.2.4 DejaVu - Beispiel einer sicheren Auswahltechnik

DejaVu ist eine Implementierung einer sicheren Auswahltechnik für C-Programme [RH97]. DejaVu generiert für zwei Programmversionen P und P' Repräsentationen aller Prozeduren dieser Programme als Kontrollflussgraphen. Die Knoten eines Graphen sind beschriftet mit den Anweisungen der respektiven Programme. Dieser Ansatz basiert auf bereits vorhandenen Testergebnissen, die festhalten, ob für jeden Test t aus T und jede Kante e des Kontrollflussgraphen von P, t die Kante e traversiert. Auf Basis dieser Daten instrumentiert DejaVu den Quellcode des zu testenden Systems durch Hinzufügen von eigenem Code.

Danach traversiert DejaVu per Tiefensuche parallel für jede Prozedur aus P und ihr modifiziertes Gegenstück aus P' die beiden Kontrollflussgraphen. Dabei weisen Zeiger auf den jeweils aktuellen Knoten des jeweiligen Graphen. Während der Traversierung untersucht DejaVu die Knoten (Anweisungen) der beiden Graphen und die Kanten (Kontrollfluss), die diese Knoten verlassen. Wenn diese identisch sind, geht der Algorithmus zum nächsten Nachfolgeknoten über, andernfalls wird die Kante, über die der aktuelle Knoten erreicht wurde, als ”gefährliche Kante” markiert. Daraufhin endet die aktuelle Rekursion und der Algorithmus kehrt zum Ausgangsknoten der gefährlichen Kante zurück. Die Traversierung des Graphen endet, wenn alle gefährlichen Kanten auf diesem Wege identifiziert wurden. Zum Testen von P' werden jetzt alle Testfälle t aus T ausgewählt, die im Kontrollflussgraphen von P mindestens eine gefährliche Kante abdecken.

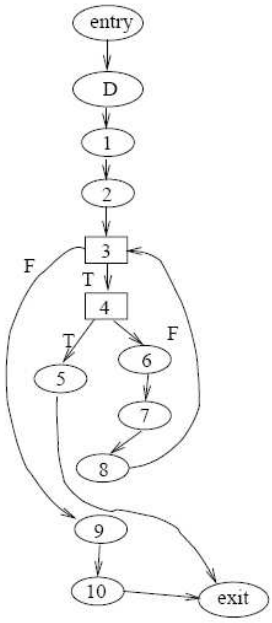
Abbildung 4.2.4 illustriert den Algorithmus an einem Beispiel. Dargestellt werden das Programm avg mit dem dazugehörigen Kontrollflussgraphen (links) und das modifizierte Programm avg’ mit seinem Kontrollflussgraphen (rechts). Die Unterschiede zwischen avg und avg’ sind in der hinzugefügten Anweisung 5a und der entfernten

```
procedure avg

1. int count = 0
2. fread(fileptr,n)
3. while (not EOF) do
4.     if (n<0)
5.         return(error)
       else
6.         numarray[count] = n
7.         count++
       endif
8.     fread(fileptr,n)
   endwhile
9. avg = calcavg(numarray,count)
10. return(avg)
```

```
procedure avg'

1. int count = 0
2. fread(fileptr,n)
3. while (not EOF) do
4.         if (n<0)
5a.            print("bad input")
5.             return(error)
           else
6.             numarray[count] = n
           endif
8.         fread(fileptr,n)
   endwhile
9. avg = calcavg(numarray,count)
10. return(avg)
```



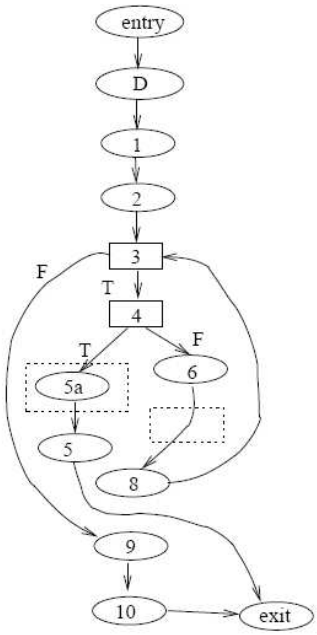


Abbildung 4.1: Programme avg und avg' und deren Kontrollflussgraphen

Anweisung 7 zu finden. Der Algorithmus beginnt die Traversierung des Graphen an den respektiven Entry Knoten und folgt identischen Pfaden in beiden Graphen, indem er identisch beschrifteten Kanten solange folgt, wie diese identische Folgeknoten besitzen. Wenn der Algorithmus den Knoten 4 in avg und avg' betrachtet, stellt er fest, dass die Zielknoten der mit "T" beschrifteten Kante unterschiedlich sind und fügt die Kante (4, 5) der Menge der gefährlichen Kanten hinzu. Jetzt bricht der Algorithmus die Traversierung des aktuellen Pfades ab: jede denkbare weitere gefährliche Kante in diesem Pfad kann nur über die Traversierung der aktuellen gefährlichen Kante erreicht werden. Deswegen deckt jeder Testfall, der die Kante (4, 5) abdeckt, zwangsläufig auch die folgenden gefährliche Kanten in diesem Pfad des Graphen ab. Danach wird der Algorithmus die mit "F" beschrifteten und in beiden Programmen aus dem Knoten 4 herausgehenden Kanten untersuchen. Wenn die Knoten 6 der beiden Programme verglichen werden, entdeckt der Algorithmus, dass die Folgeknoten der ausgehenden Kanten aus Knoten 6 unterschiedlich sind. Die Menge der gefährlichen Kanten wird nun um die Kante (6, 7) ergänzt und der Algorithmus bricht die Traversierung des aktuellen Pfades ab. Die folgenden Traversierungen entdecken keine gefährlichen Kanten mehr.

Im Beispiel sind die mit "D" beschrifteten Knoten so genannte Deklarationsknoten. Diese dienen dem Algorithmus zur Modellierung von Variablendefinitionen. Solche Knoten enthalten Informationen über alle Deklarationen. Auf diese Weise kann der Algorithmus Modifikationen, die nur die Deklaration von Variablen betreffen, aber nicht im Kontrollflussgraphen reflektiert werden, identifizieren. Wenn beispielsweise die Typendeklaration der Variablen count von int auf long geändert wird, entdeckt der Algorithmus eine Änderung der beiden Deklarationsknoten und fügt die Kante (entry, D) der Menge der gefährlichen Kanten hinzu.

Nachdem alle gefährlichen Kanten aufgespürt wurden, wird eine Abdeckungsmatrix konsultiert, um eine Menge von Testfällen zu selektieren. Auf das Beispiel bezogen, soll die Testsuite aus Tabelle 4.1 für avg vorhanden sein.

Testfall	Eingabe	Erwartete Ausgabe
1	leere Datei	0
2	-1	Fehler
3	1 2 3	2

Tabelle 4.1: Testsuite für das Programm avg aus 4.2.4

Während der Ausführung des Programms werden die von den jeweiligen Testfällen abgedeckten Kanten festgehalten. Für dieses Beispiel decken die Testfälle 1, 2 und 3 die Kante (entry, 1) ab, die Testfälle 1 und 3 die Kanten (2, 9) und die Testfälle 2 und 3 die Kante (3, 4). Tabelle 4.2 repräsentiert die Kantenabdeckungsmatrix für diese Testsuite und das Programm avg.

Der letzte Schritt besteht nun in der Auswahl der Testfälle, die auf der Menge der gefährlichen Kanten und der Kantenabdeckungsmatrix basiert. An Hand der Matrix wird deutlich, wie die gefährlichen Kanten (4, 5) und (6, 7) durch die Testfälle 2 und 3

Kante	Testfall
(entry, 1), (1, 2), (2, 3)	1, 2, 3
(3, 9), (9, 10), (10, exit)	1, 3
(3, 4)	(2, 3)
(4, 5), (5, exit)	2
(4, 6), (6, 7), (7, 8), (8, 3)	3

Tabelle 4.2: Kantenabdeckungsmatrix für die Testsuite aus Tabelle 4.1 und das Programm avg aus 4.2.4

abgedeckt werden. Demnach sind diese beiden Testfälle Ergebnis der Anwendung der sicheren Auswahltechnik DejaVu auf das modifizierte Programm avg'.

Damit DejaVu Sicherheit (für formalen Beweis siehe [RH97]) gewährleisten kann, müssen die folgenden drei Prämissen erfüllt sein:

1. Die Testfälle aus T ergeben korrekte Resultate, wenn sie auf P ausgeführt werden.
2. Alle für P' obsoleten Testfälle aus T wurden entfernt.
3. Wiederholte Anwendungen von T auf P und P' ergeben deterministische Resultate,

In einer konkreten Anwendung von DejaVu auf einige kleinere Programme (100-500 Zeilen Code) und ein umfangreicheres Programm (50.000 Zeilen Code) wurden Kostenersparnisse von 45% für die kleineren und 95% für das große Programm festgestellt (basierend auf dem Modell von Leung und White).

Harrold stellt, aufbauend auf der Idee der Kontrollflussgraphentraversierung, eine Implementierung einer sicheren Auswahltechnik für Java Programme vor [HJL+01]: *RETEST*. RETEST unterstützt spezifische Eigenschaften von Java, bzw. generelle Eigenschaften objektorientierter Sprachen. Konkret handelt es sich dabei um Polymorphismus, das dynamische Binden und die Behandlung von Ausnahmen. Der Algorithmus kann auf beliebige Java Programme angewendet werden. Eine Analyse der benutzten Libraries der zu untersuchenden und instrumentierenden Programme P und P' erfolgt nicht, so dass die Analysekosten unabhängig von deren Einsatz sind.

4.2.5 Vergleich der Auswahltechniken

In einer empirischen Untersuchung haben Graves et al. [GHK+01] die fünf verschiedenen Regressionstest-Auswahltechniken (siehe Abschnitt 4.2.2), basierend auf dem Modell von Leung und White, bezüglich ihrer Kosten und Nutzen untersucht. Besonderes Augenmerk wurde dabei auf die Verringerung der Testsuitegröße und die Feh-

lererkennung der reduzierten Testsuite gelegt. Ergebnis der Studie waren die folgenden Feststellungen, die allerdings nicht als allgemeingültig verstanden werden sollten. Trotzdem zeigen sie schon deutliche Tendenzen auf und dienen als Ausgangspunkt für weitere Forschungen.

- Die Anwendung der Minimierungstechnik resultiert in den kleinsten und ineffizientesten Testsuites. In Fällen, wo das Testen selber sehr teuer ist, mag die Minimierungstechnik kosteneffektiv sein, selbst bei großer Wichtigkeit der Fehlererkennung. Allerdings ergeben sich durch eine zufällige Auswahl von Testfällen nur unwesentlich größere Testsuites, die aber bezüglich der Fehlererkennung, den resultierenden Testsuites der Minimierungstechnik ebenbürtig sind, wohlgemerkt bei geringeren Analysekosten.

- Sichere Techniken (eine davon ist DejaVu) und Datenflusstechniken[2] haben beide fast identische Verhaltensweisen was Testauswahl und Fehlererkennung angeht. Beide resultieren üblicherweise in Testsuites derselben Größe und erkennen dieselben Fehler. Datenflusstechniken haben immer höhere Analysekosten als die besten beiden sicheren Techniken. Daher sind sie für reine Testauswahl generell nicht zu empfehlen. Allerdings kann der Einsatz einer Datenflusstechnik auch Vorteile mit sich bringen, die nicht durch das Kosten-Nutzen-Modell erfasst werden. Die Anwendung einer Datenflusstechnik kann beispielsweise helfen, Teile von P' zu identifizieren, die nicht adäquat von der Testsuite T getestet werden (Schritt drei des formalen Regressionstestablaufs).

- Sichere Techniken finden alle Fehler (per Definition), die durch vorhandene Testfälle aus T aufgedeckt werden können. Im Durchschnitt werden dabei 60% aller Testfälle aus T ausgewählt. Allerdings gibt es auch Situationen, abhängig von der Struktur der zu testenden Programme und der vorhandenen Testfälle, in denen gar keine Reduktion der Testsuitegröße festgestellt wurde. Zudem sind generell nur geringfügig größere Testsuites, die durch zufällige Auswahl zu Stande kommen, fast so effektiv wie die ”sicheren” Testsuites.

- Grundsätzlich sind die jeweiligen Ergebnisse nicht nur abhängig von der gewählten Auswahltechnik, sondern auch von Struktur und Umfang des Programms, der Zusammensetzung der Testsuites, der speziellen Testweise, sowie der speziellen Charakteristika der Änderungen an P, die zu P' führen. Dies führt zum Problem der Interpretierbarkeit der Ergebnisse, da manche Effekte fälschlicherweise als Eigenschaften einer speziellen Auswahltechnik interpretiert werden könnten.

Im nächsten Abschnitt soll näher auf die Probleme des letzten Punktes eingegangen werden, also insbesondere der Frage auf den Grund gegangen werden, welche sonstigen Faktoren Einfluss auf die Kosten und die Effektivität von Auswahltechniken nehmen.

[2] DejaVu basiert auf Kontrollflussansatz, daher der Zusammenhang

4.3 Einflussfaktoren

Wie bereits erwähnt, hängt die Effektivität einer Auswahltechnik nicht nur von den technischen Eigenschaften der Technik selber ab, sondern auch von anderen Einflussfaktoren. An dieser Stelle sollen exemplarisch zwei Faktoren und ihre Auswirkungen auf die Effektivität der verschiedenen Auswahlverfahren vorgestellt werden.

4.3.1 Intervalllänge zwischen Tests

Die Intervalllänge zwischen Tests ist gegeben durch die Anzahl von Modifikationen, die ein Entwickler vornimmt, bevor er den nächsten Regressionstest durchführt. In ihrer Studie stellen Kim et al. folgende zwei Hypothesen auf [KPR03]:

- Die Größe der ausgewählten Testsuite T' für P' ist abhängig von der Häufigkeit, in der die Testsuites nach Modifikationen ausgeführt werden.
- Die Effektivität bezüglich der Fehlererkennung der ausgeführten Testsuites ist ebenfalls abhängig von dieser Häufigkeit.

Anders formuliert wird hier also der Einfluss von unterschiedlichen Intervalllängen zwischen Regressionstests auf Kosten und Nutzen von Auswahltechniken thematisiert. In der Studie wird im Folgenden ein Experiment mit Programmen und Testfällen verschiedener Struktur und verschiedenen Umfangs durchgeführt. Die Ergebnisse und Schlussfolgerungen, wenn auch erneut nicht verallgemeinerbar, sind dennoch interessant und deuten darauf hin, dass beide Hypothesen wahr sind. Von der Datenflusstechnik abgesehen, sind alle Techniken Bestandteil des Experiments. Als sichere Technik wird unter anderem die DejaVu Implementierung verwendet. Ergebnis des Experiments sind folgende Erkenntnisse:

- Je länger das Intervall zwischen Tests ist (also je mehr Modifikationen durchgeführt werden, bevor tatsächlich getestet wird), desto weniger Fehler werden durch die Ausgangstestsuite aufgedeckt. Dies betrifft alle eingesetzten Auswahltechniken. Grund dafür sind mögliche Interaktionen zwischen verschiedenen Fehlern, die die Fehlerentdeckung erschweren. Formal formuliert kann man festhalten, dass bei wachsenden Testintervalllängen die Informationen über ein Programm P und die Testsuite T immer mehr Aussagekraft über den Zustand von P' verlieren.
- Nicht überraschend selektierten die sicheren Techniken, bei gleicher Effektivität, weniger als oder gleich viele Testfälle wie Retest-All. Aufgrund dessen sind sichere Techniken solange der Retest-All Anwendung vorzuziehen, wie die Analysekosten geringer sind als die Kosten der nicht ausgeführten Tests (nach Modell von Leung und White). Allerdings wurden bei wachsender Intervalllänge zwischen Tests immer mehr Testfälle von den untersuchten sicheren

Techniken ausgewählt, so dass in diesen Fällen der Einsatz von Retest-All anzuraten ist.

- Eine zufällige Auswahl von Testfällen ist überraschend günstig und dabei trotzdem effektiv. Dies gilt sowohl für relativ geringe Abstände zwischen aufeinander folgenden Tests als auch für größere Intervalllängen. Interessanterweise approximiert die Effektivität dieses Ansatzes bei wachsenden Intervalllängen die der Anwendung von Retest-All. Insbesondere enstehen dann auch weniger Abweichungen zwischen verschiedenen Testläufen mit Testfällen eines fixen Prozentsatzes. Das Experiment hat mit Prozentsätzen von 25% und 50% zur zufälligen Auswahl der Testfälle der Ausgangssuite gearbeitet.

- Bezüglich der Minimierungstechnik konnte ein großer Unterschied zwischen der Anwendung auf kurze und lange Intervalle festgestellt werden. Bei kurzen Intervallen wurden durchgängig nur ein oder zwei Testfälle ausgewählt und die Effektivität der Fehlerentdeckung betrug im Schnitt nur 15% der Quote von Retest-All. Beim anderen Extrem werden vier bis fünf Testfälle ausgewählt (insgesamt nur 2% aller Testfälle) und die Effektivität wächst auf immerhin 60% der Effektivität von Retest-All. Aufgrund dessen mag es empfehlenswert sein, die Minimierungstechnik auf teure Testfälle und große Intervalllängen anzuwenden.

- Auch in dieser Studie wurde wieder festgestellt, dass die Struktur von Programmen und Testfällen einen großen Einfluss auf die Leistung der Auswahltechniken haben.

4.3.2 Priorisierung von Testfällen

Bezüglich der Struktur von Testsuites konnte festgestellt werden, dass die Reihenfolge in der die einzelnen Testfälle einer Testsuite ausgeführt werden, großen Einfluss auf die Effektivität der Fehlererkennung haben kann [RUCH01]. In welcher Reihenfolge die Fälle abgearbeitet werden, hängt von den Zielen des Entwicklers ab. Folgende Ziele sind denkbar:

- Eine Verbesserung der Fehlererkennungseffektivität der Testsuite - die Wahrscheinlichkeit, Fehler im Laufe des Regressionstestprozesses früher zu erkennen.

- Eine Verbesserung der frühen Abdeckung des Codes eines Systems, um eine gewisse Codeabdeckungsrate schneller zu erreichen.

- Eine Verbesserung der Wahrscheinlichkeit, Fehler, als Folge bestimmter Codemodifikationen, früher zu erkennen.

Diese Ziele sind rein qualitativ und dienen alle der frühen Steigerung des Vertrauens in die Zuverlässigkeit eines Systems. [RUCH01] beschreibt eine Methodik um den Erfolg oder Misserfolg der Priorisierungsstrategien zu quantifizieren. Außerdem werden

Techniken zur Priorisierung von Testfällen und deren Einsatzkriterien vorgestellt. Konkret bestimmen diese Techniken die Reihenfolge von Testfällen, entweder basierend auf

- deren Abdeckung von Code Komponenten, oder
- deren Abdeckung von Code Komponenten, die nicht bereits durch vorherige Testfälle abgedeckt sind, bzw.
- deren geschätzter Fähigkeit, Fehler aufzudecken.

Auf Grundlage dieser Techniken werden einige Experimente durchgeführt. Dabei werden die Ergebnisse der Anwendung der Techniken auf verschiedene Testsuites und Programme, den Ergebnissen, die der Einsatz von zufällig georderten Testsuites und optimal georderten Testsuites liefert, gegenübergestellt. Die Analyse der Daten bestätigt, dass selbst der Einsatz der günstigsten Priorisierungstechnik (bezüglich der Analysekosten), eine Verbesserung der Fehlererkennungseffektivität bewirkt. Eine Kombination mit verschiedenen Auswahltechniken wurde nicht durchgeführt, aber als möglicher Ausgangspunkt für weitere Forschung erwähnt.

4.4 Regressionstests in der Praxis

Wie bereits dargestellt, werden in der industriellen Praxis überwiegend eine Ad-Hoc Auswahl von Testfällen und Retest-All betrieben, häufig auch in Kombination. In diesem Fall wird der Entwickler schon während der Entwicklung und nach einer überschaubaren Anzahl zusammenhängender Modifikationen eine Menge von Tests ausführen. Die Wahl der konkreten Testfälle wird dabei abhängig sein von den Veränderungen, die der Entwickler vorgenommen hat sowie von seinem Wissen über die Zusammenhänge im System. Er wird also insbesondere versuchen, nur die Testfälle zu wählen, die irgendwie von den durchgeführten Abwandlungen betroffen sind. Bevor er seine akkumulierten Modifikationen dann in ein Versionsmanagementsystem eincheckt, wird er üblicherweise noch einmal alle Testfälle ausführen.

Zur Unterstützung des automatisierten Ausführens von Testfällen gibt es Rahmenwerke, die es ermöglichen, ein oder mehrere Testfälle bzw. Testsuites auszuwählen und diese dann auszuführen. Im Anschluss daran erhält man eine aggregierte Zusammenfassung, wie viele der Tests fehlgeschlagen sind und kann bei Bedarf für jeden gescheiterten Test den Grund für das Scheitern überprüfen. Beide hier vorgestellten Rahmenwerke dienen dem Testen von Java Code, wobei es von JUnit auch Implementierungen für viele andere Sprachen wie c oder Python gibt. Beide dieser Rahmenwerke automatisieren Tests nur insofern, als dass eine Menge von Testfällen automatisch ausgeführt wird, eine automatische Auswahltechnik wird von keinem der beiden Rahmenwerke realisiert. TestNG gibt dem Entwickler immerhin feingranulare Mechanismen zur manuellen Auswahl von Testfällen in die Hand.

4.4.1 JUnit

JUnit [jun] ist das etablierte Open Source Regressionstest-Rahmenwerk in der Java Welt. JUnit wurde von Erich Gamma und Kent Beck entwickelt und wird von Entwicklern zur Implementierung von Unittests eingesetzt. Kennzeichnend für JUnit Testklassen ist die strikte Separierung zwischen einzelnen Testfällen. Alle Testfälle werden hintereinander, in einer nicht-definierten Reihenfolge ausgeführt. Für jeden Testfall wird vorausgesetzt, dass dieser seiteneffektfrei abläuft. Insbesondere ist es auch nicht vorgesehen, dass Initialisierungsmethoden auf Klassenebene globale Fixtures für die Testfälle der aktuellen Testklasse aufsetzen. Fixtures sind Konfigurationen, deren Verhalten vorhersehbar ist. Dies können beispielsweise Objekte sein, die von Testfällen benutzt werden. Statt dessen werden diese Fixtures vor jedem Testfall wiederholt initialisiert und der Zustand des vorherigen Testfalls, bzw. dessen Umgebung, verworfen. Aufgrund der Einsicht, dass in der Praxis auch Seiteneffekte zwischen verschiedenen Testfällen wünschenswert sein können, wurden in der aktuellen Version 4.0 von JUnit einige dieser Beschränkungen aufgeweicht.

4.4.2 TestNG

TestNG [tes] ist ebenso wie JUnit ein Open Source Projekt und bietet eine Obermenge dessen Funktionalitäten. Insbesondere können JUnit Testsuites auch problemlos von TestNG ausgeführt werden. Außerdem löst es sich von dem rigiden JUnit Paradigma der Seiteneffektfreiheit der einzelnen Testfälle. Als Ergebnis dessen stellt das TestNG Rahmenwerk dem Entwickler viele nützliche Erweiterungen zur Verfügung.

Jeder Testfall kann zu einer oder mehreren Gruppen gehören. Gruppen von Testfällen können selektiv ausgeführt werden, ohne dass die komplette Testsuite durchgeführt werden muss. Auf diese Weise kann man einen manuellen Einsatz der Ad-Hoc Auswahl von Testfällen realisieren, indem man Testfälle beispielsweise basierend auf technischen Merkmalen gruppiert. Andererseits kann man aber auch die Testreihenfolge, durch geschickte Zuordnung von Testfällen zu Gruppen priorisieren. Sowohl für Gruppen von Testfällen als auch für einzelne Testfälle kann man andere Gruppen oder Testfälle angeben, die vor der Ausführung der aktuellen Gruppe bzw. des aktuellen Testfalls bereits erfolgreich abgelaufen sein müssen. Andernfalls wird der aktuelle Test übersprungen.

Darüber hinaus bietet TestNG gegenüber JUnit weitere Vorteile, die allerdings rein technischer Natur sind. Erwähnenswert ist etwa die Möglichkeit, einzelne Testfälle gleichzeitig in mehreren Threads auszuführen, um auf diese Art und Weise Code auf Threadsicherheit zu testen.

In der Java Community erfreut sich TestNG wachsender Beliebtheit, unter anderem auch deshalb, weil die Entwicklung schneller voranschreitet als die von JUnit.

4.4.3 Generierung von Testfällen

Abgesehen von Frameworks wie JUnit und TestNG, die die Durchführung und Auswertung von Tests weitestgehend automatisieren, gibt es außerdem Werkzeuge, die den Entwickler durch automatische Testgenerierung unterstützen. Ein Vertreter dieser Klasse von Werkzeugen ist *Parasoft JTest* [jte]. Im Gegensatz zu JUnit und TestNG ist JTest nicht Open Source, sondern ein kommerzielles Softwareprodukt. JTest unterstützt Entwickler auf vielfältige Art und Weise. Es führt Code- und Anweisungsüberdeckungsanalysen durch und bietet viele Hilfen im Umgang mit Tests. Darunter fallen die Generierung von Unittests, die Ausführung der Tests und das Nachhalten von Tests und deren Resultate. JTest ist in der Lage, funktionale Tests für beliebige Java Anwendungen zu generieren. Diese Tests sind JUnit kompatibel.

Die JTest Anwendungsanalyse, auf dessen Grundlage die Tests entstehen, lässt sich grob in zwei Phasen unterteilen:

4.4.3.1 Statische Analyse

Basis der statischen Analyse sind einmal der Programmcode der untersuchten Anwendung und zum anderen etwa 500 Regeln, die auf den Erfahrungen von Experten und verschiedenen Kodierungsstandards basieren. Die Regeln können individuell angepasst werden, so dass nur eine bestimmte, frei wählbare Teilmenge auf den zu analysierenden Code angewendet wird. Im Falle von Regelverletzungen werden nicht nur die Verletzungen selber angezeigt, sondern darüber hinaus auch weitere Informationen, wie beispielsweise eine Motivation, die erläutert, warum die Befolgung der aktuellen Regel sinnvoll ist, zusammen mit Darstellungen, die dies illustrieren. Außerdem gibt es zu jeder Regel eine Quellenangabe, die deren Herkunft beschreibt.

4.4.3.2 Dynamische Analyse

Im Zuge der dynamischen Analyse werden die zu testenden Softwarekomponenten durch den Entwickler oder einen Mitarbeiter der Softwarequalitätssicherung ausgeführt und wie von einem Endbenutzer bedient. JTest protokolliert das Verhalten der Komponente auf Ebene des Programmcodes. Insbesondere hält JTest fest, welcher Pfad durch den Programmcode genommen wurde und erstellt auf dieser Grundlage vollautomatisch Testfälle, die diesen Pfad reproduzieren. In anderen Worten werden Benutzerinteraktionen auf Codeebene ”aufgezeichnet”, um sie später wieder ”abspielen” zu können. Dieser Ansatz ist besonders geeignet zur Erstellung von Regressionstests für GUI Anwendungen.

JTest unterstützt auch ”Design by Contract”. Dabei werden für jede Methode deren Vor- und Nachbedingungen in dem zu dieser Methode gehörenden Javadoc-Block angegeben. Diese Bedingungen werden von JTest zur intelligenten Generierung von Testfällen herangezogen. Diesbezüglich gibt es auch wissenschaftliche Ansätze, bei-

spielsweise das Rahmenwerk *TestEra* [MK01]. Auch TestEra realisiert automatisiertes Testen von Java Methoden, basierend auf deren Vor- und Nachbedingungen.

Wenn keine Vor- und Nachbedingungen vorhanden sind, werden Testfälle auf Grundlage der Signatur der zu testenden Methode erstellt. Dabei werden bestimmte Werte für die Parameter der Methode eingesetzt. Im Falle eines primitiven Typen, wie beispielsweise *int*, sind dies defaultmäßig die Werte -1, 0, 1. Für *String* Objekte werden sowohl null Werte als auch der leere String verwendet. Grundsätzlich sind die verwendeten Werte vom Entwickler konfigurierbar. Dieser Generierungsansatz ist offensichtlich nur auf das Erstellen von black-box Tests ausgelegt. Dies ist auch der Grund, warum die solcher Art erstellten Testfälle häufig nur als Ausgangspunkt für weitere Testfälle, die manuell durch den Entwickler erstellt werden müssen, dienen.

Natürlich kann JTest einem Entwickler die Arbeit, Testfälle zu entwerfen nicht vollständig abnehmen, aber als unterstützendes Werkzeug ist es sicherlich sinnvoll einsetzbar, gerade auch in großen Projekten, um Kodierungskonventionen zu forcieren.

4.5 Zusammenfassung

Ziel dieser Ausarbeitung ist es, einen Überblick über das Themengebiet der Regressionstests zu geben. Diese werden durchgeführt, um Softwarekomponenten zu validieren und zugleich sicherzustellen, dass Modifikationen an Software keine unerwünschten Seiteneffekte (sprich Fehler) verursachen. Dieser Prozess, der sowohl die Erstellung als auch die Durchführung und Auswertung von Testfällen einschließt, ist sehr teuer. Ansätze, um zumindest die Durchführungskosten zu reduzieren, wurden in Form von fünf verschiedenen Auswahltechniken vorgestellt (siehe Abschnitt 4.2.2). Konkret sind dies die Minimalisierungs- und Datenflusstechniken sowie der Retest-All Ansatz, die Ad-Hoc Auswahl und die zufällige Selektion von Testfällen. Außerdem gibt es sichere Auswahltechniken, die garantieren, dass die Teilmenge von Testfällen, die aus der Anwendung der Technik resultiert, genauso effektiv ist wie die Menge der Ausgangstestfälle. Der Retest-All Ansatz ist eine triviale sichere Technik. DejaVu ist eine andere sichere Technik, deren Anwendung in Abschnitt 4.2.4 an einem ausführlichen Beispiel illustriert wurde.

Der anschließende Vergleich der Auswahltechniken in Abschnitt 4.2.5 liefert keine allgemeingültigen Aussagen, legt aber nahe, dass sichere Auswahltechniken den Datenfluss- und Minimierungstechniken in aller Regel überlegen sind und ihr Einsatz zu Kostenersparnissen führen kann. Die zufällige Auswahl von Testfällen führte zu überraschend guten Ergebnissen. Geringfügig größere Testsuites, die durch zufällige Auswahl zu Stande kommen, sind fast so effektiv wie die sicheren Testsuites.

In den folgenden zwei Abschnitten wurden Studien vorgestellt, die sich mit der Frage nach dem Einfluss der beiden Faktoren ”Intervalllänge zwischen Tests” und ”Priorisierung von Testfällen” auf die Effektivität der verschiedenen Auswahltechniken befasst haben. Anhand der durchgeführten Experimente konnte nachgewiesen werden,

dass beide Faktoren tatsächlich teilweise großen Einfluss haben. Auch hier wollen die Autoren der Studien die Ergebnisse allerdings nicht als allgemeingültig verstanden wissen.

Abschließend wurden die beiden Regressionstest-Rahmenwerke JUnit und TestNG unter Bezug auf die Praxis vorgestellt. Beide Rahmenwerke ermöglichen dem Entwickler eine weitestgehende Automatisierung der Durchführung und Auswertung von Unittests während der Entwicklung. Einen Schritt vorher setzt Parasoft JTest an, das JUnit-kompatible, funktionale Tests für beliebige Java Anwendungen generieren kann (siehe Abschnitt 4.4.3).

Literaturverzeichnis

[Bea04] Pierre Bourque and Robert Dupuis et al. *The Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2004.

[GHK+01] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, 2001.

[HJL+01] M.J. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. Spoon, and A. Gujarathi. Regression test selection for java software. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, November 2001.

[jte] Parasoft jtest url=http://www.parasoft.com.

[jun] Junit url=http://www.junit.org.

[KPR03] Jung-Min Kim, Adam A. Porter, and Gregg Rothermel. An empirical study of regression test application frequency. *Software Testing, Verification and Relieability*, 13(2):65–83, 2003.

[Lig02] Peter Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum, Akademischer Verlag, 2002.

[LW91] H. Leung and L. White. A cost model to compare regression test strategies. In *International Conference on Software Maintenance*, pages 201–208, 1991.

[MK01] D. Marinov and S. Khurshid. Testera: A novel framework for testing java programs, 2001.

[RH97] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.

[RUCH01] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10), 2001.

[Sim05] Anthony J.H. Simons. Testing with guarantees and the failure of regression testing in extreme programming. *Lecture Notes in Computer Science*, 3556:118–126, 2005.

[tes] Testng url=http://testng.org.

Kapitel 5

Redocumentation

Matthias Vianden

Inhaltsverzeichnis

5.1 Einleitung

Bei der Wartung eines Softwaresystems kommt es häufig vor, dass nur noch ein geringer Teil der ursprünglichen Entwickler für die Wartung zur Verfügung stehen. Manchmal wird die Wartung sogar von Entwicklern übernommen, welchen das zu wartende System nicht aus der Entwicklung heraus bekannt ist. Somit können die Entwickler nicht alle Teile der Software kennen und müssen sich im Fehlerfall in die entsprechenden Teile einarbeiten. Dabei ist es wichtig, dass ihnen eine entsprechende Dokumentation zur Verfügung steht, damit sie sich nicht mühsam durch den Quelltext der Anwendung arbeiten müssen. Dabei kommen 19% der Wartungen aufgrund von Dokumentationsfehlern zustande (Platz 2 hinter „Fehler in der Systemlogik“ mit 32 %) [Ein].

Häufig sind die Anwendungen deshalb unzureichend oder falsch dokumentiert, so dass die Entwickler doch wieder in den Quelltext schauen müssen. Um diesen Effekt in der Wartungsphase nicht noch zu verschlimmern, kann die *inkrementelle Redokumentation* als Teil des Wartungsprozesses helfen das Softwaresystem nach und nach während der Wartung zu dokumentieren.

In Kapitel 2 (Dokumentationstechniken) werden die unterschiedlichen Typen von Dokumentationen und Dokumenten vorgestellt und wie diese optimalerweise aussehen sollten. Außerdem werden die Probleme von traditionellen Dokumentationstechniken betrachtet. In Kapitel 3 geht es um den Prozess der Inkrementellen Redokumentation. Hierzu wird ein entsprechender Wartungsprozess vorgestellt. Kapitel 4 zeigt die Erfahrungen mit dem vorgestellten Wartungsprozess anhand einer Fallstudie. In Kapitel 5 werden unterschiedliche Tools zum Dokumentieren von Softwaresystemen vorgestellt.

5.2 Dokumentationstechniken

Um die inkrementelle Dokumentation einzuordnen, werden in diesem Kapittel die verschiedenen Typen von Dokumentationen vorgestellt und die Probleme mit traditionellen Dokumentationstechniken betrachtet.

5.2.1 „in-the-small“ Dokumentation

Als „in-the-small“ Dokumentation bezeichnet man eine Dokumentation, welche die isolierten Teile der Software also zum Beispiel einzelne Module, einzelne Prozeduren oder Funktionen beschreibt [Til93].

Üblicherweise wird hierbei eine Standard-Kopfzeile über jede Prozedur oder Funktion gesetzt, welche kurz die Aufgaben dieser Funktion beschreibt. Außerdem werden häufig die Ein- und Ausgabeparameter aufgelistet und deren Funktion kurz beschrieben. Weiterhin werden auch die einzelnen Datenstrukturen und das Modul selber kurz

erläutert (Siehe Abbildung 5.1).

Eine solche Dokumentation ist durchaus hilfreich, um lokale Zusammenhänge besser zu verstehen. Sie hilft aber nicht, um in großen Softwaresystemen mit vielen Entwicklern die globalen Zusammenhänge zwischen den einzelnen Modulen zu verstehen. Da jeder Entwickler häufig nur seinen Teil der Software versteht und sich in andere Bereiche erst einarbeiten muss.

Dieses ist natürlich im Wartungsfall ein noch größeres Problem, da hier (wenn überhaupt) nur ein Teil der Entwickler des ursprünglichen Systems zur Verfügung stehen. Somit müssen die für die Wartung zuständigen Entwickler sich schnell in große bestehende Systeme einarbeiten müssen und dort die globalen Zusammenhänge verstehen, um dann im Fehlerfall schnell die fehlerhafte Stelle im System zu finden. Wenn nun die Wartungsintervalle länger werden und die für die Wartung zuständigen Entwickler in anderen Projekten eingebunden werden, müssen sie sich häufig im Wartungsfall wieder neu in das System einarbeiten. Dies führt dazu, dass Fehler nur langsamer behoben werden können.

5.2.2 „in-the-large“ Dokumentation

Bevor man nun die lokalen Zusammenhänge und die Details versteht muss man erst die globale Struktur eines Softwaresystems verstehen. Dieses wird insbesondere deutlich, wenn man sich überlegt, dass in einem großen System schnell mehrere tausend Module vorhanden sein können. Um diese große Anzahl Module einigermaßen zu strukturieren, muss man sie in Subsysteme aufteilen und das ganze visualisieren. Dies kann zum Beispiel durch ein Blockdiagramm (UML-Klassendiagramme) geschehen.

Damit diese Dokumentation immer aktuell ist, sollte sie automatisch aus dem Quelltext des Softwaresystems generiert werden. Hierdurch vermeidet man das sonst übliche Problem, dass die Dokumentation „out-of-date“ ist. Sie zeigt also immer die aktuellen Interaktionen im System.

Ein Problem ist nun, die vielen Module eines großen Systems logisch zu ordnen. Dieses kann halbautomatisch geschehen, indem ein Entwickler der sich mit dem System auskennt die verschiedenen automatisch aus dem Quelltext ausgelesenen Module zu logischen Subsystem zusammensetzt [WTMS95]. Hierdurch gibt der Entwickler sein Wissen über die Zusammenhänge des Systems weiter und archiviert sie. Somit entstehen verschiedene Schichten von Subsystemen und Modulen, so dass für einen schnellen Überblick nur die Subsysteme angeschaut werden müssen. Dadurch ist es auch weiterhin möglich in die Tiefe (also die Module) hineinzuschauen und dort zum Beispiel die konkreten Aufgaben einer einzelnen Methode zu betrachten.

Zusätzlich zu den erwähnten Blockdiagrammen sollten auch noch Beschreibungen zu den Subsystemen und Modulen in textueller Form vorliegen, so dass, wenn der Name des Moduls nicht aussagekräftig genug ist, hierzu eine kleine Erläuterung abgelegt werden kann. Außerdem kann so zum Beispiel auch ein kleines „Kochbuch“ für die

Nutzung des Subsystems / Moduls hinterlegt werden, so dass Entwickler den Funktionsumfang des entsprechenden Systems schneller nutzen können.

5.2.3 Probleme mit „traditionellen“ Techniken

Traditionelle Dokumentationen sind meistens:

- nur „in-the-small“
- nicht aktuell
- nur eine Sicht auf die Interaktionen des Systems

Das führt dazu, dass im Wartungsfall die Stellen an denen Änderungen vorgenommen werden müssen nur schwer gefunden werden können. Eine gute Dokumentation sollte es dem Entwickler ermöglichen einen schnellen Überblick über die Module und deren Interaktionen zu erlangen. Außerdem sollte sie die Möglichkeit bieten, auch einen feingranularen Blick auf das Softwaresystem werfen zu können.

5.2.3.1 Dokumentation ist „out-of-date“

Häufig werden zur Designphase im Entwicklungsprozess Dokumente angelegt, welche einen globalen Überblick über die Funktionalität aller Module und deren Interaktion beinhalten. Diese werden dann genutzt, um das System zu implementieren. Häufig kommt es aber dazu, dass bestehende Module abgeändert werden müssen. Also zum Beispiel einige Funktionalitäten hinzugefügt werden müssen. Meistens werden jedoch nun die bestehenden Dokumente nicht aktualisiert.

Im Wartungsfall verschlimmert sich diese Situation sogar, da hier auch wieder Teile geändert werden und neue Funktionalitäten (zum Beispiel im Rahmen von *Change Requests*) hinzugefügt werden. Insbesondere im Fall, dass ein betriebsverhindernder Fehler Ursache der Wartung ist, herrscht großer Zeitdruck. Dadurch werden die Dokumente erst recht nicht mehr aktualisiert. Das wiederum führt dazu, dass solche Dokumente im Wartungsfall, wo sie wichtig wären um schnell die Stelle zu lokalisieren in der Änderungen nötig sind, nicht gelesen werden oder die falschen Schlussfolgerungen aus ihnen gezogen werden. Die Dokumente vermitteln also einen falschen Eindruck des Systems.

5.2.3.2 Dokumente stellen nur eine Sicht auf das System zur Verfügung

Ein weiteres Problem mit textlichen Dokumenten ist, dass diese lediglich eine Sicht auf das System zur Verfügung stellen. Nämlich die des Entwicklers oder Systemdesigners, welcher sie ursprünglich geschrieben hat. Im Wartungsfall braucht man aber

auch andere Sichten auf das System um schnell die Stellen an denen Änderungen nötig sind zu lokalisieren. Verschiedene Sichten stellen dem mit dem System vertrauten Entwickler eine andere (detailliertere) Sicht auf das System zur Verfügung, als zum Beispiel dem Projektleiter welcher sich eher für die globalen Zusammenhänge im System interessiert.

5.2.4 Dokumente

Die Dokumentation eines Softwaresystems besteht natürlich aus einer Reihe von Dokumenten. Um die Navigierbarkeit durch die Dokumentation einfacher zu gestalten, sollten sie als HTML Seiten vorliegen. Dadurch ist es leicht möglich durch die Dokumentation zu „surfen“ (also schnell von einer Stelle zu einer anderen zu springen). Außerdem können hierdurch die unterschiedlichen Typen von Dokumenten gut untereinander verlinkt werden, so dass schnell zwischen den Ansichten gewechselt werden kann.

Nach [RMF92] sollte die Dokumentation aus den folgenden Dokumenten bestehen:

- Informationen über die Domäne
  (allgemeine Informationen über das „Umfeld“ der Anwendung.)
- Management-Dokumentation
  (Enthält Informationen über Änderungen und kann somit zeigen, welche Bereiche Probleme machen.)
- Übersicht
  (Baumstruktur der einzelnen Teile des Systems.)
- Technische Dokumentation
  (Dokumentation des Verhaltens eines konkreten Moduls.)
- Vorhandene Dokumentation
  (Zum Beispiel Anforderungen, Designspezifikationen, Workflows und so weiter.)

5.2.4.1 Informationen über die Domäne

In diesen Dokumenten sollten die allgemeinen Informationen über die Domäne, in der sich die Software bewegt, abgelegt werden. Hier sollten zum Beispiel die wichtigsten Workflows beschrieben sein und zentrale Begriffe geklärt werden.

Diese Dokumente erfüllen somit zwei Hauptaufgaben:

1. Sie erleichtert den Einstieg für neue Mitarbeiter, da diese erst einmal die globalen Zusammenhänge und Motivationen der Software verstehen müssen, ehe sie sich mit ihren eigentlich Aufgaben beschäftigen können.

2. Dienen als Begriffsreferenz, so dass zentrale Begriffe nicht missverstanden (und somit falsch benutzt) werden können.

Eine solches Dokument kann natürlich nicht aus dem Quelltext der Anwendung extrahiert werden. Es sei denn man benutzt ein spezielles Markup (zum Beispiel in der Projektdatei) um eine solche Dokumentation zusammen mit dem Quelltext abzulegen. In der Regel wird es aber so sein, dass diese Dokumente einmal zum Projektstart verfasst werden und dann irgendwo für alle Mitarbeiter zugreifbar abgelegt werden. Dieses ist bei solchen allgemeinen Dokumenten auch nicht tragisch, da sich die Prozesse nicht so schnell ändern werden beziehungsweise die Begriffe immer dasselbe bedeuten werden.

5.2.4.2 Management-Dokumentation

Die Managementdokumente sollen dem Softwaremanager (oder Projektleiter) die Teile der Software zeigen, an denen häufig Änderungen vorgenommen werden. Somit kann er schnell sehen, welche Teile des Softwaresystems häufig Probleme bereiten.

Eine solche Dokumentation kann zum Beispiel aus einer Reihe von LOG-Files bestehen, in denen jede Änderung und deren Details vermerkt werden. Am besten sollte es möglich sein, mit Hilfe dieser Dateien und den Blockdiagrammen der Anwendung einen grafischen Überblick über die Änderungen zu bekommen. Zum Beispiel können die einzelnen Symbole im Blockdiagramm je nach Häufigkeit der Änderungen unterschiedlich eingefärbt werden. Somit ist ein schneller Überblick gewährleistet und der Softwaremanager muss nicht erst umständlich viele Zeilen LOG-Datei durchlesen.

5.2.4.3 Übersicht

Die Übersichtsdokumente sind zum Beispiel die verschiedenen Sichten auf die Subsysteme des Softwaresystems oder komplette UML-Klassendiagramme. Sie dienen dazu gezielt einzelne Stellen im Softwaresystem zu finden, oder einen Überblick über die Interaktion von Modulen oder Subsystemen zu erlangen. Übersichtsdokumente sind deshalb in der Regel Blockdiagramme. Sie können aber auch textuell sein, zum Beispiel in Form der Klassenhierarchie in der Baumansicht von *JavaDoc*. Allerdings sind in der Regel grafische Ansichten schneller zu verstehen, da hier ein schnellerer Überblick über die Subsysteme / Module / Klassen möglich ist.

5.2.4.4 System / Technische Dokumentation

Die Funktionalitäten eines einzelnen Moduls oder einer einzelnen Klasse werden in der Systemdokumentation beschrieben. Diese ist meistens (wie auch die Übersicht) aus dem Softwaresystem selbst erstellt, indem spezielle Kommentarheader über den

einzelnen Methoden ausgelesen werden. In dieser Dokumentation kann der Entwickler ganz genau sehen wozu einzelne Methoden einer Klasse genutzt werden.

Zu dieser Dokumentation gehören auch die automatischen Testfälle, da sie die Interaktionen der einzelnen Klassen zeigen. Die Testfälle selbst müssen natürlich auch dokumentiert sein um sie sinnvoll einordnen und interpretieren zu können.

5.2.4.5 Bestehende Dokumentation

Zu den bestehenden Dokumenten gehören zum Beispiel Designspezifikationen oder auch die einzelnen Change Requests. Die bestehenden Dokumente sollten natürlich mit den anderen Typen von Dokumenten verlinkt werden. Somit ist es dann zum Beispiel möglich bei einem Subsystem auf die entsprechende Designspezifikation zu verweisen, so dass bei Fragen dieses schnell wieder erreichbar ist. Außerdem kann man zu den einzelnen „Change Requests“ die geänderten Module / Klassen und Subsysteme verlinken, so dass hierdurch auch automatisch die Management-Dokumentation erstellt wird.

5.3 Inkrementelle Redokumentation

Als *Redokumentation* bezeichnet man den Prozess ein bestehendes Softwaresystem nachträglich zu dokumentieren.
Häufig kommt es vor, dass insbesondere Altsysteme in weiten Teilen undokumentiert sind. Im Wartungsfall muss der zuständige Entwickler mühselig alle Quellen durcharbeiten um die Stellen, an denen er Änderungen vornehmen muss ausfindig zu machen und das System zu verstehen. Das führt wiederum dazu, dass Änderungen und Fehlerbehebung nicht schnell durchgeführt werden können. Die Redokumentation kann an einem Stück oder inkrementell durchgeführt werden.

Wenn man die Redokumentation an einem Stück vollzieht, dann kann es gegebenenfalls dazu führen, dass Teile des Softwaresystems dokumentiert werden, die nie wieder „angefasst“ werden müssen. Außerdem ist das ganze natürlich mit erheblichen Kosten verbunden. Es kann außerdem dazu führen, dass Änderungen wieder nicht in der Dokumentation nachgezogen werden, da die Dokumentation nicht fester Bestandteil des Wartungszykluses ist.

Die Redokumentation sollte also ein Schritt im Wartungszyklus sein. Dadurch kann sie auch inkrementell vollzogen werden, indem immer nur die Teile der Software dokumentiert werden, von denen der Entwickler gerade die Bedeutung verstanden hat. Hierdurch wird auch verhindert, dass Teile des Systems dokumentiert werden, die nie mehr „angefasst“ werden müssen.

Die so entstehende Dokumentation hilft nun im weiteren Wartungsprozess die Stellen im Softwaresystem an denen Änderungen nötig sind schneller zu finden. Sie dient

nicht dazu, zum Beispiel ein Handbuch für die Software zu erstellen.

> „Redokumentation heilt lediglich das Verständnis der Software und Speichert dieses, somit macht es das Verstehen der Software zu einem späteren Zeitpunkt einfacher.“ [Raj97]

5.3.1 Vorgehensweise

Die Redokumentation wird fest in den Softwarewartungsprozess integriert. Dieser kann somit folgendermaßen aussehen:

Initialisierung

1. Änderungsanfrage
   („change request“ oder Fehlermeldung)

Suche im System

2. Bestehendes System verstehen
   (um die Änderungsanfrage zu verstehen)
3. Änderungen lokalisieren
   (Stellen an denen Änderungen vorgenommen werden müssen identifizieren)

Implementierung und Prüfung

4. Redesign
   (Design für diese Änderung anpassen)
5. Änderung implementieren
   (Bestehendes Softwaresystem ändern)
6. Nachwirkungen der Änderungen
   (gegebenenfalls weitere Stellen im System anpassen)
7. Prüfung
   (alle Änderungen korrekt und vollständig?)

Redokumentation

8. *Redokumentation*
   (Über das System gewonnenes Wissen archivieren)

Auslieferung

9. Ggf. „Configuration Management“ und „Release“
   (Auslieferung des geänderten Systems)

(Siehe [Raj97] und [AJR04])

5.3.1.1 Initialisierung

In der Regel werden Fehlermeldungen oder Change Requests über ein Ticketsystem oder über Dokumente an die Entwickler weitergegeben. Diese müssen sie dann gemäß ihrer Wichtigkeit einstufen und entsprechend bearbeiten.

Bei betriebsverhindernden Fehlern zum Beispiel wird es normalerweise so sein, dass der Entwickler sofort mit der Bearbeitung des Fehlers beginnt, um dem Kunden möglichst schnell ein stabiles System zur Verfügung zu stellen.

5.3.1.2 Suche im System

Wenn der Entwickler mit der Bearbeitung begonnen hat, muss er natürlich erst einmal die Stellen im Softwaresystem finden, die er ändern muss. Hierzu muss er das bestehende System verstehen, wobei es je nach Dokumentationsstand des Systems mehrere Möglichkeiten gibt:

1. Den Quelltext der Anwendung so lange durcharbeiten und allen Aufrufen folgen, bis das System verstanden ist.

2. Wenn die entsprechenden Teile schon dokumentiert sind und es die entsprechenden Stufen in der Dokumentation gibt (also verschieden detaillierte Sichten), dann kann er sich von „oben“ nach „unten“ durch die Dokumentation hangeln. Dieses wird deutlich schneller gehen, als das durcharbeiten des Quelltextes, da der Entwickler nicht mehr den kompletten Quelltext einer Methode lesen muss, sondern sich nur die Beschreibungen durchlesen muss. Außerdem muss er in der Regel nicht mehr so tief in das System eindringen, da ihm häufig die Beschreibung von benachbarten Modulen ausreichen wird, um deren Bedeutung zu verstehen. Somit muss er sich den Quelltext der entsprechenden Module nicht genauer anschauen. Durch die Verlinkung der einzelnen Seiten hat der Entwickler zusätzlich die Möglichkeit schnell zwischen den einzelnen Teilen hin und her zu springen.

Im Fall, dass ein betriebsverhindernder Fehler die Ursache der Wartung ist, kann es außerdem vorkommen, dass diese Phase sehr kurz gehalten wird und die Änderungen „quick-and-dirty“ in das System eingebaut werden um den konkreten Fehler zu beheben. Diese Änderungen sollten dann aber, wenn mehr Zeit zu Verfügung steht, zurückgenommen werden und der Fehler erneut mit dem gegebenenfalls notwendigen Redesign behoben werden.

5.3.1.3 Implementierung und Prüfung

Den Anfang der Implementierungsphase bildet das Redesign. Hier sollte das Design des bestehenden Systems so geändert werden, dass es die neuen Aufgaben erledigt beziehungsweise die vorhandenen Fehler behebt. Bei drastischen Änderungen im Design oder bei der Entwicklung neuer Teile sollte eine entsprechende Designspezifikation verfasst werden und diese dem Dokumentensystem zugeführt werden. Somit kann man bei der Bearbeitung dann auf dieses Dokument verlinken und kann dort noch einmal die initialen Ideen nachschauen.

Anschließend sollten die entsprechenden Änderungen implementiert werden. Hierbei muss dann entweder die fehlerhafte Stelle korrigiert werden oder das neue Feature implementiert werden. Diese Änderungen sollten in einem entsprechenden Versionskontrollsystem mit sinnvollen Kommentaren versehen, abgespeichert werden. Hierdurch kann der Manager (Projektleiter) sich die Entwicklung der Software anschauen und entsprechend schnell Subsysteme finden, welche häufig Fehler verursachen. Somit kann er dann zum Beispiel dafür sorgen, dass Zeit zur Verfügung steht um das anfällige Subsystem neu zu designen.

Im Anschluss an die Änderungen welche konkret diesen „change request“ beziehungsweise Fehler betreffen, müssen gegebenenfalls weitere Teile des Softwaresystems geändert werden, um wieder mit den gerade geänderten Teilen funktionieren zu können. Hierzu ist es wichtig, dass zum Beispiel durch einen „call-graph“ die Teile im System gezeigt werden, welche durch die gerade gemachten Änderungen betroffen sind. So kann er schnell prüfen ob an diesen Stellen auch Anpassungen nötig sind.
Alternativ können die entsprechenden Stellen natürlich auch durch Textsuche in den Quellen gefunden werden. Dies wird allerdings problematisch, wenn es sich zum Beispiel um einen Methodennamen handelt, welcher in diversen Klassen vorkommt. Da nun auch wieder ein großes Ergebnis durchstöbert werden muss.

Zum Schluss muss geprüft werden, ob die gemachten Änderungen korrekt und vollständig sind.

vollständig: Die Änderung ist vollständig, wenn im Fehlerfall der beschriebene Fehler (und alle seine Abwandlungen) behoben ist beziehungsweise die im „change request“ geforderten Änderungen gemacht wurden.

korrekt: Die Änderung ist korrekt, wenn sie außer dem beschriebenen Fehler beziehungsweise den geforderten Änderungen zu keinen weiteren Änderungen im System geführt hat.

Um dieses zu prüfen sollten automatisierte Tests oder mit dem System vertraute Softwaretester die einzelnen Funktionalitäten der Software überprüfen. Sollten die Tests vollständig sein, ist sichergestellt, dass die Stellen im System die bisher fehlerfrei waren auch nach der Änderung noch fehlerfrei sind. Für die neuen Funktionalitäten sollten neue Test geschrieben werden beziehungsweise der beschriebene Fehler ebenfalls

getestet werden. Somit wird verhindert, dass durch eine erneute Änderung am System dieser Fehler erneut auftritt.

5.3.1.4 Redokumentation

In diesem Schritt werden nun die im Schritt 2 gesammelten Erkenntnisse über das System festgehalten. Es gibt nun zwei mögliche Szenarien:

1. Das Subsystem ist nicht dokumentiert.
   In diesem Fall werden die entsprechenden Methoden, Klassen und Module kommentiert und entsprechend ihres logischen Zusammenspiels zu Subsystemen zusammengefasst. Die Dokumentation wird also für diesen Teil (neu) erstellt.
2. Das Subsystem ist bereits dokumentiert.
   In diesem Fall wird die bestehende Dokumentation auf die gemachten Änderungen angepasst. Außerdem sollte die Dokumentation geändert werden, wenn im Schritt 2 aufgefallen ist, dass es zum Beispiel sprachliche Defizite in einigen Kommentaren gab oder gewisse Funktionalitäten nicht richtig beschrieben wurden.

Diese Teile können fast ausnahmslos im Quelltext der Anwendung geschehen, da die entsprechenden Dokumente aus diesem generiert werden. Nur das zusammenfassen zu Subsystemen muss gegebenenfalls in einer entsprechenden Anwendung geschehen wenn es nicht im Quelltext zum Beispiel durch spezielle *Tags* möglich ist.

5.3.1.5 Auslieferung

Das Softwaresystem wird in der Regel nicht nach jedem „change request“, sondern zu festen Termine ausgeliefert. Bei einem betriebsverhindernden Fehler wird das System natürlich sofort nach der Fehlerbehebung ausgeliefert. Das Softwaresystem wird meistens durch einen entsprechenden Manager oder Projektleiter ausgeliefert. Die Auslieferung selbst besteht im Wesentlichen daraus, dass der aktuelle Stand im Versionskontrollsystem „ausgecheckt“, erzeugt, gegebenenfalls noch einmal getestet und schließlich an die Kunden ausgeliefert wird. Außerdem müssen noch entsprechende „release notes“ geschrieben werden, wenn sie nicht automatisch erzeugt werden. In den „release notes“ werden zum Beispiel die Änderungen seit der letzten Auslieferung aufgeschrieben.

5.4 Erfahrungen mit Redokumentation

Anhand der nachfolgenden Studie sollen die Vorteile der inkrementellen Redokumentation erläutert werden.

5.4.1 Ergebnisse der Studie: [AJR04]

In dieser Studie wurde Redokumentation während der Wartung des Softwaresystems **Commsrvc** eingesetzt. Dieses ist ein verteiltes System, welches dazu dient, Schecks zu verarbeiten. Dazu liest es den Scheck per Scanner ein, sendet seine Daten zu einem zentralen Rechner, welcher die Daten dann an die Bank weiterleitet, die den Scheck ursprünglich ausgestellt hat. Diese kann nun die Deckung des Schecks überprüfen und dann gegebenenfalls das Geld freigeben. **Commsrvc** wurde in *Pascal* geschrieben und ist in mehrere hierarchische Ebenen aufgeteilt. Diese sind zum Beispiel eine Kommunikationsebene oder eine Ebene zum Auslesen der Schecks. Die Studie wurde über 4 Jahre (von 1997 bis 2000) durchgeführt.

Commsrvc entwickelte sich in diesem Zeitraum folgendermaßen:

Initial (Release 1)

- Anzahl Funktionen: 147
- Anzahl Codezeilen: 5739
- Anzahl Kommentarzeilen: 998 (ca. 17,39% der Codezeilen)

1997

- Anzahl Funktionen: 248
- Anzahl Codezeilen: 10177
- Anzahl Kommentarzeilen: 1199 (ca. 11,78% der Codezeilen)

2000

- Anzahl Funktionen: 289
- Anzahl Codezeilen: 11932
- Anzahl Kommentarzeilen: 1470 (ca. 12,31% der Codezeilen)

In diesem Zeitraum hat es 52 Fehlermeldungen und Change Requests gegeben, welche sich zum Teil in mehrere Teile gegliedert haben. Die Wartung wurde nach dem oben beschriebenen Wartungsprozess durchgeführt. Wenn man sich nun anschaut, wie viel Zeit die Entwickler pro Änderung einer Komponente gebraucht haben, sieht man, dass diese drastisch nach unten geht je mehr Code redokumentiert wurde.

1997	25,95 % Redokumentiert	315 Minuten pro Änderung einer Komponente
1998	68,51 % Redokumentiert	142,6 Minuten pro Änderung einer Komponente
1999	76,12 % Redokumentiert	121,4 Minuten pro Änderung einer Komponente
2000	83,39 % Redokumentiert	103,2 Minuten pro Änderung einer Komponente

Es ist zusätzlich interessant zu sehen, wie lange die Entwickler für das Schreiben der entsprechenden Kommentare benötigt haben, da dies ein zusätzlicher Aufwand ist, der selbstverständlich auch Geld kostet. Die Zeiten für die Redokumentation pro Komponente ist dabei von 18,1 Minuten (1997) auf 5,3 Minuten (2000) gefallen. Das würde man auch intuitiv erwarten, da ja weniger Quelltext redokumentiert werden muss (da schon ein großer Teil (Im Jahr 2000: 83,39 %) des Systems redokumentiert ist).

Auf alle Jahre hochgerecht hat somit die Redokumentation nur einen sehr kleinen Anteil der eigentlichen Arbeit (5,48 %) gekostet. Geht man nun davon aus, dass ohne Redokumentation jede Änderung weiterhin ca. 315 Min. gedauert hätte, dann wurde durch die Redokumentation nach 52 Änderungen ca. 50 % der Zeit gespart [AJR04].

Wie oben gesehen hat, die Redokumentation schon nach einem Jahr einen sehr großen Effekt auf den Wartungsprozess, da der Aufwand pro Änderung einer Komponente um ca. 55 % (von 315 Minuten auf 142,6 Minuten) fällt. Wenn man nun auch noch die zusätzliche Zeit für die Redokumentation mit hinzunimmt, zahlt sich die Redokumentation nach ca. 1,5 Jahren aus.

Allerdings kann davon ausgegangen werden, dass sich auch ohne Redokumentation die Zeit pro Änderung verringert hätte, da die Entwickler nach 4 Jahren Wartung das System besser kennen als noch am Anfang des Wartungszeitraums. Außerdem ist es am Anfang wahrscheinlicher, dass große Änderungen im System notwendig sind, welche selbstverständlich mehr Zeit benötigen als kleine Änderungen.

Ein wichtiger Grund dafür, dass sich die Redokumentation in diesem Projekt so ausgezahlt hat, liegt sehr wahrscheinlich daran, dass **Commsrvc** so gut wie keine Dokumentation auf Komponentenbasis besessen hat, da es ursprünglich nur von einem Entwickler programmiert wurde, welcher einen guten Überblick über die einzelnen Komponenten hatte.

5.4.2 Vorteile der inkrementellen Redokumentation

Die Vorteile der Redokumentation von großen bestehenden Softwaresystemen können folgendermaßen zusammengefasst werden:

- Das Verständnis vom Anfang wird gespeichert, so dass es zu einem späteren Zeitpunkt wieder abrufbar ist.

- Je mehr Änderungen es gibt, desto mehr Verständnis wird gespeichert und desto mehr Dokumentation gibt es

- Wenn die Dokumentation durch Kommentare an den entsprechenden Stellen automatisch generiert wird, können Beschreibungen von anderen Programmierern leichter geändert werden, um Zusammenhänge genauer zu beschreiben, die Lesbarkeit zu erhöhen oder den Stil zu verbessern.

- Es werden keine voreiligen Kosten in die Dokumentation von Teilen der Software investiert, welche bei einer möglichen Wartung nicht relevant sind.

5.5 Tools

Wie in „Dokumentationstechniken“ gezeigt, sollte die Dokumentation automatisch aus dem Quellcode generiert werden. Hierzu gibt es eine Reihe von kostenlosen und kostenpflichtigen Tools. Auszugsweise werden hier *JavaDoc*, *Doxygen* und *Doc-o-matic* vorgestellt.

Außer den drei vorgestellten Tools gibt es noch eine Fülle weiterer Anwendungen, die meistens auf die Sprachen C++ und C spezialisiert sind. Im Rahmen dieser Ausarbeitung konnte nicht auf alle eingegangen werden, aber die Benutzung der einzelnen Werkzeuge ist meistens ähnlich.

5.5.1 JavaDoc

Javadoc ist das Standardtool, wenn es um die Dokumentation von Java Programmen geht [javd]. Hierbei wird der Quelltext der Anwendung ausgelesen und aus diesem HTML Seiten generiert. Hierbei werden spezielle Kommentare über Methoden, Klassen, Interfaces und so weiter ausgelesen und deren Inhalt mit in der Dokumentation ausgegeben. Die Javadoc Kommentare werden hierei mit `/**` eingeleitet und mit `*/` beendet. Außerdem können spezielle Tags verwendet werden (zum Beispiel `@see`, um Verweise zu anderen Klassen zu markieren oder `@param`, um Parameter von Methoden zu beschreiben).

Um zum Beispiel eigene Tags auszulesen beziehungsweise die Ausgabe zu erweitern, kann *JavaDoc* über *Doclets* erweitert werden. Diese beschreiben den Inhalt und die Ausgabe dessen, was dokumentiert werden soll. Nutzt man kein anderes *Doclet*, dann benutzt *JavaDoc* das von SUN zur Verfügung gestellte „standard“-Doclet. Mit diesem generiert *JavaDoc* dann die typische API-Documentation in HTML Form [javb].
Doclets können zum Beispiel dazu genutzt werden komplette *JUnit-Test* zu entwerfen (siehe [jave]). Man kann mit ihnen (zum Beispiel mit [java]) aber auch komplette Programme „reverse engineeren“ ohne den Quelltext zu besitzen.
Eine ausführliche Liste vieler weiterer *Doclets* befindet sich auf der Seite [javc].

5.5.2 Doxygen

Im Gegensatz zu *Javadoc* kann *Doxygen* auch dazu genutzt werden C++, C, Objective-C, Python, PHP und C# Programme zu dokumentieren (Pascal und Delphi über Precompiler wie zum Beispiel [pas]). Außerdem kann *Doxygen* nicht nur HTML-, sondern auch PDF-, Tex- oder PS- Dokumente erstellen (siehe [doxa]).

Pro Methode, Klasse oder Modul kann ein kurzer und ein ausführlicher Kommentar angegeben werden. Wie bei Javadoc kann der kurze Kommentar auch automatisch ausgelesen werden (der erste Satz im Kommentar). Kommentare können entweder wie bei *JavaDoc* mit `/**` eingeleitet und normal über `*/` beendet werden. Es kann aber auch der Qt-standard `/*!` zur Einleitung genutzt werden. Alternativ können einzeilige Kommentare durch `///` eingeleitet werden. Tags werden in *Doxygen* durch ein `\` eingeleitet (zum Beispiel `\param`). Abbildung 5.3 zeigt das Beispiel aus dem *Doxygen* Manual, Sektion „Documenting the code“. Viele generierte Dokumentationen sind auf [doxb] verlinkt.

Doxygen bietet außerdem die Möglichkeit Klassen, Variablen und Funktionen in Gruppen zusammenzufassen (Module). So können alle globalen Variablen eines C-Programms auf einer einzelnen Seite aufgelistet werden.

Im Gegensatz zu JavaDoc kann man mit *Doxygen* grafische Klassenhierarchien und Kollaborationsgraphen pro Klasse und sogar *Callgraphen* pro Methode generieren lassen. Diese werden dann automatisch in die Dokumente eingebunden.

5.5.3 Doc-o-matic

Im Gegensatz zu den beiden Vorangegangenen ist *Doc-o-matic* ein kostenpflichtiges Werkzeug und muss pro Arbeitsplatz lizenziert werden. Es bietet dafür aber den Vorteil einer kompletten grafischen Benutzeroberfläche.
Doc-o-matic kann sowohl C++, Java, C# als auch Delphi Programme dokumentieren. Hierbei gibt es nicht nur die Möglichkeit HTML Dokumente zu erstellen, sondern es können auch Hilfedateien und PDFs erstellt werden. In Abbildung 5.2 finden sie einen Screenshot einer durch *Doc-o-matic* erstellten HTML-Dokumentation.

Beim Einlesen des Quelltextes bietet *Doc-o-matic* an, bestehende Tags zu übernehmen. Hierbei kann der Benutzer die Bezeichnungen frei eingeben, so dass zum Beispiel die Überschriften `Author:`, `Discription:` und `Return Value:` aus dem Delphi Quelltext von Abbildung 5.1 eingelesen und entsprechend interpretiert werden können.

Doc-o-matic bietet hierbei auch die Möglichkeit Klassen, Methoden, Namespaces und so weiter im Nachhinein (also nach dem Einlesen des Quelltextes) zu dokumentieren. Diese zusätzlichen Kommentare können dann auch wieder in den Quelltext zurück geschrieben werden.

Bei der Ausgabe kann *Doc-o-matic* auch eine grafische Vererbungshirarchie ausgeben

(alternativ zur Textuellen) . Allerdings können keine Callgraphen oder Kollaborationsgraphen erstellt werden.

Eine Demoversion von *Doc-o-matic* kann auf [doc] heruntergeladen werden.

5.5.4 Vergleich

In Java geschriebene Softwaresysteme bieten sich natürlich dafür an, mit JavaDoc dokumentiert zu werden. Allerdings bietet JavaDoc nicht die Möglichkeit, grafische Ausgaben zu generieren oder Klassen und Packages zu Subsystemen zu gruppieren.
Doxygen hingegen bietet die Möglichkeit für grafische Ausgaben und bietet außerdem die Möglichkeit Gruppen (Module) zu bilden.
Durch die grafische Oberfläche ist *Doc-o-matic* sehr einfach zu bedienen. Dabei bietet es sowohl die Möglichkeit Module zu bilden, als auch grafische Klassenhierarchie zu generieren. Allerdings können keine weiteren grafischen Ansichten generiert werden und es kann nicht erweitert werden. Zudem müssen Lizenzen pro Arbeitsplatz gekauft werden, wohingegen die anderen beiden Tools kostenlos sind.

5.6 Zusammenfassung

Wir haben gesehen, dass inkrementelle Redokumentation ein sinnvoller Schritt im Wartungsprozess ist. Es hilft das Verständnis über Komponenten, auch über längere Wartungzyklen zu speichern und spart Geld, weil die Entwickler sich nicht mehr lange durch den Quelltext arbeiten müssen.

Wie in der Studie gesehen, hilft der beschriebene Wartungsprozess die Wartungscyklen zu verkürzen. Allerdings dürfen hierbei auch nicht die anderen beschriebenen Einflussfaktoren übersehen werden. Dennoch ist die inkrementelle Redokumentation ein guter Weg um die Dokumentation eines Softwaresystems zu verbessern. Durch die feste Integration in den beschriebenen Wartungsprozess gewöhnen sich die Entwickler an diesen Teil und die Dokumentation wird ein fester Bestandteil der täglichen Arbeit. Da der Aufwand die inkrementelle Redokumentation durchzuführen mit 5,48 % nicht deutlich höher ist, als der normale bei der Wartung auftretende Aufwand (ca. 5 %) [Ein], ist sie außerdem auch noch kostenschonend.

Um möglichst allen am Softwareentwicklungsprozess beteiligten Mitarbeitern die geeigneten Dokumente an die Hand zu geben sollten die genannten Typen von Dokumenten vorhanden sein. Diese sollten der einfachen Navigierbarkeit halber untereinander verlinkt sein. Um zu verhindern, dass die Dokumente nicht mehr aktuell sind, sollten sie (soweit möglich) aus dem Quellcode der Anwendung und gegebenenfalls unter Zuhilfenahme von Zusatzinformationen generiert werden. So können sich zum Beispiel die Informationen über die Anwendungsdomäne als Hypertext im Projektfile unterbringen lassen oder die Managementdokumentation aus den Übersichsdiagrammen und der Quellcodehistorie generieren lassen.

Literaturverzeichnis

[AJR04] Andrian Marcus Alexander J. Rostkowycz, Vaclav Rajlich. A case study on the long-term effects of software redocumentation. In 20th IEEE International Conference on Software Maintenance *CSMR '04*, pages 92–101, 2004.

[doc] doc-o-matic homepage
`http://www.doc-o-matic.de/`.

[doxa] Doxygen homepage
`http://www.stack.nl/~dimitri/doxygen/`.

[doxb] Doxygen projektbeispiele
`http://www.stack.nl/~dimitri/doxygen/projects.html`.

[Ein] Software engineering (einführung)
`http://ivs.cs.uni-magdeburg.de/~dumke/st1/folien.pdf`.

[java] Classdoc
`http://classdoc.sourceforge.net/`.

[javb] Doclet overview
`http://java.sun.com/j2se/1.4.2/docs/tooldocs/javadoc/overview.html`.

[javc] Doclet.com
`http://http://www.doclet.com/`.

[javd] Javadoc homepage
`http://java.sun.com/j2se/javadoc/`.

[jave] Junitdoclet
`http://www.junitdoclet.org/home/index.html`.

[pas] Pas2dox open source project
`http://pas2dox.org/`.

[Raj97] Vaclav Rajlich. Incremental redocumentation with hypertext. In 1st Euromicro Working Conference on Software Maintenance and Reengineering *CSMR '97*, pages 68–72, 1997.

[RMF92] Malcohn Munro Robert M. Freeman. Redocumentation for the maintenance of software. In Proceedings of the 30th annual Southeast regional conference (Raleigh, North Carolina; 1992), pages 413–416, 1992.

[Til93] Scott R. Tilley. Documenting-in-the-large vs. documenting-in-the-small. In the Proceedings of *CASCON '93*, (Toronto, Ontario; October 25-28, 1993), pages 1083–1090, October 1993.

[WTMS95] Kenny Wong, Scott R. Tilley, Hausi A. Müller, and Margaret-Anne D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54, 1995.

```
1    // Author: Matthias Vianden
2    // Discription:
3    //     Use the class TListIterator to iterate over entrys
4    //     in a TLists.
5    unit uniListIterator;
6
```

[...]

```
10
11   type
12      // Description:
13      //    Use Objects of this Class to iterate over TLists.
14      // Example:
15      //     aIterator := TIterator.create(aList);
16      //     aIterator.first;
17      //     while not aIterator.isDone() do
18      //     begin
19      //       [...]
20      //       aIterator.next();
21      //     end;
22     TListIterator = class(TObject)
23     private
24       myCurrentPosition : Integer; // The current position of the Iterator
25       myAssignedList    : TList;   // The assigned list of the Iterator
26
27       // Throws a Exception if the Iterator is not in its Bounds.
28       procedure checkBounds();
29     public
30       // Creates a Iterator Object to iterator over the List aList.
31       constructor create(aList : TList); reintroduce;
32
33       // returns True if the Iterator ist at its end.
34       function isDone() : Boolean;
35
36       // returns the item at the current position.
37       function getCurrent() : TObject;
38
```

[...]

```
91   // Description:
92   //     Use this methode to get the entry of the given TList at the current
93   //     Position.
94   // Return Value:
95   //     TObject The current Entry of the TList
96   // Throws:
97   //     EIteratorOutOfBounds If the Iterator is not in its Bounds, then this
98   //                          Exception will be thrown.
99   function TListIterator.getCurrent: TObject;
100  begin
101    // first check if the Iterator is in its bounds
102    self.checkBounds();
103
104    // then return the entry (cast to TObject)
105    Result :=  TObject(self.myAssignedList.Items[self.myCurrentPosition]);
106  end;
```

Abbildung 5.1: ListIterator (Delphi)

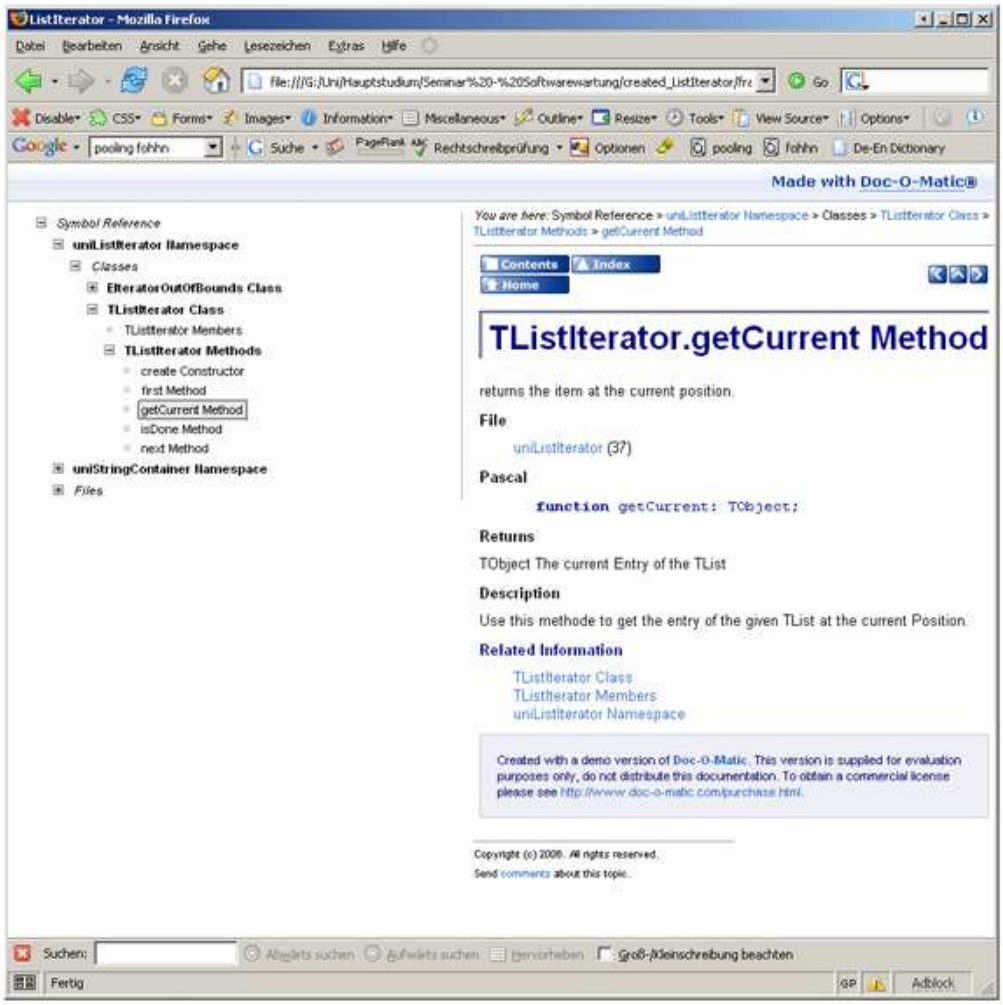


Abbildung 5.2: Generierte Dokumentation (*Doc-o-matic*)

```
1  //!  A test class.
2  /*!
3    A more elaborate class description.
4  */
5
6  class Test
7  {
8    public:
9
10     //! An enum.
11     /*! More detailed enum description. */
12     enum TEnum {
13                  TVal1, /*!< Enum value TVal1. */
14                  TVal2, /*!< Enum value TVal2. */
15                  TVal3  /*!< Enum value TVal3. */
16                }
17          //! Enum pointer.
18          /*! Details. */
19          *enumPtr,
20          //! Enum variable.
21          /*! Details. */
22          enumVar;
23
24     //! A normal member taking two arguments and returning an integer value.
25     /*!
26       \param a an integer argument.
27       \param s a constant character pointer.
28       \return The test results
29       \sa Test(), ~Test(), testMeToo() and publicVar()
30     */
31     int testMe(int a,const char *s);
```

Abbildung 5.3: Test (C++ - Doxygen)

Kapitel 6

Analyse von Versionshistorien

Mark Wiesemann

Inhaltsverzeichnis

6.1 Einleitung

Die Versionshistorie wird bei der Softwareentwicklung verwendet, um die verschiedenen Versionen des Quellcodes zu verwalten. Damit ist es z.B. möglich, zu prüfen, welcher Entwickler was und zu welchem Zeitpunkt geändert hat, alte Versionen wiederherzustellen, mehrere „Zweige“ eines Projekts (z.B. eine stabile und eine Beta-Version) zu verwalten und den Zugriff der Entwickler auf den Quellcode zu kontrollieren.

Die Informationen aus der Versionshistorie lassen sich u.a. nutzen, um Änderungen vorzuschlagen, um das Debugging zu erleichtern oder um problematische und fehleranfällige Module zu identifizieren. Auch zur Abschätzung von Release-Intervallen, von Codewachstum, von Wartungskosten oder der Fehlerzahl ist die Versionshistorie nützlich.

Versions-Kontrollsysteme (Version Control Systems, kurz: VCS) verwalten die Versionshistorie und erlauben es, jederzeit wieder auf ältere Versionen von Dateien zuzugreifen. Mit jeder neuen Version („commit“), die in einem VCS abgelegt wird, werden vom Entwickler auch Informationen über die vorgenommen Änderungen gespeichert. Beispiele für Versions-Kontrollsysteme sind CVS (Concurrent Versions System), das neuere und als CVS-Nachfolger gedachte Subversion oder das vor einem Jahr zur Verwaltung des Linux-Kernels entwickelte git.

Versions-Kontrollsysteme arbeiten meist entweder datei- oder änderungsbasiert. Während im ersten Fall jede Datei einzeln betrachtet wird, wird im zweiten Fall betrachtet, welche Dateien von einer Änderung des Quellcodes betroffen sind. Zum Beispiel arbeitet CVS dateibasiert, Subversion aber änderungsbasiert.

In dieser Arbeit werden verschiedene Ansätze vorgestellt, um mit Hilfe der Versionshistorie Vorschläge für weitere Änderungen zu generieren oder um einen Überblick über die letzten Änderungen einer Methode im Quellcode zu bekommen.

6.2 eROSE – Reengineering of Software Evolution

eROSE (Abkürzung für „Reengineering of Software Evolution“) ist ein am Lehrstuhl für Softwaretechnik der Universität des Saarlandes entwickeltes Plugin für die Software-Entwicklungsumgebung Eclipse. Mit Hilfe von Datamining auf der Versionshistorie sollen den Entwicklern verwandte Änderungen gezeigt werden. Es arbeitet ähnlich dem Vorschlagssystem von Amazon nach dem Prinzip „Programmers who changed these functions also changed . . .“.

6.2.1 Einführung

Abbildung 6.1 zeigt das Vorschlagsprinzip von eROSE am Beispiel des Source-Codes von Eclipse: In Zeile 173 wurde eine neue Konstante für eine zusätzliche Tabellenspalte eingefügt. Das eROSE-Plugin zeigt nun im unteren Bereich des Fensters verwandte Änderungen früherer Transaktionen an (Transaktionen bestehen aus mehreren Änderungen an mehreren Stellen im Code; vgl. Abschnitt 6.2.2). Während der erste Hinweis, dass die `COLUMN_COUNT`-Konstante (in Zeile 171) bei ähnlichen Transaktionen geändert wurde, dem Entwickler nicht weiterhilft, ist bereits der zweite Hinweis wichtig, denn auch das in Zeile 182 beginnende Array `columnHeaders[]` muss um ein weiteres Element erweitert werden, damit die neu definierte Konstante sinnvoll eingesetzt werden kann.

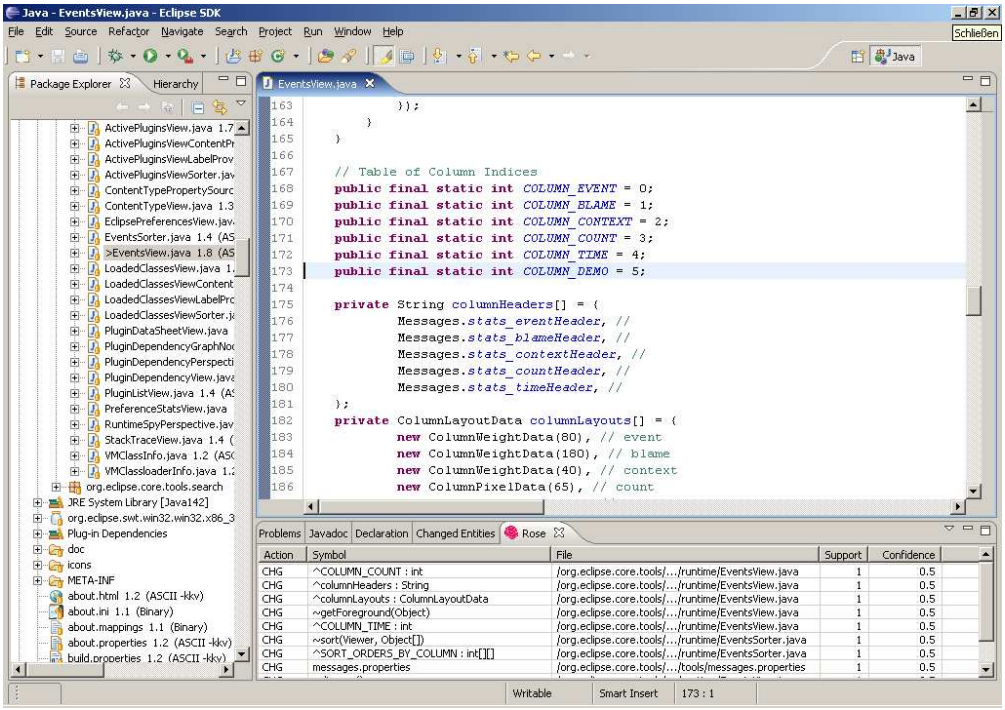


Abbildung 6.1: Das Eclipse-Plugin eROSE zeigt Stellen im Code (hier: Eclipse-Source-Code), an denen bei früheren ähnlichen Transaktionen weitere Änderungen vorgenommen wurden

6.2.2 Transaktionen

Abbildung 6.2 zeigt den Datenfluss in eROSE. Der eROSE-Server liest zunächst das Archiv eines CVS-Servers ein und gruppiert alle Änderungen zu Transaktionen. Mit Hilfe von Datamining bildet der Server aus den Transaktionen Regeln nach dem Prinzip: „Wenn die `COLUMN`-Konstanten geändert werden, dann wird üblicherweise auch das Array `columnHeaders[]` geändert.“ (vgl. Abschnitt 6.2.1) Wenn ein Entwickler einen Teil des Codes (Konstante, Variable, Methode etc.) ändert, sucht der eROSE-

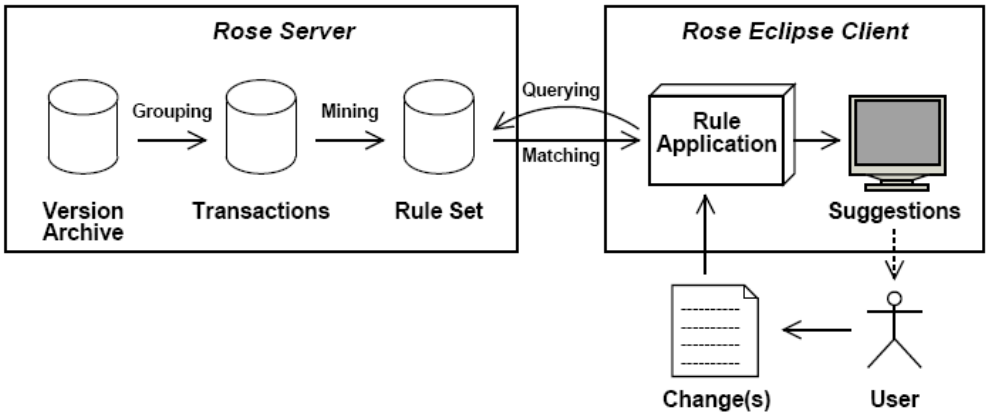


Abbildung 6.2: Datenfluss in eROSE

Client in den Regeln („Rule Set“) nach einer anwendbaren Regel und macht passende Vorschläge (z.B. die Änderung von `columnHeaders[]`).

Formal ist eine ÄNDERUNG eine Abbildung $\delta : \mathcal{P} \to \mathcal{P}$, die ein PRODUKT $p \in \mathcal{P}$ in ein GEÄNDERTES PRODUKT $p' = \delta(p) \in \mathcal{P}$ überführt. $\mathcal{P}$ ist die Menge aller Produkte und $\mathcal{C} = \mathcal{P} \to \mathcal{P}$ ist die Menge aller Änderungen. Ein Produkt ist z.B. eine Datei, eine Klasse oder eine Methode.

Mehrere Änderungen werden als Komposition zusammengefasst ($\circ : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$). Eine TRANSAKTION besteht aus mehreren Änderungen an mehreren Stellen im Code. Zum Beispiel wird die Transaktion $\Delta_{1,2}$ zwischen zwei Versionen $p_1, p_2 \in \mathcal{P}$ bestehend aus n individuellen Änderungen $\delta_1, \ldots, \delta_n$ geschrieben als $\Delta_{1,2} = \delta_1 \circ \delta_2 \circ \cdots \circ \delta_n$ mit $\Delta_{1,2}(p_1) = (\delta_1 \circ \delta_2 \circ \cdots \circ \delta_n)(p_1) = \delta_1(\delta_2(\cdots \delta_n(p_1))) = p_2$.

ENTITÄTEN werden benutzt, um alle syntaktischen Komponenten des Codes zusammenzufassen. Eine Entität ist ein Tripel (f, c, i). Dabei ist f der Name der betroffenen Datei, c ist die syntaktische Komponentenart (Konstante, Variable, Methode, Klasse, Datei etc.) und i ist der Bezeichner der Komponente.

Die Abbildung $entities$ liefert alle Entitäten, die von einer Änderung oder einer Transaktion betroffen sind. Am Beispiel von Abschnitt 6.2.1 ergibt sich:

$$entities(\Delta) = entities(\delta_1) \cup \cdots \cup entities(\delta_n) = \left\{ \begin{array}{lll} (\text{EventsView.java}, & class, & \texttt{EventsView}), \\ (\text{EventsView.java}, & constant, & \texttt{COLUMN\_COUNT}), \\ (\text{EventsView.java}, & field, & \texttt{columnHeaders[]}), \\ (\text{EventsView.java}, & method, & \texttt{getForeground()}), \\ \ldots & & \end{array} \right\}$$

Der eROSE-Server versucht, einige Probleme von CVS zu umgehen. Während andere Versions-Kontrollsysteme die Versionierung änderungsbasiert vornehmen, geht CVS dateibasiert vor. Um eine änderungsbasierte Versionierung zu bekommen, gruppiert eROSE einzelne Dateiänderungen zu Transaktionen. Dabei werden zwei aufeinander-

folgende Änderungen δ_i und δ_{i+1} von dem gleichen Entwickler und mit der gleichen Anmerkung zu einer Transaktion Δ zusammengefasst, wenn der Abstand zwischen den Änderungen maximal 200 Sekunden beträgt. Zusammenfassungen („merge“) von Zweigen („branch“) werden genauso als Transaktionen erkannt. Dies ist aber nicht erwünscht, da die Änderungen in solchen großen Transaktionen meist keinen logischen Zusammenhang haben. eROSE ignoriert daher alle Änderungen, die mehr als 30 Entitäten umfassen.

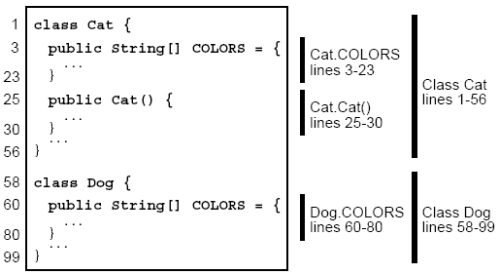


Abbildung 6.3: eROSE fasst Code zu Entitäten zusammen (aus [ZWDZ04a])

CVS kann nur Dateien und Zeilennummern für Änderungen verwalten, nicht aber Details über die Syntax. Daher parst eROSE die Dateien und ordnet Entitäten Zeilennummern (von / bis) zu. Wie in Abbildung 6.3 zu sehen ist, kann eROSE jede Änderung (vom CVS-Server nur durch Dateinamen und Zeilennummern identifiziert) den Entitäten zuordnen.

6.2.3 Regeln

eROSE benutzt Datamining, um aus Transaktionen Regeln zu bilden. Ein Beispiel für eine Regel ist:

$$\{(\text{EventsView.java}, \mathit{constant}, \texttt{COLUMN\_TIME})\} \Rightarrow \left\{ \begin{array}{lll} (\text{EventsView.java}, & \mathit{constant}, & \texttt{COLUMN\_COUNT}), \\ (\text{EventsView.java}, & \mathit{field}, & \texttt{columnHeaders[]}) \end{array} \right\} \quad (6.1)$$

Der Pfeil $\Rightarrow$ ist keine logische Implikation, sondern besagt nur, dass möglicherweise auch die Konstante `COLUMN_COUNT` und das Array `columnHeaders[]` geändert werden sollten, wenn die Konstante `COLUMN_TIME` bzw. Code davor oder dahinter geändert wurde.

Formal ist eine REGEL ein Paar (x_1, x_2) von zwei disjunkten Entitäten-Mengen x_1 und x_2. In der Regel $x_1 \Rightarrow x_2$ wird x_1 als BEDINGUNGSTEIL („antecedent“) und x_2 als AKTIONSTEIL („consequent“) bezeichnet.

Die von eROSE berechneten Regeln sind nicht hundertprozentig genau, sondern nur stochastische Interpretationen der Transaktionen, aus denen die Regeln gebildet werden, und basieren auf der Menge und Qualität der vorliegenden Daten.

Die HÄUFIGKEIT einer Menge x in einer Menge T von Transaktionen ist formal definiert als $\mathrm{frq}(T, x) = |\{t \mid t \in T, x \subseteq t\}|$.

Die Qualität einer Regel wird durch zwei Werte bestimmt:

- UNTERSTÜTZUNG: Gibt an, aus wie vielen Transaktionen die Regel abgeleitet wurde. Formal ist die Unterstützung einer Regel $x_1 \Rightarrow x_2$ durch eine Menge von Transaktionen T definiert als $\mathrm{supp}(T, x_1 \Rightarrow x_2) = \mathrm{frq}(T, x_1 \cup x_2)$.
- KONFIDENZ: Gibt die Stärke des Aktionsteils der Regel an und wird aus dem Verhältnis der Anzahl der Transaktionen, die den Bedingungsteil betreffen, zu der Anzahl der Transaktionen, die den Aktionsteil betreffen, berechnet. Formal ist die KONFIDENZ einer Regel $x_1 \Rightarrow x_2$ definiert als $\mathrm{conf}(T, x_1 \Rightarrow x_2) = \frac{\mathrm{frq}(T, x_1 \cup x_2)}{\mathrm{frq}(T, x_1)}$.

Im Beispiel aus Abschnitt 6.2.1 bedeutet das: Die Konstante `COLOR_TIME` wurde in zwei Transaktionen geändert, bei einer dieser Transaktionen wurden auch die Konstante `COLUMN_COUNT` und das Array `columnHeaders[]` geändert. Damit ergibt sich für die Unterstützung der Wert 1. Der Wert für die Konfidenz berechnet sich als $1/2 = 0,5$.

Angenommen, ein Entwickler habe eine Reihe von Änderungen $\delta_1 \circ \delta_2 \circ \cdots \circ \delta_k$ vorgenommen. Die Menge der geänderten Entitäten (bezeichnet als SITUATION) sei $\Sigma = entitites(\delta_1 \circ \delta_2 \circ \cdots \circ \delta_k)$.

Im Beispiel wurde Code nach der Konstanten `COLOR_TIME` eingefügt. Die Situation ist daher:

$$\Sigma = \{(\text{EventsView.java}, constant, \texttt{COLUMN\_TIME})\} \tag{6.2}$$

Für die Berechnung der Vorschläge für weitere Änderungen wendet eROSE matchende Regeln an. Eine Regel matcht eine Menge von geänderten Entitäten, wenn diese Menge gleich dem Aktionsteil ist. Die Menge der Vorschläge für eine Situation Σ und eine Menge von Regeln R ist definiert als die Vereinigung aller matchenden Regeln:

$$\mathrm{apply}(\Sigma, R) = \bigcup_{(\Sigma \Rightarrow x_2) \in R} x_2$$

Im Beispiel der Situation Σ aus (6.2) und der Regel r aus (6.1) schlägt eROSE den Bedingungsteil von r vor:

$$\mathrm{apply}(\Sigma, \{r\}) = \left\{ \begin{array}{lll} (\text{EventsView.java}, & constant, & \texttt{COLUMN\_COUNT}), \\ (\text{EventsView.java}, & field, & \texttt{columnHeaders[]}) \end{array} \right\}$$

Zur Berechnung der Regeln verwendet eROSE den Apriori-Algorithmus. Dieser Algorithmus beginnt mit minimalen Werten für Unterstützung und Konfidenz und berechnet dann die Menge aller Regeln in zwei Phasen. Eine Menge von Entitäten sei als HÄUFIG bezeichnet, wenn sie mehr als die minimale Unterstützung hat.

Phase 1 Der Algorithmus bildet für jede Transaktion eine Menge von Entitäten, die in der Transaktion vorhanden sind. Diese Menge vergrößert sich bei jedem Durchlauf, da eine solche Menge von Entitäten nur dann häufig sein kann, wenn ihre Teilmengen häufig sind. Am Ende ergibt sich die Menge F aller häufigen Mengen von Entitäten.

Phase 2 Der Algorithmus berechnet die Regeln aus den Mengen in F. Für jede Entitäten-Menge $E \in F$ werden die Regeln $E - X \Rightarrow X$ berechnet ($X \subseteq E$). Es werden also Regeln gebildet, die nicht die vollständige Menge der Entitäten im Bedingungsteil haben. Stattdessen kommt die Teilmenge X, die aus E herausgenommen wurde, in den Aktionsteil, um damit später die Vorschläge berechnen zu können. Diese Regeln haben alle die gleiche Unterstützung, aber unterschiedliche Konfidenz. Nur Regeln mit einer größeren als der minimalen Konfidenz werden zurückgegeben.

Normalerweise werden alle Regeln im Voraus berechnet. Da dies aber mehrere Tage dauern kann, setzen die eROSE-Entwickler zwei Optimierungen ein:

- ABHÄNGIGE AKTIONSTEILE: Es werden nur Regeln gebildet, deren Aktionsteile ähnlich zum Aktionsteil der aktuellen Situation Σ sind.
- EINZELNE BEDINGUNGSTEILE: Es werden nur Regeln gebildet, deren Bedingungsteil nur aus einer Entität bestehen. Das ist ausreichend, da eROSE sowieso die Vereinigung aller Bedingungsteile berechnet.

6.2.4 Auswertung des Ansatzes

Zimmer, Weißgerber, Diehl und Zeller haben für die folgenden vier Aspekte anhand von acht großen Open-Source-Projekten untersucht, welche Qualität die Änderungsvorschläge von eROSE haben (vgl. [ZWDZ04b]):

- NAVIGATION DURCH DEN CODE: Es sei eine einzelne geänderte Entität gegeben. Kann eROSE den Entwicklern auf andere Stellen im Code verweisen, die typischerweise auch geändert werden sollten?
- VERHINDERUNG VON FEHLERN: Kann eROSE Fehler verhindern? Weist eROSE z.B. auf eine nicht geänderte Entität hin, nachdem mehrere andere Entitäten geändert wurden?
- ABGESCHLOSSENHEIT: Angenommen, dass alle zu ändernden Entitäten geändert wurden. Schlägt eROSE fälschlicherweise weitere Änderungen vor?
- GRANULARITÄT: Wie gut sind die Ergebnisse von eROSE, wenn es nicht auf Methoden-Ebene (und kleineren Entitäten) arbeitet, sondern auf Dateien?

6.2.4.1 Navigation durch den Code

Nach einer geänderten Entität kann eROSE 15 Prozent aller weiteren zu ändernden Entitäten der gleichen Transaktion vorschlagen. In 64 Prozent aller Transaktionen verweisen die ersten drei Vorschläge auf die richtigen Stellen im Code.

Während beim Eclipse-Source-Code genau der Durchschnittswert von 15 Prozent erreicht wird, ist der Wert bei GCC mit 28 Prozent deutlich höher. KOffice kommt dagegen nur auf acht Prozent. Der Grund ist, dass in KOffice viele neue Funktionen eingefügt werden, während bei GCC der Code stabil ist und hauptsächlich nur Fehler behoben werden.

6.2.4.2 Verhinderung von Fehlern

Wenn in einer Transaktion eine Änderung vergessen wurde, kann eROSE im Schnitt nur vier Prozent dieser Fälle erkennen. Durchschnittlich ist jeder zweite Vorschlag korrekt.

Für GCC liegt die Korrektheit bei 81 Prozent und fehlende Änderungen werden bei jeder fünften Transaktion erkannt. Bei KOffice ist nur jeder vierte Vorschlag (24 Prozent) korrekt und fehlende Änderungen werden nur in 0,3 Prozent aller Fälle entdeckt.

6.2.4.3 Abgeschlossenheit

Durchschnittlich in nur zwei Prozent aller Fälle schlägt eROSE noch weitere Änderungen vor, obwohl bereits alle notwendigen Änderungen vorgenommen wurden.

Die Werte der acht Open-Source-Projekte liegen in diesem Aspekt mit einem bis zwei Prozent dicht beieinander. Nur bei GCC schlägt eROSE immerhin in etwa jedem zwanzigsten Fall noch eine weitere Änderung vor (4,7 Prozent).

6.2.4.4 Granularität

Wenn statt feiner Granularität (Methoden, Variablen, Konstanten etc.) eine gröbere Granularität verwendet wird, schlägt eROSE 26 statt 15 Prozent der Dateien vor, die in einer Transaktion geändert wurden. Die Korrektheit der ersten drei Vorschläge steigt von 64 auf 70 Prozent. Bei KOffice steigt der Wert sogar von acht auf 24 Prozent.

Trotz der höheren Prozentwerte sind die Vorschläge von eROSE mit gröberer Granularität aber gleichzeitig weniger nützlich, da nur noch Dateinamen angegeben werden, nicht aber die zu ändernde Methode.

6.2.5 Ähnlicher Ansatz von A. T. T. Ying

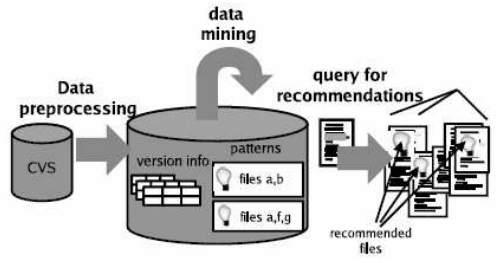


Abbildung 6.4: Drei Stufen im Ansatz von A. T. T. Ying (aus [Yin03])

Parallel und unabhängig zur Entwicklung von eROSE hat Annie Tsui Tsui Ying im Rahmen Ihrer Masterarbeit an der University of British Columbia einen ähnlichen Ansatz entwickelt. Wie in Abbildung 6.4 zu sehen ist, verwendet auch dieser Ansatz Datamining, um aus der Versionshistorie Muster und Regeln zu berechnen und daraus wiederum dem Entwickler Vorschläge für weitere Änderungen anbieten zu können.

Im Unterschied zu eROSE arbeitet dieser Ansatz aber nur auf der Ebene von Dateien und nicht z.B. auf der Ebene von Methoden. Ein weiterer Unterschied ist, dass die eROSE-Entwickler beim Datamining Beziehungsregeln verwenden, während Yings Ansatz analysiert, wie häufig sich Teile des Codes in einer Transaktion wiederholen. Dadurch ist hier nur der Wert für die Unterstützung relevant, nicht aber der Wert für die Konfidenz.

Ying hat im Gegensatz zu Zimmer, Weißgerber, Diehl und Zeller anhand der Open-Source-Projekte Eclipse und Mozilla auch die Qualität der korrekten Änderungsvorschläge untersucht. Für einen C-Programmierer ist es z.B. nicht sehr hilfreich, wenn er darauf hingewiesen wird, dass nach der Änderung von `example.h` auch die Datei `example.c` geändert werden sollte. Ying hat die Vorschläge daher in die Kategorien „überraschend“, „neutral“ und „offensichtlich“ eingeordnet. Die meisten Änderungsvorschläge für Eclipse und Mozilla fielen in die Kategorien „neutral“ oder „offensichtlich“. Die Kategorie „überraschend“ trat nur selten auf, da die meisten Quellcode-Dateien dieser Projekte direkt oder indirekt zusammenhängen.

6.3 Weitere Ansätze

Neben eROSE gibt es weitere Ansätze, um mit Hilfe der Versionshistorie die Entwicklung von Software zu erleichtern, indem Vorschläge für weitere Änderungen gegeben, verwandte Bugreports und Mailinglisten-Einträge angezeigt oder die letzten Änderungen einer Methode dargestellt werden.

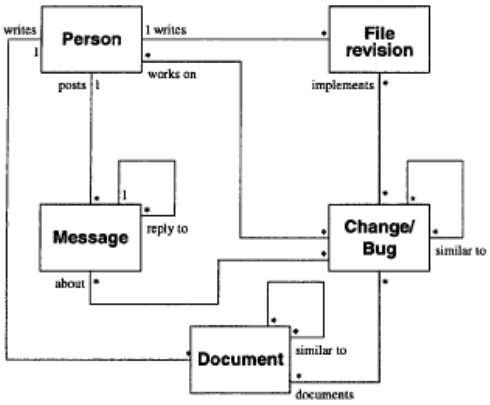


Abbildung 6.5: Beziehungen zwischen Artefakten, die Hipikat analysiert

6.3.1 Hipikat

Hipikat ist ein gemeinsames Projekt der University of British Columbia, des IBM Ottawa Software Lab und des National Research Council of Canada. Der Name stammt aus der Westafrikanischen Sprache Wolof und bedeutet übersetzt „weitgeöffnete Augen“.

Im Gegensatz zu eROSE berücksichtigt es nicht nur Versionhistorien, sondern auch Daten aus Bugtracking-Systemen wie Bugzilla, Online-Dokumentationen und Mailinglisten oder Newsgroups, um dem Entwickler bei der Programmierung von Software zu helfen. Ein weiterer Unterschied zu eROSE ist, dass Hipikat Vorschläge für weitere Änderungen nur auf Dateiebene bietet, während eROSE standardmäßig die Vorschläge auf kleinerer Ebene (z.B. Methoden) bietet.

Abbildung 6.5 zeigt die vier verschiedenen Artefakt-Typen, die Hipikat analysiert:

- Bug-Reports, Feature-Requests und Änderungswünsche aus Bugzilla-Einträgen
- Versionen von Dateien in Versions-Kontrollsystemen (bisher nur CVS)
- Nachrichten aus Mailinglisten und Newsgroup
- Dokumente, z.B. Beschreibungen über das Design einer Software auf den Webseiten eines Projekts

Diese Artefakte werden von den Mitgliedern des Projekts oder anderen Personen erstellt. Bugzilla-Einträge bilden einen zentralen Punkt in der Abbildung, da z.B. in einer Version einer Datei ein Änderungswunsch implementiert sein kann, da es weitere Dokumente zu diesem Wunsch geben kann und da mehrere Personen z.B. in einer Newsgroup über die gewünschte Änderung diskutieren können.

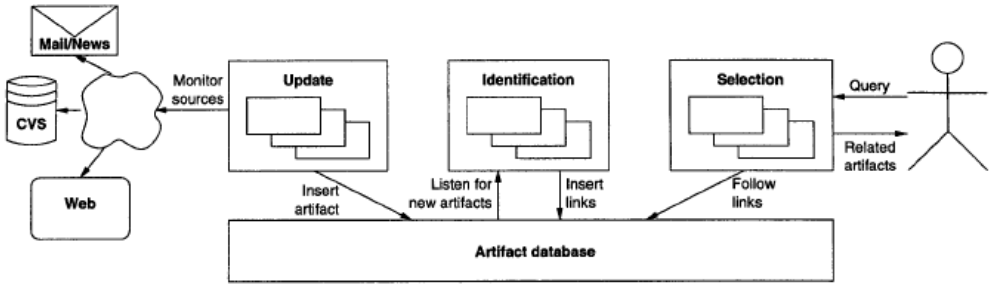


Abbildung 6.6: Server-Architektur von Hipikat

Die Struktur des Hipikat-Servers ist in Abbildung 6.6 dargestellt. Die Hauptkomponente ist die Datenbank, die die Artefakte verwaltet. Der Server hat drei Funktionen, die in Modulen gekapselt sind.

1. AKTUALISIERUNG („Update“): Dieses Modul ist in vier Untermodule aufgeteilt. Ein Modul durchsucht die Bugzilla-Seiten, ein Modul durchsucht die Versionshistorie eines CVS-Servers, ein weiteres Modul durchsucht über das NNTP-Protokoll Newsgroups und das letzte Modul durchsucht z.B. die Eclipse-Website. Neue oder geänderte Artefakte werden in die Datenbank eingefügt und Listener im Identifizierungsmodul werden über die Aktualisierungen benachrichtigt.

2. IDENTIFIZIERUNG („Identification“): Mehrere Untermodule analysieren die Artefakte in der Datenbank. Der so genannte „log-matcher“ durchsucht z.B. die Kommentare in der Versionshistorie nach IDs von Bugzilla-Einträgen und der „newsgroup-thread-matcher“ analysiert die „References“-Einträge von Newsgroup-Postings und (re)konstruiert daraus zusammenhängende Threads.

3. AUSWAHL („Selection“): Dieses Modul analysiert und beantwortet Anfragen des Entwicklers mit Hilfe der Artefakt-Datenbank. Die Ergebnisse werden im XML-Format zurückgegeben.

Für das Eclipse-Projekt konnte Hipikat im September 2002 auf 21.668 Bugzilla-Einträge (sowie weitere 72.536 Kommentare), 125.429 CVS-Versionseinträge, 36.864 Newsgroup-Postings und auf 1.459 Webseiten zurückgreifen.

Abbildung 6.7 zeigt die Suchmaske und die Ergebnisansicht innerhalb von Eclipse. Als Beispiel wurde hier über die Suchmaske nach dem Bugzilla-Eintrag mit der ID 116 gesucht. Unterhalb der Ergebnisansicht zeigt Hipikat Verweise auf ähnliche Artefakte an (vgl. Abbildung 6.8).

Neben der Suchmöglichkeit über die Suchmaske bietet Hipikat auch die Möglichkeit, über ein Kontextmenü die Funktion „Query Hipikat“ aufzurufen. Wenn z.B. in der „Console“-Ansicht eine Exception angezeigt wird, kann man über diese Funktion Hipikat nach verwandten Artefakten wie Bugzilla-Einträgen suchen lassen.

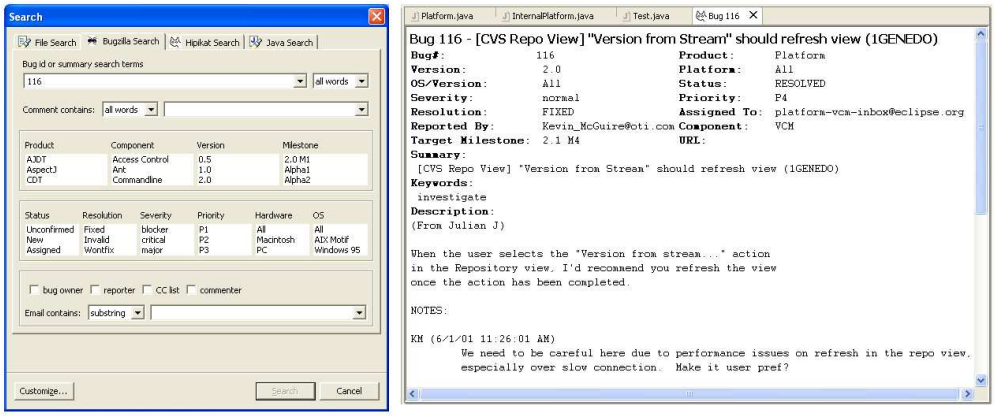


Abbildung 6.7: Suchmaske und Ergebnisansicht von Hipikat (aus [CM03b])

Die Ergebnisse von Hipikat sind nur Berechnungen und Analysen, die keine hundertprozentige Genauigkeit bieten können. In [CM03a] haben Davor Cubranic und Gail C. Murphy für das Eclipse-Projekt die Genauigkeit untersucht. Am 24. August 2002 gab es 9.418 Bugs, die als geschlossen markiert waren. Die Identifizierungs-Untermodule „log-matcher“ und „activity-matcher“ konnten für 60 Prozent (5.688) der Bugs Verknüpfungen zur Versionshistorie in der Artefakt-Datenbank herstellen. Allerdings wurden auch für weitere 2.810 Bugs, die noch nicht geschlossen waren, solche Verknüpfungen hergestellt.

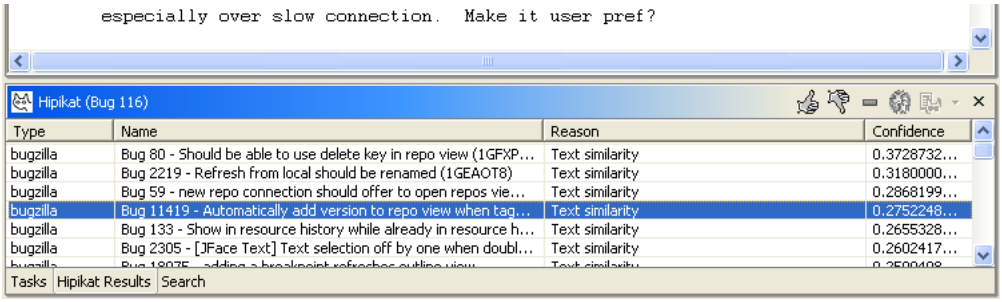


Abbildung 6.8: Hipikat zeigt ähnliche Bugs unterhalb der Ergebnisansicht in Eclipse an

6.3.2 CVSSearch

CVSSearch ist ein von Annie Chen, Eric Chou, Joshua Wong, Andrew Y. Yao, Qing Zhang, Shao Zhang und Amir Michail an der University of New South Wales entwickeltes webbasiertes Programm, um nach Teilen des Quellcodes anhand von CVS-Kommentaren zu suchen. Der Vorteil gegenüber Kommentaren im Code ist, dass gerade bei größeren Open-Source-Projekten kaum Kommentare im Code vorhanden oder nur von schlechter Qualität sind, während die CVS-Kommentare hauptsächlich dazu dienen, die anderen Entwickler über die aktuellen Änderungen zu informieren, so dass die Qualität der Kommentare hier meist deutlich besser ist.

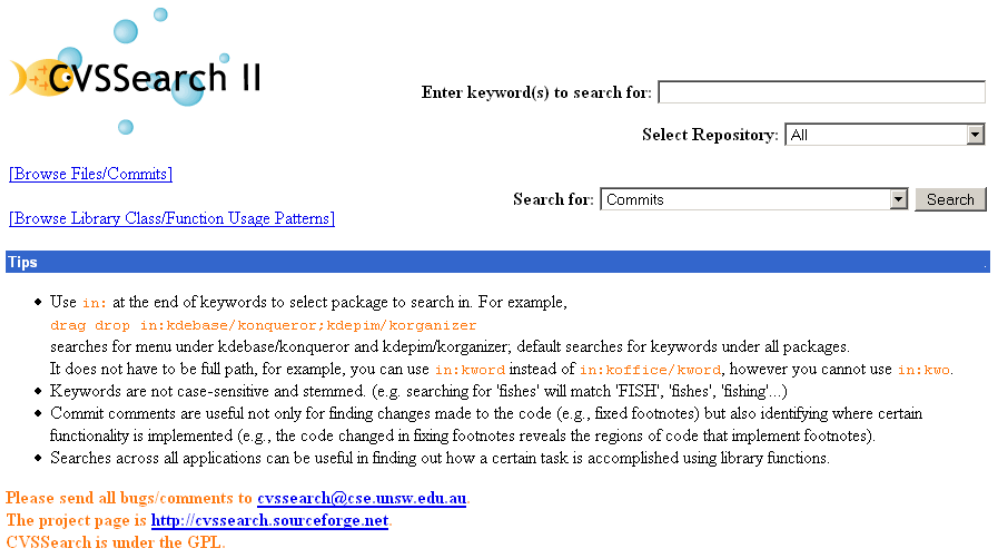


Abbildung 6.9: Suchmaske von CVSSearch

Abbildung 6.9[1] zeigt die Suchmaske von CVSSearch. Die Ansicht der gefundenen Dateien und eine Detailansicht zu einer Datei sind in Abbildung 6.10 zu sehen.

CVSSearch benutzt eine Abbildung zwischen den CVS-Kommentaren und den Code-Zeilen, auf die sich die Kommentare beziehen. Diese Abbildung wird nur für die Zeilen der aktuellsten Version einer Datei erstellt, während aber alle Versionen dieser Datei betrachtet werden. Wenn z.B. die Zeile 3 in der aktuellsten Version zum ersten Mal in Version 1.2 eingefügt wurde und danach in den Version 1.4 und 1.5 geändert wurde, dann muss Zeile 3 mit den Kommentaren aus den Versionen 1.2, 1.4 und 1.5 verknüpft werden. CVSSearch benutzt die „Managing Gigabytes“-Datenbank und speichert für jede Zeile die verknüpften Kommentare.

Wenn über die Suchmaske nach einem oder mehreren Begriffen gesucht wird, kombiniert CVSSearch die Ergebnisse aus seiner Datenbank mit den Ergebnissen von *grep*. Auf diese Weise wird aus zwei Perspektiven gesucht: zum einen der Blick in den Code, zum anderen der Blick auf das, was die Entwickler über den Code schreiben.

Für eine Auswertung wurden durch Studenten insgesamt 703 Suchanfragen gestellt. Davon wurden 40 Prozent durch die Suche in den CVS-Kommentaren am besten beantwortet, 32 Prozent am besten durch die Suche mit *grep* und 28 Prozent wurden mit beiden Varianten gleich gut beantwortet.

6.3.3 Version Editor

Version Editor ist ein von David L. Atkins an den Bell Laboratories entwickelter Editor, der auf die Editoren *vi* und *Emacs* aufsetzt. Er arbeitet mit Versionskontrollsy-

[1] http://web.archive.org/web/20030806094001/horn.cse.unsw.edu.au/~cvssearch/Query.cgi

Abbildung 6.10: Ergebnisübersicht und -details von CVSSearch (aus [CCW+01])

```
String FindSource(String base, String dir) {
    DIR * dirp = opendir(dir);
    for (int i = 0; i < NS; ++i) {        // Loop over suffix list
        String tmp = base + suffix[i];    // Target name to find
        for (dirent *de = readdir(dirp); de != NULL; de  = readdir(dirp))
           if (tmp == de->d_name) {       // We found it, stop looking
               return tmp;
           }
        rewinddir(dirp);
    }
    closedir(dirp);
    return "";      // No match was found
}

"findsource.c", line 13 of 19
```

Abbildung 6.11: Codebeispiel (aus [Atk04])

stemen zusammen, die das Source Code Control System (SCCS) oder das Revision Control System (RCS) unterstützen – SCCS und RCS wurden inzwischen von CVS abgelöst.

Die Idee des Version Editor ist es, nicht nur den aktuellen Code zu zeigen, sondern auch diejenigen Zeilen anzuzeigen und zu kennzeichnen, die seit dem letzten Checkout geändert oder gelöscht wurden. In der Standardkonfiguration des Version Editor zeigt er Zeilen, die geändert wurden in fetter Schrift an. Gelöschte Zeilen werden unterstrichen dargestellt. Zusätzlich werden für die Zeile, in der sich der Cursor befindet, Informationen zur letzten Änderung angezeigt (Zeilennummern der Änderung, Kommentar zum Commit).

```
String FindSource(String base, String dir) {
    DIR * dirp = opendir(dir);
    String result;     // The filename, if found
    for (int i = 0; i < NS; ++i) {        // Loop over suffix list
        String tmp = base + suffix[i];    // Target name to find
        for (dirent *de = readdir(dirp); de != NULL; de = readdir(dirp))
           if (tmp == de->d_name) {       // We found it, stop looking
               result = tmp;
               break;
               return tmp;
           }
        rewinddir(dirp);
    }
    closedir(dirp);
    return result; // Return the found name (may be null)
    return "";      // No match was found
}

Deleted by MR 595 by vz,97/11/15,approved [Stop source search at 1st match]
MR 467 by dla,97/09/21,integrated [Find source using list of suffixes]
"findsource.c", line 15 of 23
```

Abbildung 6.12: Version Editor zeigt geänderten Code (hinzugefügte Zeilen fett, gelöschte Zeilen unterstrichen; aus [Atk04])

Als Beispiel sei ein großes Programm angenommen, das u.a. Dateien ausliest und die-

se auch mit Daten füllt. Das Programm sei lange Zeit ohne Probleme gelaufen, aber es wurde nun entdeckt, dass es plötzlich zu viele gleichzeitig geöffnete Dateien gibt. Abbildung 6.11 zeigt eine der Methoden, die Dateien öffnen und schließen. Während die Methode `FindSource` auf den ersten Blick richtig erscheinen mag, ist die Anzeige der Funktion mit Version Editor (vgl. Abbildung 6.12) hilfreich, um den Fehler zu erkennen. Der Editor zeigt hier zusätzlich die geänderten und gelöschten Zeilen an, so dass der Entwickler auf einen Blick sehen kann, was an dieser Methode geändert wurde. Der Entwickler kann nun besser erkennen, dass die Methode aus der `for`-Schleife heraus verlassen wird, dass aber weiterhin die Datei erst nach der `for`-Schleife geschlossen wird.

6.3.4 VssConneXion

VssConneXion ist ein von EPocalipse Software entwickeltes Plugin für Borland Delphi und bietet eine Anbindung von Delphi an Visual SourceSafe, ein von Microsoft entwickeltes Versions-Kontrollsystem. Neben Standardfunktionen wie Checkout und Commit bietet es auch eine Option namens „Source Analysis“.

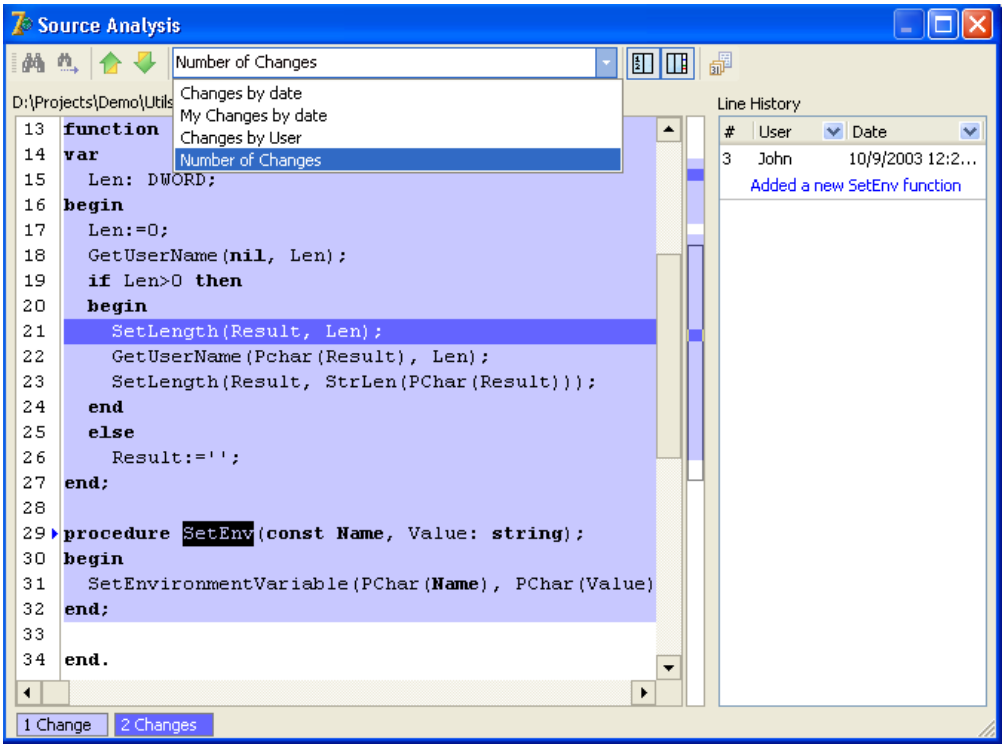


Abbildung 6.13: VssConneXion zeigt an, wie häufig die einzelnen Zeilen geändert wurden (aus [EPo06])

Das Konzept ist das gleiche wie beim Version Editor, allerdings werden die Informationen zu gelöschten und geänderten Zeilen nicht direkt im Editor angezeigt, sondern in einem separaten Fenster. Dieses Fenster bietet vier verschiedene Ansichten:

- Änderungen der einzelnen Zeilen mit Angabe des Änderungsdatums
- Änderungen des Entwicklers, der sich an der Visual SourceSafe-Datenbank angemeldet hat
- Entwickler, die die einzelnen Zeilen geänderten Zeilen geändert haben (vgl. Abbildung 6.13)
- Anzahl der Änderungen der einzelnen Zeilen (vgl. Abbildung 6.14)

Die Angaben zum Entwickler, zur Anzahl oder zum Datum werden durch farbliche Unterlegung dargestellt. Zusätzlich wird zur aktuell markierten Zeile die Versionshistorie mit der Versionsnummer, dem Namen des Entwicklers, dem Änderungsdatum und dem Kommentar angezeigt.

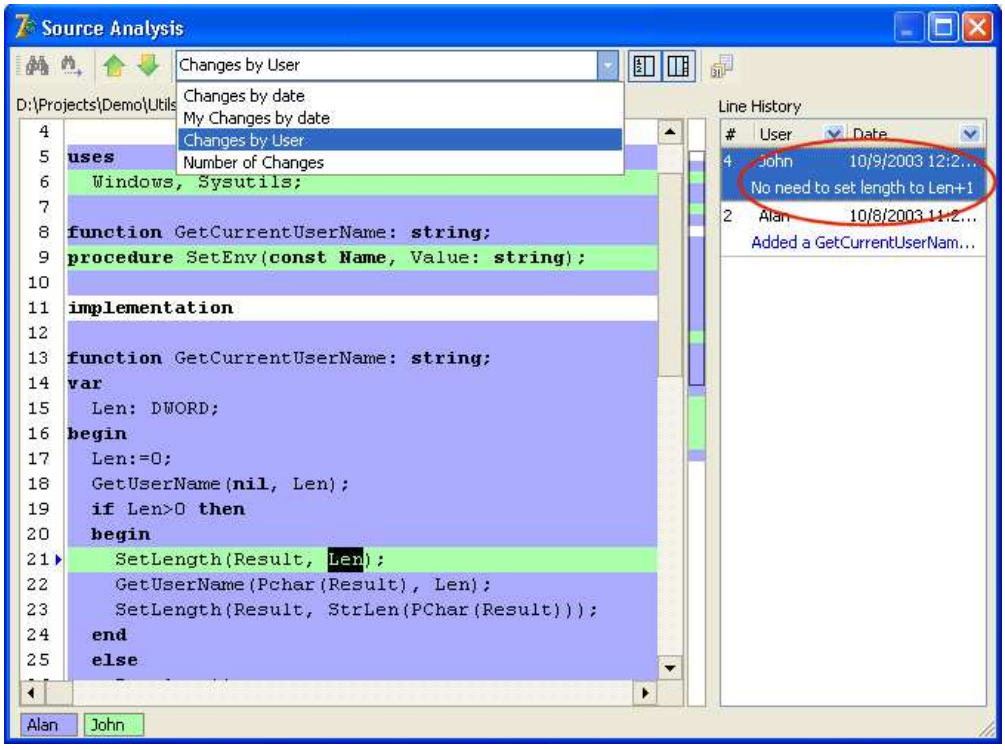


Abbildung 6.14: VssConneXion zeigt an, welcher Entwickler welche Zeile (zuletzt) geändert hat (aus [EPo06])

6.4 Bewertung und Ausblick

eROSE kann zwar durchschnittlich nur einen relativ geringen Teil (15 Prozent) der weiteren zu ändernden Entitäten in einer Transaktion vorschlagen, allerdings verdoppelt sich dieser Anteil, wenn der Code wie bei GCC stabil ist. Gleichzeitig sind die Vorschläge durchschnittlich in zwei Drittel aller Fälle korrekt. Somit kann man sich

nicht allein auf die Vorschläge von eROSE verlassen, bekommt aber häufig eine wertvolle Hilfe.

Hipikat konnte in der Genauigkeitsanalyse 60 Prozent der Bugs mit der Versionshistorie verknüpfen. Somit kann man sich auch auf Hipikat nicht allein verlassen, bekommt aber durch die Verknüpfung von Code, Bug-Reports und Mailinglisten-Einträge eine gute Unterstützung.

Die Analyse zu CVSSearch zeigte, dass die Strategie, sowohl mit Hilfe der Versionshistorie als auch mit Hilfe von *grep* zu suchen, sinnvoll ist. Zwar konnten 28 Prozent der etwa 700 Suchanfragen mit beiden Suchvarianten gleich gut beantwortet werden, aber 40 bzw. 32 Prozent der Anfragen besser über die Kommentare in der Historie oder mit *grep*.

Hinter Version Editor und VssConneXion steckt das gleiche Konzept, durch die grafische Oberfläche ist das „Source Analysis“-Fenster aber deutlich übersichtlicher und damit auch sinnvoller zu verwenden als im Version Editor.

Das Eclipse-Plugin eROSE ist als Alpha-Version 0.0.5 zum Download verfügbar, die Version 0.1.0 ist für Juni 2006 geplant. Version 1.8.0 von Hipikat wurde im Juli 2003 fertiggestellt, zur Zeit ist aber – wegen eines geplanten Redesign des Clients – kein Download möglich. Von CVSSearch war die Version 2.0 Beta2 verfügbar[2], zur Zeit ist aber nur der direkte Zugriff auf den CVS-Server möglich. Vom Version Editor konnte keine Download-Version gefunden werden. VssConneXion ist in Version 4 als 30-Tage-Testversion und als Vollversion verfügbar.

eROSE und VssConneXion werden weiterentwickelt, bei Hipikat und CVSSearch liegen die letzten Änderungen mehr als ein Jahr zurück und Version Editor war nie zum freien Download verfügbar. Die drei erstgenannten Tools sind aus meiner Sicht nützlich, leider ist aber keine installierbare Version von Hipikat im Internet zu finden.

6.5 Zusammenfassung

In dieser Arbeit wurden verschiedene Ansätze vorgestellt, um dem Entwickler auf verschiedene Weisen bei der Programmierung zu unterstützen. eROSE und Hipikat sind Plugins für die weit verbreitete Software-Entwicklungsumgebung Eclipse. CVSSearch ist webbasiert und Version Editor setzt auf die bekannten Editoren *vi* und *Emacs* auf. VssConneXion ist ein Plugin für Borland Delphi.

Alle Ansätze verwenden die Versionshistorie, um entweder Vorschläge für weitere Änderungen (eROSE, Hipikat) anzuzeigen, Code-Fragmente zu suchen (CVSSearch), hinzugefügte bzw. gelöschte Code-Zeilen (Version Editor) oder die Anzahl bzw. Autoren der Änderungen (VssConneXion) anzuzeigen. Hipikat greift zusätzlich zur Versionshistorie auch auf Webseiten, Bugzilla-Datenbanken, Mailinglisten und News-

[2] http://web.archive.org/web/20031217162927/http://cvssearch.sourceforge.net

groups zu. CVSSearch verwendet neben der Versionshistorie auch *grep*, um aus zwei Perspektiven heraus Code-Fragmente zu suchen.

Literaturverzeichnis

[Atk04] David L. Atkins. *The Story of the Version Editor*, 2004. `http://www.cs.uoregon.edu/~datkins/ve.html`.

[CCW+01] Annie Chen, Eric Chou, Joshua Wong, Andrew Y. Yao, Qing Zhang, Shao Zhang, and Amir Michail. CVSSearch: Searching through Source Code using CVS Comments. In *IEEE International Conference Software Maintenance (ICSM)*, pages 364–374, 2001.

[CM03a] Davor Cubranic and Gail C. Murphy. Hipikat: Recommending Pertinent Software Development Artifacts. In *Proc. 25th International Conference on Software Engineering (ICSE)*, pages 408–418, Mai 2003.

[CM03b] Davor Cubranic and Gail C. Murphy. *Sample Scenarios*, 2003. `http://www.cs.ubc.ca/labs/spl/projects/hipikat/scenarios.html`.

[EPo06] EPocalipse Software. *EPocalipse Software – VssConneXion*, 2006. `http://www.epocalipse.com/vcx.htm`.

[Yin03] Annie Tsui Tsui Ying. Predicting Source Code Changes by Mining Revision History. Master's thesis, University of British Columbia, Canada, Oktober 2003.

[ZWDZ04a] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. *Data Mining Version Histories*, 2004. `http://www.st.cs.uni-sb.de/papers/icse2004/gradkoll.pdf`.

[ZWDZ04b] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining Version Histories to Guide Software Changes. In *26th International Conference on Software Engineering (ICSE)*, 2004.

Chapter 7

Visualization of Software Evolution

Kejun Xu

Contents

Abstract

Version Control and Bug Tracking Systems contain a large amount of historical information that can be analyzed to provide deep insight into the evolution of a software project. And to gain higher level evolutionary information about large software systems is a key to validate past and future development processes. This paper will demonstrate a framework to analyze the software feature evolution based on three main sources: (1) Modification requests (MRs) taken from the version control system, (2) Problem Reports (PRs) taken from the bug tracking system, and (3) features extracted from the executable program. A RHDB (Release History Database) can be built up from these three main sources to store the filtered, validated and qualified set of data. Queries are used to retrieve the required set of data from the RHDB. A feature is a functionality provided by a software system and is implemented by the retrieved files. The retrieved results show the logically coupling of features. And the analysis of feature evolution, parts or phases of architectural deterioration can be based on the feature dependencies The visualization of the feature dependencies is to improve the understanding of a software system evolution and to reduce the complexity of the existing system.

7.1 Introduction

Software maintenance and evolution are characterized by their huge cost and slow speed of implementation. Yet they are inevitable activities in the software development process. Almost all software systems, which are successful in the software market, have to deal with large amounts of user-generated requests for change and improvements.

Software maintenance and evolution are the most expensive activities in the software process, consuming 60on a software system [1]. The majority of the software budget in large companies is devoted to adapting (i.e., evolving) existing software rather than developing new software. A very widely cited survey study by Lientz and Swanson in the late 1970s [LIEN80] exposed the very high percentage of life-cycle costs that were being expended on maintenance.

7.1.1 Basic definitions

Software maintenance is defined in IEEE Standard 1219 [IEEE93] as: The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.

A similar definition is given by ISO/IEC [ISO95], again stressing the post-delivery nature: The software product undergoes modification to code and associated documentation due to a problem or the need for improvement. The objective is to modify

the existing software product while preserving its integrity.

The term software evolution lacks a standard definition. Software evolution is a continuing process that encompasses not only ab initio development but all activities, enhancement, adaptation or fixing, that occurs after the first operational release [2].

From the definitions in the section 1.1, it can be concluded that software maintenance refers to the general post-delivery activities, and software evolution refers to the lifetime of software from its initial development to its shutdown.

Figure 1 shows the differences between them visually.

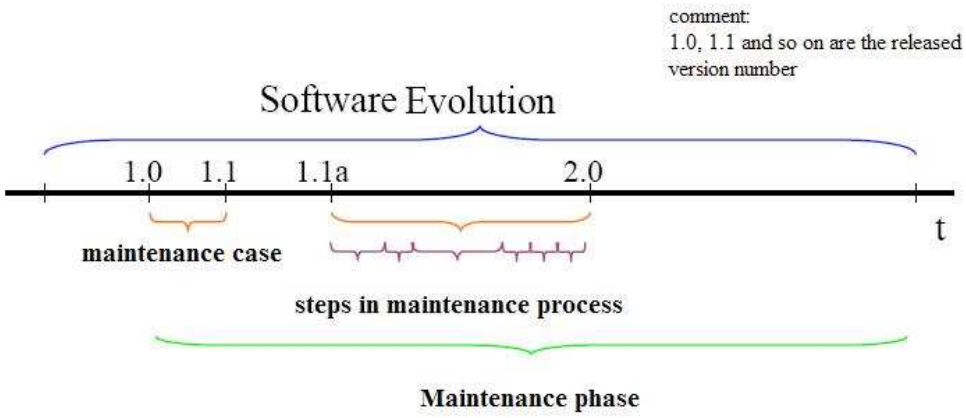


Figure 7.1: Software Maintenance vs. Software Evolution [3]

Lientz and Swanson in [LIEN80] categorized the maintenance activities into four classes:

- *Adaptive* modifications of the software product to properly interface with a changing environment
- *Perfective* enhancements to product to either add new capabilities or modify existing functions
- *Corrective* diagnosis and correction of errors
- *Preventive* preventing problems in the future.

7.1.2 Aspects of Software Maintenance and Evolution

Technical and managerial aspects are two essential aspects of software maintenance and evolution. Technical aspects are from the technical point of view to observe software evolution. There are six main research areas [4]:

- *Legacy systems and migration*: The legacy systems can be reused, tailored or migrated in the development process of a new software system.
- *Reverse engineering and program understanding*: Computer-Aided Software Engineering (CASE) tools help to obtain a graphical representation of the implementation of a software system.
- *Re-engineering*: Re-engineering is to fix bugs, to correct functional errors, or to alter the implementing strategy by revising the source code.
- *Transformation-based evolution*: It is not to change the functionalities of software system, but to refactor or restructure the source code.
- *Traceability and consistency maintenance*: the changes to the files of a software system should be traceable and consistent.
- *Visualisation and analysis of evolution histories*: The historical change data can be used to analyze the evolution of a software system and the evolution result can be visualized to gain a better understanding.

Managerial aspects are from the software development process and management point of view to understand software evolution. There are three main research areas [4]:

- *Evolutionary process models*: the development of a software system
- *Software configuration management*: the management towards the software changes and versions.
- *Estimation techniques*: how much cost will have to be spent to the changes?

This paper will focus on visualizing the software evolution by analyzing the information got from the Version Control System, Bug Tracking System, and the extracted feature from the executable program. Thus, the activity of this paper is from the technical aspect of software evolution.

7.2 Version Control and Bug Tracking System

Many software projects use a version control repository, such as CVS, ClearCase, or SourceSafe to record the modifications of their source code. These repositories keep track of every change to any source file of the project, including metadata about the change, such as author and date when it happened. Over time, the amount of revisions to a project becomes enormous. For example, the Mozilla project is composed of 35,000 files which have been modified 450,000 times in 5.5 years of development (from March 1998 to Aug. 2003) by 500 different developers.

Since these changes describe interesting aspects of the evolution of a software system, they are a very valuable source for retrospective analysis techniques which explore, for

example, change rates (number of changes within a certain amount of time), or error proneness (number of errors) of a software project.

Version historical data may be enhanced with the data from Bug Tracking Systems that report the bugs, change requests, and problems of software systems in the past maintenance activities. A database storing the historical information, enabling reasoning about the past and anticipating future evolution of software projects can be built up by both information sources.

The following section is going to give a short introduction to the CVS Version Control System and the Bugzilla Bug Tracking System.

7.2.1 CVS

CVS, the Concurrent Versioning System, is arguably the most widely used version control management system available in the market and has become a de-facto standard in the development of open source projects.

The information stored in the CVS repository is quite valuable as it can help answering many questions. For instance, it can assist developers in knowing who has modified which files and when; it can also help the administration in trying to understand the modification patterns of the project and the way the different team members interact. Finally, it can help in the recovery of the evolution of the project.

While CVS is a very powerful tool, there are many difficulties to extract and visualize the valuable information. One of the main disadvantages of CVS is that it is not transaction oriented. In other words, when a developer proceeds to "commit" a group of changes to a number of files, CVS does not keep track of all the files modified by this commit operation. It treats each change to a file independently of the other files included in the commit. After the commit has taken place, CVS does not know which files were modified together. This information is important because it highlights coupling among these files: if two files are modified at the same time, it shows that they share something in common. The commit operation is referred as a modification request (MR). A MR is therefore a collection of revisions to files that are modified at the same time.

CVS is a group of command-line program. Several GUI applications have been built, such as, winCVS, cvsWeb, LinCVS and so on. Some integrated development environments (such as Eclipse) provide a GUI to CVS. The tools are created around the CVS commands and options and provide nothing more than a fancy GUI to the actual commands.

7.2.2 Bugzilla

Bug report data from the Bugzilla Bug Tracking System can be seen as the additional information to the modification requests in CVS. Access to the Bugzilla system is enabled via HTTP and reports can be retrieved in XML format. The information will be used later to classify the corresponding modification requests found in CVS. This enables the identification of error-prone files or modules which are candidates for reimplementation or re-design.

Besides some administrative information such as contact information, mailing addresses, discussion, etc., the Bug Tracking System also provides some interesting information for the evolutionary view such as, bug id, bug severity, affected product or component. The bug report data is in XML format. Some important tags are explained as following:

- *Bug id:* This ID is referenced in modification requests. Since the IDs are stored as free text in the CVS repository, the information cannot be reliably recovered from the Bug Tracking System.
- *Bug status* (status whiteboard): Describes the current state of the bug and the states can be unconfirmed, assigned, resolved, etc.
- *Product*: Determines the product which is affected by a bug. In Mozilla project, the products can be Browser, MailNews, NSPR, Phoenix, Chimera, etc.
- *Component:* Determines which component is affected by a bug. In Mozilla project, the components can be Java, JavaScript, Networking, Layout, etc.
- *Dependson:* Declares which other bugs have to be fixed first, before this bug can be fixed.
- *Blocks*: List of bugs which are blocked by this bug.
- *Bug severity*: This classification field for a bug report. The values can be blocker, critical, major, minor, trivial, enhancement.
- *Target milestone:* Possible target version when changes should be merged into the main trunk.

7.3 Software Evolution

It is very challenging to gain the higher-level evolutionary information about large software system, as the complexity of software system is increasing and the architecture of software system is deteriorating over its lifetime.

In chapter 2, Concurrent Version System (CVS) and Bug Tracking System (Bugzilla) have been discussed. It is useful to analyze the information from CVS and Bugzilla in

several ways: (1) detect the logically coupled files; (2) identify the error prone classes with affected components or products; (3) estimate the code maturity with respect to the probability of remaining bugs.

This chapter will focus on how the evolution of a large software system can be analyzed based on the sources from the MRs taken from CVS, the PRs taken from Bugzilla, and the features extracted from the executable program.

7.3.1 Framework of software evolution analysis system

In [5], M. Fischer and H. Gall have described the framework how to analyze and visualize feature evolution. The analysis is based on three main sources: (1) MRs taken from CVS, (2) PRs taken from Bugzilla, and (3) the executable program to extract feature information. From these three sources a Release History Database (RHDB) can be built up, which provides a filtered, validated and qualified set of data. Queries are used to retrieve the required set of data from the RHDB. The retrieved results show the coupling property among the retrieved files, and these files can be categorized by a feature. The evolution of features is along with the growth of files over time. The results of the evolutionary analysis on the feature level can greatly support the illustration of dependencies and changes in large software systems.

Figure 2 shows the information flow and process steps.

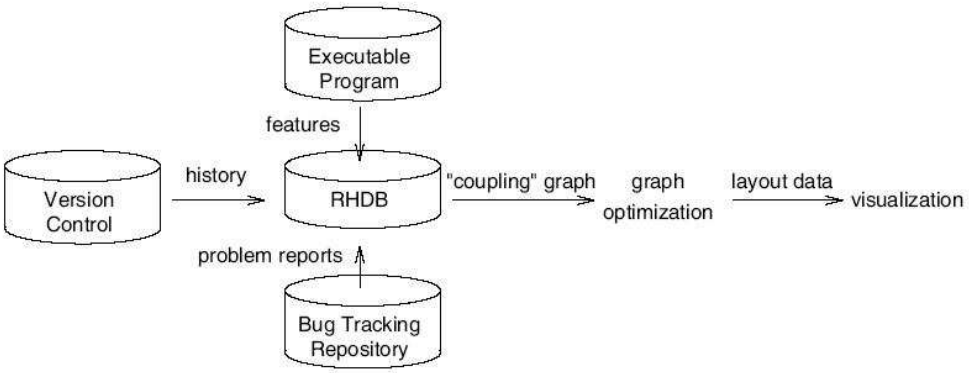


Figure 7.2: Information flow and process steps

7.3.1.1 Feature extraction

A Feature is a prominent or distinctive aspect, quality, or characteristic of a software system or systems [6]. So a feature can be seen as a realized functional requirement and an observable result of value to a user. Features are used in communication between users and developers, it is important to know which features are affected by the future functional modifications of a software system.

The goal of feature extraction process is to gain the information about an executable program to map the abstract concept of features onto a concrete set of files, which implement a certain feature. The extracted information is about particular releases of a product and can be used to apply evolutionary analysis on the feature level.

For the three sources of a RHDB, MRs and PRs are taken from the Version Control and Bug Tracking Repository respectively. And how can the features be extracted from the executable program? To extract the required feature data by utilizing code, the software reconnaissance technique [7] can be applied. Software Reconnaissance is based on a comparison of traces from different test cases. The target program is instrumented to produce a trace of components executed as each test is run. Then test cases are run, some "with" and some "without" the desired feature. For example to locate the spell checker of a word processor, the programmer would run several tests that include a spell check and several others that do not. The "marker" components for the spell check feature are found by taking the set of components executed in the "with" tests and subtracting the set of components executed in the "without" tests. GNU tools [8] have been also used successfully to extract feature data [9].

In [5], M. Fischer and H. Gall have described the feature extracting by executing the defined scenarios. A scenario is a sequence of user inputs triggering actions of a system that yields an observable result to an actor [12]. A scenario is said to execute a feature if the observable result is executed by the scenario's actions. A scenario may execute multiple features. Scenarios resemble use cases but do not include options or choices, so a use case subsumes multiple scenarios.

The feature extracting process is described in the following way: the defined scenarios (see figure 3) were executed first, then the call graph information can be created using the GNU profiler. The call graph information was used to retrieve all functions and methods visited during the execution of a single scenario. The files containing these functions and methods are mapped to a feature, which shows the functionality of a software system and is a higher level abstract representation of these retrieved files.

7.3.1.2 Build up RHDB structure

The RHDB is very essential in whole analysis of feature evolution. How to populate a RHDB from version control and bug tracking systems has been investigated and described in [10]. M. Fischer, M. Pinzger, H. Gall in [13] have also described the detailed feature extraction process and how to import the extracted feature information into the RHDB database.

Figure 4 depicts the database structure showing the primary entities and their relationships. The entity projstruct reflects the hierarchical structure of the source tree. The entity evalresult is used to store the query results. Every file of the CVS repository has a corresponding entry in the cvsitem table storing the attributes extracted from the log file as described in list 1. To resolve the n:n relationship between symbolic names (i.e. tags) and revisions of files, the two entities cvsalias and cvsitemalias are intro-

Scenario	Description	Feature	Fill style	ID	Files
Core	Mozilla start / blank window / stop	Core		00	705
HTTP	TrustCenter.de via HTTP*	Http		01	28
HTTPS	TrustCenter.de via SSL/HTTP	Https		02	6
File	read TrustCenter.de from file	—		—	—
XML	XML Base (see http://www.w3c.org/)*	Xml		09	65
MathML	mathematic in Web pages*	MathML		08	13
About	'about:' protocol	About		10	3
fBlank	read blank html page†	Html		03	76
hBlank	blank html page via HTTP*	—		—	—
Image	—	Image		04	3
ChromeGIF	Mozilla logo: chrome://global/content/logo.gif	ImageGIF		07	4
PNG	image: Portable Network Graphics*	ImagePNG		05	10
JPG	image: Joint Photographic Experts Group*	ImageJPG		06	16

Figure 7.3: Features extracted from Mozilla

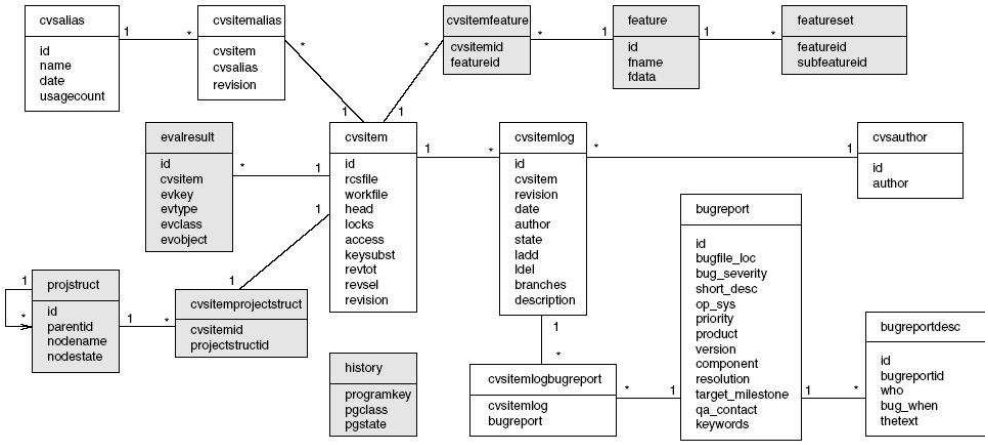


Figure 7.4: RHDB structure

duced. cvsalias holds the symbolic name information, cvsitemalias contains a record for each entry extracted from the symbolic names section found in log files. Data about modification reports is stored in the cvsitemlog table. It contains an entry for every modification entry found in the log file. Corresponding author information is handled by cvsauthor.

Bug reports are directly imported from the bug tracking system into the bugreport table. The current attributes of this entity are derived from the Bugzilla system and may be extended to address further bug tracking systems. Particularly, the link of bug reports with modification reports is important for software evolution analysis. This link is realized by the cvsitemlog-bugreport table as n:n relation. The table contains the bug report numbers found in the modification reports together with the respective modification report ID. The entity projstruct reflects the hierarchical structure of the source tree. Every node hosts subtree information such as, for example, number of files or lines added/deleted in this subtree. This data is used, for instance, in the process of creating module history information for the selection of modules with a certain size. The cvsitemfeature, feature, and featureset tables are used in the feature evolution analysis processes. The table history contains results which are valid for the whole evolution database such as time scale information or state information about executed queries.

7.3.1.3 Populating RHDB and analyzing the retrieved results

Figure 5 shows the process to import CVS and Bugzilla data into RHDB. First the initial source code tree can be created by checking out the source code directly from the Mozilla CVS repository. A script can be used to traverse through the source tree structure to retrieve the modification reports from the CVS repository. Each item of the source tree is described by the corresponding log information. The log contains the revision number, modification date, modification report text, etc. and can be parsed by a perl script and stored in RHDB and the following entities of the RHDB are populated: cvsitem, cvsitemlog, cvsauthor, cvsitemalias, cvsalias. Bug report identifiers are extracted from the modification reports. The bug ID found here are stored in the table cvsitemlogbugreport. The bug report IDs are used to retrieve bug report descriptions from the Bugzilla database, which can be accessed vis HTTP. The XML format bug reports are parsed and the extracted data are stored in tables bugreport, bugreportdesc. Now the CVS version control data combined with Bugzilla bug report data are stored in RHDB. After the extraction process in [13], all file IDs, which characterize a feature, together with the released ID are imported into RHDB.

The query results can be analyzed to provide a clear picture about a particular feature or a set of features. The hidden dependencies of files, which are structurally unrelated but logically coupled, have a good potential to show feature evolution. Grouping PRs can reveal hidden dependencies between features and it can also be used to identify groups of commonly modified program code.

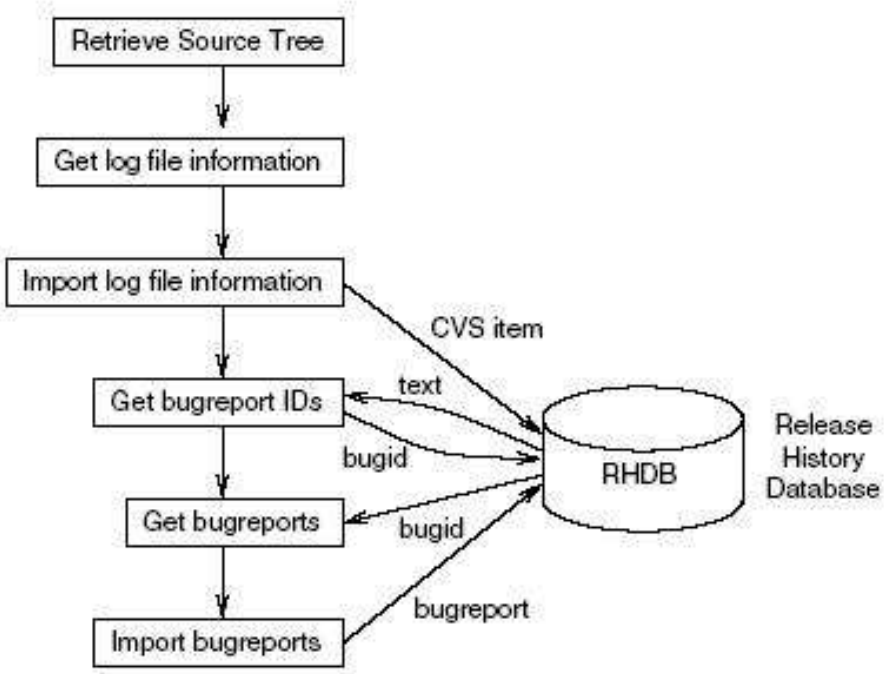


Figure 7.5: Import process

7.3.2 Visualizing the feature evolution

The value of visualizing the software evolution can be answered by its definition. Software visualization is a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing system under consideration[11].

In [5], M. Fischer and H. Gall have provided two views to visualize the feature evolution in large software systems. One is the feature view focusing on projecting the coupled PRs onto the selected features and the other one is the project view projecting PRs onto the directory structure of the project-tree.

7.3.2.1 Feature view

This view is to visualize the features and their dependencies via PRs. Figure 6 and 7 are taken from [5]. In figure 6, the left column lists all the features and each feature is described with a number and a name. The middle column lists the actual number of PRs that features have in common. The right column lists the total number of PRs for every feature within a period. Figure 7 visualizes the results got in figure 6. The thickness of lines shows the degree of coupling between two features.

	Feature										Period				
Feature	01	02	03	04	05	06	07	08	09	10	<2000	2000	2001	2002	Total
01/Http	0	20	10	7	0	2	2	0	7	4	24	63	155	129	371
02/Https		0	2	1	0	0	0	0	2	0	0	1	69	61	131
03/Html			0	6	0	1	46	16	29	1	87	116	140	122	465
04/Image				0	0	15	5	0	3	2	5	5	54	31	95
05/ImagePNG					0	0	0	0	0	0	0	0	3	5	8
06/ImageJPG						0	0	0	0	0	1	0	22	11	34
07/ImageGIF							0	17	36	0	15	61	75	38	189
08/MathML								0	67	0	2	9	32	72	115
09/XML									0	1	7	34	105	110	256
10/About										0	2	1	2	1	6

Figure 7.6: Total number of “couplings” between features

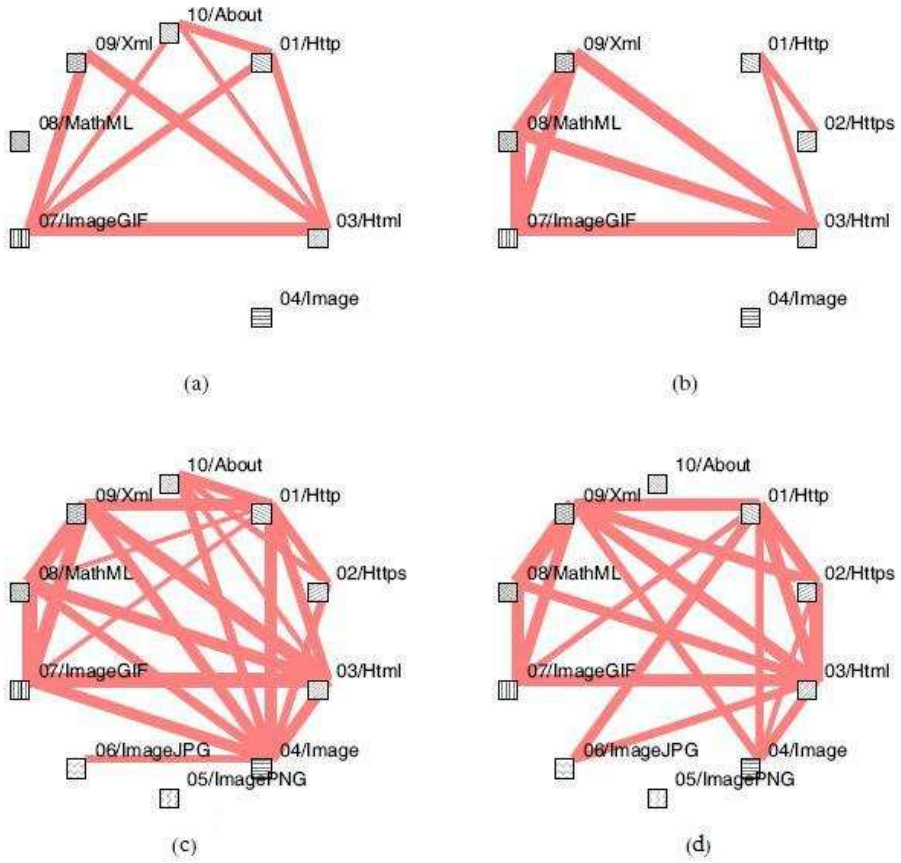


Figure 7.7: Dependencies between features: (a) 1999 (b) 2000 (c) 2001 (d) 2002

Figure 7 shows the dependencies between features within each year from 1999 to 2002. (a) and (b) have 6 features in the graphs. The feature “About” is removed and the feature “Http” is added in (b). A feature removed from the graph is possibly due to that no PRs are addressed to this feature or this feature is not coupled with some other features within this period. (c) has much more features and the dependencies between features are more complicated compared with (b). It shows that there are many development activities happened from the year 2000 to 2001 and more features are introduced in 2001. The dependencies between features are complicated. It is possible that the developers did not perform a sufficient design. (c) and (d) have the same 10 features in the graphs, but the dependencies between the features are different. The dependencies of features in (d) are less than in (c). For example, the dependence between ImageJPG and Image has been removed in 2002 and the dependence between ImageGIF and Html has become weaker compared with 2001. It is possible that there is noticeable effort spent for restructuring from 2001 to 2002.

In general, this method is very visual to view all the features provided by the software system and it is also very vivid to observe the feature evolution by analyzing the feature dependencies.

7.3.2.2 Project-tree view

This view is to visualize the reflection of PRs onto the directory structure of the project-tree. Figure 8 is taken from [5]. In figure 8, it shows the relation of features Http, Https, and Html via PRs. The project directory tree is shown with gray nodes connected by black dashed lines. The root node is labeled ’Root’ and the features are indicated by filled in boxes with different styles. Coupling between nodes are results of common PRs. The thicker line means that the number of PRs that the two nodes have in common is higher. There are three features Http, Https, and Html are depicted in figure 8. The nodes netwerk.base, netwerk.protocol.http, and security.manager.ssl are interesting since they are coupled via 90 PRs for base-http and 40 PRs for each of the other two edges [5]. This indicates a high degree of coupling between the features Http and Https.

7.4 Other approaches to software evolution

This paper is based on the research work of M. Fischer and H. Gall [5] and follows the approach they have worked out. There are some research papers providing the approach to analyze then visualize the software evolution based on release history information. Some of these research papers have used Mozilla as a case study to investigate the software evolution issue.

In [14], Taylor and Munro described an approach to visualize aspects of large software such as active areas, changes made, or sharing of workspace between developers across a project. The aspects are visualized by using animated revision towers and detail

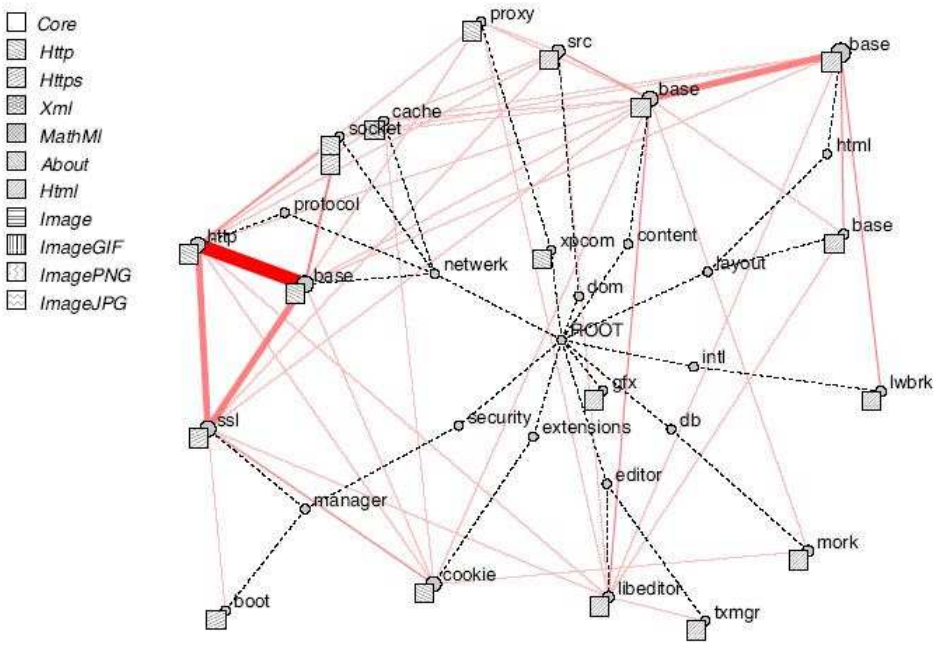


Figure 7.8: Relation of features Http, Https, and Html via PRs

views. Since their approach is purely based on revision history, additional important information such as PRs or feature data are not considered for visualization.

Similar to the framework described in this paper and used to produce the evolution results in [5], Draheim and Pekacki proposed a framework for accessing and processing revision data via predefined queries and interfaces [15]. And the linkage of their data model with other evolutionary information such as PRs, feature data as required for the analysis in [5] and making them accessible for external queries is not covered in their paper.

In [16], Kemerer and Slaughter used modification reports as the basis for their analysis. A refined classification scheme for modification reports (i.e. corrective, adaptive, perfective enhancement, and new program) for an analysis of ordered change events is described. As a result, they were able to reveal different phases of a system's life cycle. While they have investigated longitudinal evolution of software system, in [5] the focus is on visualizing hidden dependencies between components of a system reflected by any kind of traceable pattern, e.g., commonly and frequently changed modules or common PRs.

In [17,18], Lanza and Ducasse used rectangles in a matrix view to depicte several releases of a software system. The width and height of rectangles represent specific metrics (e.g., number of methods, number of instance variables of classes) according to the history of classes. Based on this generated evolution matrix, classes are assigned to different evolution categories such as, for example, pulsar (class grows and shrinks

repeatedly) or supernova (size of class suddenly explodes). While they focused on analyzing the evolution of source code, in [5] they focused on the features evolution of a software system.

7.5 Conclusion

This paper has described a framework to analyze the feature evolution of a software system based on the query results from the Release History Database (RHDB), which is built up from Modification Requests (MRs) taken from Concurrent Version System (CVS), Problem Reports (PRs) taken from Bugzilla, and the features extracted from the executable program and stores the general information about entities and artifacts of the software's source code and modification reports. This paper is from the perspective, which is to use a graphical representation to show the dependencies between features based on PRs, to analyze the evolution of software systems. Two specific views of feature relationships and dependencies of a large software system can be established: (1) the feature view provides a projection of PRs onto the files that realize a particular feature to indicate the hidden feature dependencies that have evolved over time. (2) the project view provides a projection of PRs onto the project directory structure of a system and depicts the logical coupling of software parts via features.

The approach to analyze software evolution in [5] can be used to point to parts and phases of architectural deterioration on the basis of feature dependencies. To visualize the analysis results of software evolution is to make use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing system. The visualization results can be used (1) to assess the point of time when some restructuring or reengineering activities should be started, and (2) to estimate the likely amount of resources which will be required from changing particular features.

Bibliography

[1] David Hearnden, Paul Bailes, Michael Lawley, Kerry Raymond. *Automating Software Evolution*. iwpse, pp. 95-100, Principles of Software Evolution, 7th International Workshop on (IWPSE'04), 2004

[2] M. M. Lehman, J. F. Ramil. *Effort Estimation from Change Records of Evolving Software*. ICSE 2000, Poster Summary

[3] http://seal.ifi.unizh.ch/evolution/ SW Wartung und Evolution

[4] http://w3.umh.ac.be/genlog/cours/evolution-EN.html

[5] Fischer M, Gall H. *Visualizing feature evolution of large-scale software based on problem and modification report data*. Journal of Software Maintenance and Evolution 16(6): 385-403, 2004.

[6] Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA, 1990; 8.

[7] Wilde N, Gomez JA, Gust T, Strasburg D. *Locating user functionality in old code*. Proceedings International Conference on Software Maintenance. IEEE Computer Society Press: Los Alamitos CA, 1992; 200-205.

[8] http://www.gnu.org The Free Software Foundation. GNU's Not Unix! Free Software Foundation: Boston MA, 1996-2004

[9] Eisenbarth T, Koschke R, Simon D. *Aiding program comprehension by static and dynamic feature analysis*.Proceedings IEEE International Conference on Software Maintenance (ICSM'01). IEEE Computer Society Press: Los Alamitos CA, 2001; 602-611.

[10] Fischer M, Pinzger M, Gall H. *Populating a release history database from version control and bug tracking systems*.Proceedings International Conference on Software Maintenance (ICSM'03). IEEE Computer Society Press: Los Alamitos CA, 2003; 23-32.

[11] C. Knight and M.Munro. *Comprehension with[in] Virtual Environment Visualisations*.Proceedings of the IEEE 7th International Workshop on Program Comprehension, pp4-11, May 5-7, 1999.

[12] T. Eisenbarth, R. Koschke, and D. Simon. *Aiding Program Comprehension by Static and Dynamic Feature Analysis*.In Proceedings of the International Conference on Software Maintenance. IEEE Computer Society Press, November 2001.

[13] Fischer M, Pinzger M, Gall H. *Analyzing and relating bug report data for feature tracking*.Proceedings 10th Working Conference on Reverse Engineering (WCRE). IEEE Computer Society Press: Los Alamitos CA, 2003; 90-99.

[14] Taylor CMB, Munro M.*Revision towers: A Proceedings 1st International Workshop on Visualizing Software for Understanding and Analysis*. IEEE Computer Society Press: Los Alamitos CA, 2002; 43-50.

[15] Draheim D, Pekacki L. *Process-centric analytical processing of version control data*. Proceedings 6th International Workshop on Principles of Software Evolution (IWPSE'03). IEEE Computer Society Press: Los Alamitos CA, 2003; 131-136.

[16] Gall H, Hajek K, Jazayeri M. *Detection of logical coupling based on product release history*. Proceedings International Conference on Software Maintenance. IEEE Computer Society Press: Los Alamitos CA, 1998; 190-198.

[17] Lanza M. *The evolution matrix: Recovering software evolution using software visualization techniques*. Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE'04). ACM Press: New York, 2001; 37-42.

[18] Lanza M, Ducasse S. *Polymetric views - a lightweight visual approach to reverse engineering*. IEEE Transactions on Software Engineering 2003; 29(2):782-795.

Kapitel 8

Assessment of Legacy Systems

Nelson Rodrigues Lé

Inhaltsverzeichnis

8.1 Einführung

Legacy Systeme (Altsysteme) sind Großsysteme, die schon vor einiger Zeit entwickelt wurden, und seit dem zwar weiterentwickelt wurden, allerdings zumeist auf dem damaligen Stand der Technik. Das betrifft sowohl die Software als vor allem auch die Hardware des Systems. Dabei handelt es sich keineswegs um schlechte Systeme. Zumeist sind es Systeme die über Jahre hinweg zuverlässig ihren Dienst vollbracht haben. Trotzdem stellen viele dieser (Alt-)Systeme für die Unternehmen, in denen sie eingesetzt werden, eine Last dar, denn die Kosten, um so eine Altsystem zu unterhalten, sind nicht selten immens. Es gibt einige Möglichkeiten wie man mit einem Altsystem verfahren kann. Man kann ein neueres System entwickeln, welches das Altsystem ersetzt, man kann das Altsystem als Teil eine neuen Systems weiterführen, oder es an einigen Stellen erneuern und überarbeiten, so dass es in Zukunft leichter ist, das System weiterzuentwickeln.
Laut Cimitile und Fasolino [CF01] wurde ca. 78% der IT-Investitionen der italienischen Regierung in 1995 für die Erhaltung von Altsystemen ausgegeben. Es stellte sich heraus das das System aus 230.000 KLOC (tausend Zeilen Code) bestand, wobei große Teile des Systems den Charakteristika eines Altsystems entsprachen. Um die Bewertung eines solchen Altsystem und um die Entscheidung welche Maßnahmen ergriffen werden sollen, geht es in dieser Ausarbeitung. Es werden verschiedene Methoden präsentiert um ein bestehendes Altsystems zu analysieren. Dabei werden wir besonderen Augenschein auf die Methoden des RENAISSANCE Projekts setzten.

8.2 Das RENAISSANCE Projekt

8.2.1 Das Projekt

RENAISSANCE ist ein von der europäischen Union gefördertes Forschungsprojekt, das es Softwareunternehmen ermöglichen soll, systematisch und objektiv Legacy Systeme zu bewerten, und Entscheidungen über die Zukunft solcher Altsysteme treffen zu können.

8.2.2 Ziele

Ziel des RENAISSANCE-Projekts, ist es systematische Methoden zur Analyse von Altsystemen zu entwickeln, die bei der Entscheidung zur Weiterentwicklung oder der Neuentwicklung helfen. RENAISSANCE basiert auf folgenden zwei Grundideen:

- Neuentwicklung muss Unternehmen- und Projektspezifisch sein.
  Die Bewertungsmethoden sind so entwickelt, dass sie in jeglichen Unternehmen und Projekten eingesetzt werden können.

- Sowohl an neu entwickelten Prozessen als auch an neu entwickelten Systemen muss stetig weiterentwickelt werden.
  Ein System, das aus einem Altsystem hervorgeht, muss erweiterbar sein. Weiterentwicklung steht im Mittelpunkt einer solchen Software.

Um eine Beurteilung über ein System treffen zu können, muss man es es zuerst verstehen. Dafür stellt RENAISSANCE einige Bewertungsmethoden, die dabei helfen sollen das System aus technischer und aus Sicht des Unternehmens tiefgründig zu analysieren, um eine Entwicklungsentscheidung treffen zu können. Ransom, Sommerville und Warren [RSW98] stellen einige typische Fragen und mögliche Antworten einer solchen Bewertungsanalyse vor:

- *Ist das System unverzichtbar für das Unternehmen, welches es einsetzt?* Idealerweise sollte ein System nicht essentiell für das Bestehen eines Unternehmens sein.

- *Welche sind die Unternehmensziele?* Um eine Bewertung über das System abgeben zu können muss man die Struktur und die Aufgaben eines Unternehmens kennen, damit man diese auf das System übertragen kann. Unternehmensziele lassen sich in Systemanforderungen umsetzen.

- *Welche Systemanforderungen gibt es?* Die Systemanforderungen ergeben sich aus den Unternehmenszielen, und sollten von Anfang an klar definiert sein, damit man sich anschließend ein Urteil erlauben kann, ob diese im System auch umgesetzt wurden.

- *Auf welche Laufzeit ist das Systems beschränkt?* Die Laufzeit eines Systems ist von Beginn an beschränkt. Sei es durch Lizenzen der Software, oder einfach nur durch die Zeiten die Software und Hardware als Lebensdauer besitzen. Im schnelllebigen IT-Markt gilt sowohl Software als auch Hardware nach wenigen Jahren als überholt.

- *Welche Laufzeit erwartet das Unternehmen von dem System?* Ist ersichtlich das ein in System in absehbarer Zeit unwichtig für ein Unternehmen wird, so ist klar dass keine immense Kosten in ein Reengineering gesteckt werden. Soll das System allerdings über längere Zeit eines der Schlüsselelemente des Kerngeschäfte des Unternehmens sein oder werden, dann sollte man über solche Maßnahmen nachdenken.

- *In welchem Zustand befindet sich das System?* Eine Applikation in einem schlechten technischen Zustand kostet meist Zeit und Geld. Sofern es mit der erwarteten Laufzeit des Systems übereinstimmt, lohnt sich der Aufwand das System in einen besseren technischen Zustand, wie z.B. bessere Erweiterbarkeit, zu bringen.

- *Ist das Unternehmen offen für Neuerungen?* Die Einstellungen eines Unternehmens gegenüber Veränderung beeinflusst den Erfolg eines solchen Projekts.

- *Hat das Entwicklungsteam genügend Kapazitäten?* Es bedarf einiger Erfahrung mit Software-Wartung, sowie auch genügend Erfahrung und Fachwissen der Mitarbeiter. Ein solches Unternehmen benötigt auch die entsprechenden arbeitserleichternden Werkzeuge.

8.2.3 Vorbereitungen

Um die Beurteilungsmethoden an das Unternehmen anzupassen, bedarf es einiger Vorbereitungen. Es müssen zwei Parameter berücksichtigt werden:

- Bewertungstechniken:
  Sowohl Expertenmeinungen als auch Metriken sollten bei der Bewertung von Altsystemen berücksichtigt werden. Dabei müssen genügend Personen mit dem entsprechendem Fachwissen zur Verfügung stehen.

- Bewertungsgenauigkeit:
  Die Genauigkeit einer Bewertung kann durch Iterationen mit verschiedenen Variablen aber auch durch feingranularere Messungen verfeinert werden.

Als Vorbereitung können in einem Unternehmen beispielsweise erstmal alle vorhanden Systeme einer oberflächlichen Analyse unterzogen werden, um dann festzustellen welche Systeme, oder Teile eines Systems, für detailliertere Bewertungsanalysen in Frage kommen. In dieser Phase kommt es noch nicht sonderlich darauf an das System tiefgründig zu analysieren und bewerten, sondern auf eine schnelle Beurteilung der gegebenen Umstände.

8.2.4 Experten, Metriken und Informationsquellen

Expertenmeinungen sind in allen Bereichen gefragt. Allerdings werden dafür Leute benötigt, die über das nötige „Know-How“ verfügen, und dass nicht nur aus Entwicklersicht.

- Zusätzlich benötigt man *Unternehmensexperten*, die sich sehr gut im Unternehmen, welches die zu analysierende Software betreibt, auskennen. Denn nur jemand der die Geschäftsprozesse des Unternehmens kennt, kann auch beurteilen, wie gut diese in der Applikation umgesetzt wurden. Für diese Rolle eignen sich führende Angestellte die seit längerem das Unternehmen kennen.

- Man benötigt außerdem einen, oder mehrere *Anwendungsexperten*, also jemanden der sich mit der Applikation auskennt und seine Funktionen kennt und beherrscht. Prädestiniert für die Rolle sind zumeist die Anwender. Also Mitarbeiter des Unternehmens die die Applikation tagtäglich benutzen.

- Ein Entwickler der die Applikation erweitert bzw. pflegt ist auch von Nutzen. Dieser *Administrationsexperte* ist mit Abstand der Experte den man am seltensten im Unternehmen vorfindet. In diesem Fall muss man sich meist mit Source Code, Dokumentationen, Pflichtenheften und allen möglichen Quellen begnügen, die es zu dem Altsystem gibt. Metriken können zur Bewertung eines Systems auch herangezogen werden. Oft kann man durch recht simple Metriken, wie z.B. die Anzahl der Codezeilen (LOC) oder der Anweisungen (NOS) bereits mögliche Problemstellen im System erkennen. Auch können Metriken zur Bestimmung der Programmkomplexität, wie z.B. die McCabe-Metrik, Aufschluß über den Stand des Systems geben, und ein Indiz dafür sein, auf welchen Programmteilen man den Hauptaugenmerk beim fundierten Testen setzt.

8.2.5 Bewertung des wirtschaftlichen Wertes

Ziel bei der Bewertung des wirtschaftlichen Wertes des Systems, ist es herauszufinden wie eng das Altsystem mit dem Geschäftsablauf verwurzelt ist. Es geht darum den Aufwand abzuschätzen, den man benötigt um das Altsystem zu verändern, dabei ist es erst einmal egal ob das System erneuert, erweitert oder ausgetauscht werden soll. Es muss festgestellt werden ob das Altsystem im Geschäftsablauf eine periphere Rolle spielt, und man es ohne weiteres über eine gewisse Zeit abstellen kann, um eine Neu- bzw. Weiterentwicklung und realen Umständen testen zu können, oder aber ob es so geschäftskritisch ist das man nur bedingt Zeit zur Verfügung hat, bzw. alles „on the fly“, im täglichen Geschäftsablauf, erledigt werden muss.
Die Bewertung des wirtschaftlichen Wertes wird in zwei Stufen unterschieden:

- Grobe Analyse
  In dieser Stufe, die bei jedem System zuerst durchgeführt werden sollte, treffen die Experten eine Vorselektion. Systeme mit geringem Wert für das Unternehmen werden aussortiert und die anderen werden weiter analysiert.

- Detaillierte Analyse
  Es handelt sich hierbei um eine sehr zeitintensive Analyse des wirtschaftlichen Wertes des System. Hierfür benötigt man geeignete Experten aus dem Unternehmen z.B. Fachkräfte die das System nutzen aber auch Führungskräfte die die Geschäftsprozesse kennen. Anhand eines systemspezifischen Fragebogens werden grundlegende Daten zum System gesammelt wie zum Beispiel:

  – Mängel und Probleme des Systems
  – Verknüpfungen des Systems mit anderen Nachbarsystemen
  – Kostenabschätzung für einen Ausfall des System über eine Zeitperiode

  Ein solcher Fragebogen muss jedes mal aufs Neue erstellt werden, denn er muss auch auf spezifische Feinheiten des Systems und des Unternehmens eingehen.

Außerdem hilft es oft die Benutzung des System neutral beobachten zu lassen. Einem Außenstehenden fallen oft Fehlverhalten im Arbeitsablauf auf, die beim Nutzer bereits routiniert sind und daher von diesem gar nicht als Fehlverhalten erkannt werden.

8.2.6 Bewertung der Infrastruktur

Zu der Infrastruktur eines Systems zählen Hardware und Supportsoftware unter Berücksichtigung der Infrastruktur des Unternehmens.

8.2.6.1 Hardware

Unter Hardware versteht man verschiedene Komponenten, die gewartet werden müssen. Dabei kann es vorkommen das Systeme über mehrere Orte verteilt sind, was eine Analyse des Systems dementsprechend erschwert. Es gibt für verschiedenste Komponenten des Systems unterschiedlich Wartungskosten. Es gilt zu Prüfen welche Komponenten im Handel noch erwerblich sind und welche bereits nicht mehr angeboten werden. Welche Teile des System haben noch Garantie, oder gar einen Wartungsvertrag, welche Komponenten stellen eine bereits veraltete Technologie dar. Dabei sollten alle Komponenten des Systems betrachtet werden, wie z.B. Großrechner, Arbeitsstationen, Laufwerke, Netzwerk, Drucker, Eingabegeräte und Anzeigegeräte. Wie die Bewertung des wirtschaftlichen Wertes kann auch die Hardware in zwei Stufen analysiert werden. In der ersten, oberflächlichen Analyse wird die Hardware als ganze von Experten unter die Lupe genommen. Bei der detaillierten Analyse werden dann die einzelnen Komponenten des Systems begutachtet. Dabei können zum Beispiel, sowohl bei der oberflächlichen als auch bei der detaillierten Analyse, folgend Merkmale begutachtet werden:

- Hersteller
- Wartungskosten
- Ausfallrate
- Alter
- Performanz
- Eignung für die Problemstellung
- Möglichkeit der Wiederbeschaffung
- Fehlerquote

Um ein solche Analyse zu bewerten empfehlen Ransom, Sommerville und Warren [RSW98] jeder Komponente, bei der detaillierten Analyse, bzw. dem Gesamtsystem

bei der oberflächlichen Analyse, eine Wert zwischen 0 und 4 zuzuordnen. Dabei geht die Skala von 0 wie „erfüllt die Anforderungen nicht“ bis 4 wie „erfüllt die Anforderungen voll und ganz“. Die Summe der Werte der einzelnen Komponenten sind dann der Gesamtwert des Systems. Ein Gesamtwert der die Hälfte der maximalen Gesamtpunktzahl nicht überschreitet, bedarf besonderer Aufmerksamkeit.

Beispiel für die Bewertung des Alters von Hardware

- 0 = Hardware kann man nicht mehr kaufen
- 1 = überholte Technik, nur noch bedingt erwerbbar
- 2 = Umbruch der Technik, sowohl neue als auch alte noch verfügbar
- 3 = Komponente zwar aktuell, aber neuere Technik bereits auf den Markt
- 4 = aktuelle Hardwarekomponente

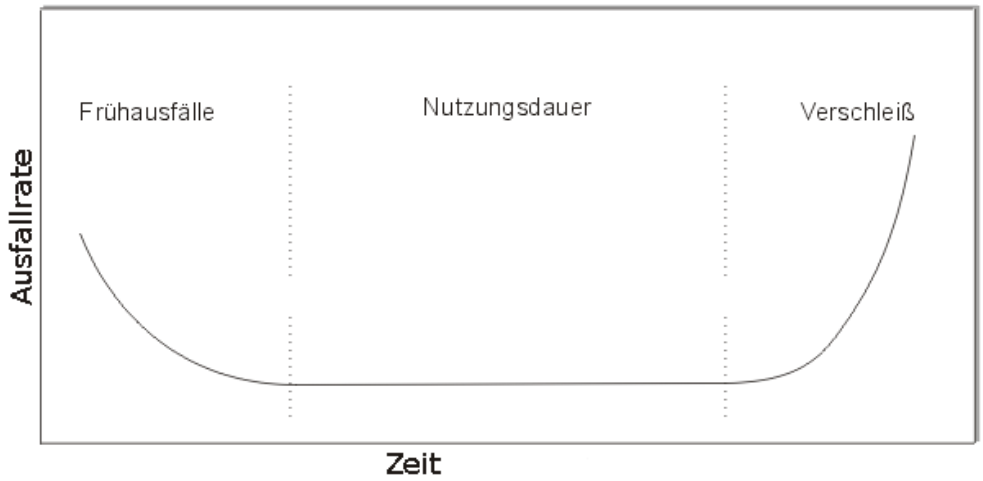


Abbildung 8.1: Eine „Badewannen-Kurve“

Die Laufzeit einer Hardwarekomponente verläuft typischerweise wie die sogenannte Badewannenkurve in Abbildung 8.1. Dabei fällt die Ausfallrate zu Beginn. Die Ausfälle während dieser Phase basieren hauptsächlich auf Fehler die bereits bei Auslieferung des Produktes vorhanden waren, wie z.B.

- Materialfehler
- Montagefehler
- Fertigungsfehler

Die Fehler werden dann durch Austausch oder Reparatur behoben und die Ausfallrate sinkt und erreicht eine in etwa konstant geringe Ausfallrate. Ausfallursachen während dieser Phase sind zumeist „Anwenderfehler“ wie

- Fehlbedienung
- falsche Umgebungsbedingungen
- fehlende Wartung

Nach einiger Zeit, die komponentenspezifisch ist, steigt die Ausfallrate dann steil an, man spricht nun von der Verschleißphase, denn die verarbeiteten Materialien haben nur eine begrenzte Lebensdauer, und verschleißen nach einiger Zeit bzw. nutzen sich ab.
Für das Gesamturteil der Hardware sollte die Bewertungen der einzelnen Komponenten gewichtet in das Gesamturteil einfließen. Zudem sollte die Verbindung zwischen Hard- und Software ebenfalls bewertet werden, d.h. wie wirkt sich ein Wechsel der Hardware auf die Software aus?

8.2.6.2 Software

Die Supportsoftware eines Systems besteht zumeist aus mehreren Komponenten (z.B. Betriebssystem, Datenbank, Netzwerksysteme, etc.) deren Wartung meist aus dem Einspielen der neuesten Updates besteht. Die einzelnen Komponenten sind oft voneinander abhängig, und darüber hinaus auch von der eingesetzten Hardware. Die zu bewertenden Merkmale sind die gleichen wie bei den Hardwarekomponenten, allerdings kann man diese noch um weitere Punkte ergänzen, wie:

- Lizenzkosten
- Häufigkeit von Updates/ BugFixes/ Patches
- Support

Auch hier sollte man bei der Bewertung berücksichtigen welche Abhängigkeiten die Software zum System hat, dafür ist es notwendig das Personen die die Bewertung der Software durchführen, diese Beziehung kennen und verstehen.

8.2.6.3 Unternehmensinfrastruktur

Unter die Unternehmensinfrastruktur fallen die Abteilung, die das System wartet und weiterentwickelt, sowie auch die Abteilung die das System anwendet. In einigen Unternehmen, bzw. bei einigen Systemen sind das ein und die selbe Abteilung, es kann aber auch sein das Wartung und Entwicklung aus dem Unternehmen ausgelagert wurde, und von einem anderen Unternehmen betreut wird.
Hierbei handelt es sich um ein schwer zu bewertenden aber sehr wichtigen Teil für das Gesamturteil. Ransom, Sommerville und Warren [RSW98] empfehlen folgende Faktoren für die Bewertung der Unternehmensinfrastruktur zu analysieren:

- *Unternehmensart und Anwender*
  Das Unternehmen sollte entweder eine qualifizierte Entwicklungsabteilung haben, oder sämtliche Entwicklungs- und Wartungsprozesse an ein anderes, dafür qualifiziertes, Unternehmen ausgelagert haben. Es sollte auch untersucht werden ob die Nutzer des Systems lediglich wiederkehrende Standardaufgaben mit dem System ausüben, oder komplexere Tätigkeiten durchführen.

- *Technisches Know-How*
  Egal ob die Entwicklungs- und Wartungsaufgaben ausgelagert wurden oder nicht. Die dafür zuständige Abteilung / Unternehmens muss über entsprechendes Know-How verfügen. Das spiegelt sich zum Beispiel in den entsprechenden Werkzeugen wieder, aber auch ob das Unternehmen für diese Tätigkeiten zertifiziert ist und nach einem Standard arbeitet.

- *Lehrgänge und Fortbildung*
  Sowohl die Entwicklungsabteilung als auch die Anwender profitieren von regelmäßigen Fortbildungen und Schulungen.

- *Qualifikationen der Systembetreuung*
  Wenn die Systembetreuung nur über unzureichendes Know-How verfügt, sind größere Veränderungen im System nicht ratsam.

- *Einstellung zu Veränderungen*
  Kleinere Unternehmens zeigen sich oft offener für Veränderungen als große bürokratisierte Unternehmen, die sich gerne quer stellen für jeglich Veränderungen.

8.2.7 Bewertung der Applikation

Die Bewertung der Anwendungssoftware des Altsystems lässt eine Menge Spielraum für die Genauigkeit der Untersuchung. Man unterscheidet abermals zwei Stufen:

- *Systemstufe.* Das System wird als Einheit bewertet, ähnlich wie groben Analyse bei der Bewertung von Hardware und Supportsoftware.

- *Komponentenstufe.* In dieser Stufe werden ein Teil, oder alle Komponenten des Systems analysiert und bewertet. Bei größeren Systemen sucht man sich repräsentative Komponenten zur genaueren Betrachtung aus. Denn größere Systeme bestehen oft aus Hunderten Komponenten und Subsystemen die man nicht alle detailliert bewerten kann. Auf Komponentenebene kann man wieder die Subsystem ohne jegliche Berücksichtigung derer Komponenten bewerten und in dem man auf die Komponenten der Subsysteme eingeht. In beiden Fällen sind sowohl Expertenbewertungen als auch Metriken von Nutzen. Folgende Merkmale lassen sich unter anderem bewerten:

  – Komplexität (z.B. McCabe-Metrik)
  – Daten

- Dokumentation
- externe Abhängigkeiten
- Fehlerprotokoll
- Größe (z.B. KLOC)
- Sicherheit

Ein Gesamturteil ergibt sich dann aus den gewichteten Teilergebnissen, ähnlich den zuvor erläuterten Bewertungen.

8.2.8 Auswertung

Nachdem man nun den wirtschaftlichen Wert des Systems, die Infrastruktur und die Applikation selbst analysiert und bewertet hat, muss man jetzt aus den gewonnen Daten ein Gesamtbewertung konstruieren. Sneed [Sne91] stellt zur Gesamtbewertung ein Auswertungsquadrant (Abbildung 8.2) vor. Mit diesem lässt sich eine Beurteilung des Systems anhand der erzielten Messungen aufstellen. Dabei werden die technische Qualität des Systems und der wirtschaftlichen Wert einbezogen. Die vorherigen Ergebnisse werden jeweils auf eine Skala von 0 bis 100 skaliert und in den Quadranten eingetragen. Dabei stellt die x-Achse den wirtschaftlichen Wert und die y-Achse die technische Qualität des Systems dar.

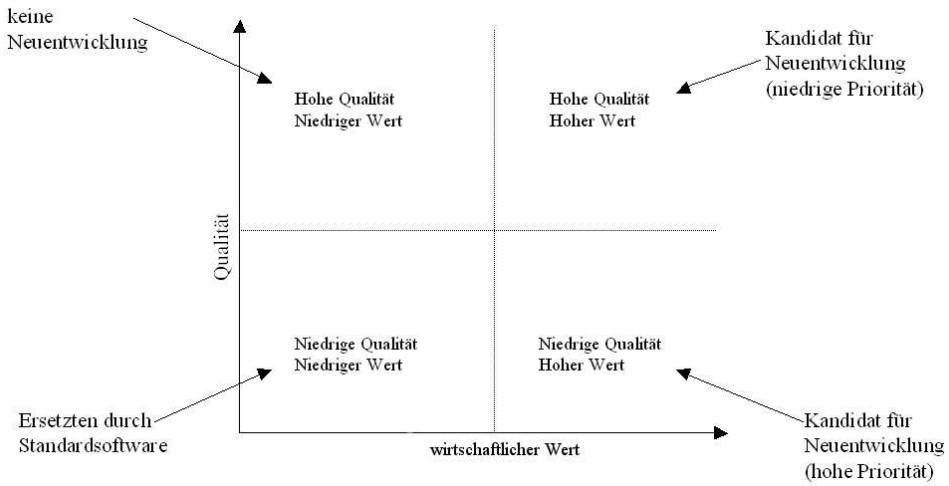


Abbildung 8.2: Auswertungsquadrant

Beispiel: Ein System das einen wirtschaftlichen Wert von 85 erzielt und und eine Qualitätswert von nur 30 wäre ein guter Kandidat für ein Reengineering.
Allerdings sollte das Ergebnis des Auswertungsquadranten nur als Richtwert verstanden werden. Gerade bei „knappen“ Ergebnisse, die an der Schwelle zu einem (oder mehreren) anderen Quadrat des Quadranten liegen, sollte man die Teilergebnisse nochmals genauer Untersuchen und gegeneinander abschätzen. Vor einer endgültigen Gesamtbewertungen sollte man folgende Punkte nochmals bedenken:

- *Analyse der Unternehmensinfrastruktur*
  Entscheidet man sich für eine Neuentwicklung des Systems, so muss man das auch unter erneuter Analyse der Unternehmensinfrastruktur betrachten. Denn ändert man beispielsweise das Konzept des Systems von einem prozeduralen auf ein objektorientiertes Konzept, so muss man darauf achten, das diese Entscheidung auch mit den Fähigkeiten der Mitarbeiter der Abteilung Entwicklung und Wartung übereinstimmt.

- *Analyse der Bewertungen*
  Stellen wir uns ein System mit hohem wirtschaftlichen Wert aber niedriger Qualität vor. Laut Auswertungsquadrant ein potentieller Kandidat für ein Reengineering. Stellt sich aber heraus, dass das schlechte Abschneiden der Qualität am schlechten Abschneiden eines Subsystems liegt und für dieses Subsystem in absehbarer Zeit ein Update mit vielversprechenden Änderungen vorliegt. So empfiehlt es sich zu warten, das Update einzuspielen, und die Qualität des Subsystems nochmals zu analysieren.

- *Entscheidungen basieren auf Detaillierte Analyse*
  In der oberflächlichen Analyse fallen oftmals kritische Aspekte, wie zum Beispiel auslaufende Supportverträge oder Lizenzen, nicht auf. Erst in der detaillierten Analyse lassen sich diese auffinden.

- *Betrachtung von externen Faktoren*
  Ein gutes Beispiel hierfür sind Gesetzesänderungen. Solche Probleme, die nicht akut aber langfristig absehbar sind, kann man beim Bewerten der Software, bzw. der Applikation, nicht berücksichtigen, man darf sie allerdings nicht aus den Augen verlieren, so dass sie ihren Weg in die Gesamtbewertung finden.

Wir sehen, es ist also auch eine Betrachtung außerhalb der Messungen notwendig um eine abgerundete und zuverlässige Gesamtbewertung des Altsystems abgeben zu können. Ein tiefgründiges Verständnis für die das System umgebenden Komponenten und externen Faktoren ist notwendig um ein Altsystem zu analysieren und die richtigen Entscheidungen bezüglich Neuentwicklungen zu treffen.

8.3 Legacy System Assessment

Natürlich gibt es auch andere Methoden um ein Altsystem zu bewerten. Canfora Cimitile [CC97] beschreiben zum Beispiel den Lebenszyklus (life cycle) eines Systems. Man geht davon aus das ein System eine kontinuierliche Entwicklung durch 4 Phasen durchläuft:

- *Vereinfachung*
  In dieser Phase geht es darum die Größe des Systems zu reduzieren, dadurch das unter anderem sogenannter „dead code“ also Programmteile, die nicht mehr

benötigt werden, oder im schlimmsten Fall nie benötigt wurden, eliminiert werden.

- *Einfache rückwirkende und präventive Wartung*
  In dieser Phase werden Fehler korrigiert, kleinere Anpassungen an geänderte Geschäftsprozesse, und das System gewartet. Da die Tätigkeit oft aus dem Unternehmen ausgelagert wird, und die Bezahlung an der Größe des Systems gekoppelt ist, ist es sinnvoll den Code in der ersten Phase zu reduzieren.

- *Komplexere adaptive Wartung*
  In dieser Phase werden größere Änderungen am Programm durchgeführt, wie zum Beispiel eine neue GUI, oder Migrationen auf Client/Server-Technologien.

- *Austausch*
  Zur dieser letzten Phase eines Systems gehört auch die Neuentwicklung des neuen Systems. Während dieser Phase werden am Altsystem ausschließlich Veränderungen in Notfällen durchgeführt.

Bevor ein System von einer Phase in die nächste übergeht, so muss man es vorher auf folgende 4 Aspekte analysieren:

- *wirtschaftlicher Wert*
  Wie auch beim RENAISSANCE-Projekt, wird hier der festgestellt wie eng das System mit den Geschäftsprozessen verwurzelt ist.

- *Modularität*
  Wie einfach lassen sich die einzelnen Komponenten des Systems identifizieren und von einander trennen. Sind sie unabhängig von einander?

- *Alterungsprozess*
  Drückt das Altern des Systems, verursacht durch versäumte Änderungen und Weiterentwicklung, aus.

- *Verschleiß*
  Drückt das Altern des Systems, verursacht durch kontinuierlich durchgeführte Änderungen und Weiterentwicklung, aus

8.3.1 Messen des wirtschaftlichen Wertes

Der wirtschaftliche Wert kann an 4 grundlegenden Dingen gemessen werden: An seinem *Nutzen*, *Einfluss auf den Umsatz*, *Informationsrelvanz*, *Spezialisierung*. Die Datenerhebung kann, wie auch bei RENAISSANCE, durch Beobachtung, Fragebögen, Befragungen, aber auch durch Analyse der Dokumentationen.

8.3.2 Messen der Modularität

Für die Messung der Modularität eines Systems sollte eine ausführliche Dokumentation der einzelnen Komponenten vorhanden sein. Man wird, wenn möglich, Befragung der Programmierer und Entwickler vornehmen und auch das Software Design analysieren, sofern es dokumentiert ist. Ansonsten empfiehlt es sich den Code zu analysieren um daraus auf das Software Design schließen zu können.

8.3.3 Messen des Alterungsprozess

Durch die ständige Neuerungen im Bereich von Hard-/Software kommt es dazu, das ein System überholte Technologien einsetzt. Um ein System dahingehend zu analysieren, untersucht man die gesamte Infrastruktur (Hardware) des Systems, schaut sich Benutzerhandbücher und Dokumentationen an, holt sich Expertenmeinungen ein und schaut punktuell in den Source Code.

8.3.4 Messen des Verschleiß

Die ständigen Weiterentwicklungen an einem System haben auch negative Seiten. So kommt es, das durch ständige Neuerungen auch neue Fehler mit ins System gelangen. Zumeist wächst das System mit der Zeit und die Performance sinkt. Neuerungen werden meist gar nicht oder nur unzulänglich dokumentiert. Um einen solchen 'Verschleiß' zu messen benötigt man am besten den Source Code von verschieden Versionen die im Laufe der Zeit durch Weiterentwicklungen entstanden sind um mit Metriken einzelne Versionen bewerten zu können um sich ein Bild über den Entwicklungsverlauf machen zu können. Auch sind Testergebnisse der Performance der einzelnen Versionen, die möglicherweise über die Zeit gesammelt wurden, hilfreich, wie auch Problemberichte und Änderungsberichte um feststellen zu können in welchen Intervallen Fehler bzw. Änderungen auftreten.

8.3.5 Urteilsbildung

Nachdem man nun alles an Daten gesammelt hat, kann man nun entscheiden in welcher der vier Phase sich die Software befindet. Bewertet man die gemessenen Attribute jeweils mit „hoch“ oder „niedrig“, so kann man zu folgendem Urteil kommen:
Ein System das einen hohen wirtschaftlichen Wert hat aber dessen Alterungsprozess als niedrig eingestuft wird, ist in der Phase der „einfachen rückwirkenden und präventiven Wartung“ anzusiedeln. Sollte aber der Alterungsprozess als „hoch“ eingestuft werden so hängt die Zuordnung der Phase von den anderen Variablen ab. Sind wirtschaftlicher Wert und Modularität hoch und der Verschleiß gering, so stufen wir das System in die Phase „komplexere adaptive Wartung“ ein. Andernfalls, wenn sowohl

wirtschaftlicher Wert als auch Modularität gering sind der Verschleiß dagegen aber hoch, dann ist das System ein Kandidat für den Austausch.

8.3.6 Bewertung und Vergleich

Auch wenn sich beide vorgestellten Methoden ähneln so gibt es doch Unterschiede. Während man bei der RENAISSANCE-Methode das Legacy System erst aus einer groben Sicht betrachten um dann immer tiefer in die entsprechenden Details einzublicken, versucht man bei der „Legacy System Assessment"-Methode verstärkt auf den Alterungsprozess und den Verschleiß des Systems im Gesamten zu achten.
Als Gemeinsamkeit fällt auf das beide den wirtschaftlichen Wert des Systems für die Gesamtbeurteilung in Betracht ziehen. Es ist ein wichtiger Aspekt für die Bewertung von Legacy Systemen der das Verständnis für die Geschäftsprozesse bringt. Daher ist die Analyse des wirtschaftlichen Wertes des Systems in beiden Methoden auch der Beginn der Analyse. Darauf aufbauend kann man dann das System analysieren, und das mit verschieden Blickpunkten.

8.4 Zusammenfassung

Legacy Systeme sind altgediente Großsysteme, deren Bewertung und Begutachtung recht komplex ist. Es gibt dafür einige Bewertungsmethoden, von welchen wir nun zwei Methoden zur Bewertung von Altsystemen kennengelernt, und feststellen müssen, dass es dabei sehr um das Verstehen des Systems, aber auch des Unternehmens und seiner Geschäftsprozesse, geht. Man benötigt verschiedene Experten die Fachwissen aus den Bereichen des Unternehmens und des Systems mitbringen um zu einer Entscheidung zu kommen, wie man mit dem System weiter verfahren soll. Es bedarf einige Zeit um alle Vorbereitungen zu treffen, um ein Altsystem komplett zu analysieren. Eine Bewertung muss auf ein Altsystem maßgeschneidert sein und kann nicht generell für alle Systeme gelten.
Das RENAISSANCE-Projekt setzt bei der Bewertung des Altsystems auf die Bewertung der Infrastruktur, und der Applikation im Einzelnen. Während die „Legacy System Assessment"-Methode ihren Schwerpunkt auf das Messen von Modularität, Alterungsprozess und Verschleiß des Systems setzen. Beide haben jedoch eine Gemeinsamkeit, nämlich das Betrachten des wirtschaftlichen Wertes des Systems für das Unternehmen, welches das Altsystem einsetzt.
Beide Methoden sind eine Leitlinie zur Analyse eines Systems. Bei nicht ganz eindeutigen Ergebnissen, sollte man seine Messung nochmals analysieren und an den entsprechenden Stellen detailliertere Messung durchführen, um eine genaue Einschätzung des Systems und Entscheidung über die Zukunft selbiges treffen zu können.

Literaturverzeichnis

[CC97] G. Canfora and A. Cimitile. A reference life-cycle for legacy systems. *Workshop on Migration Strategies for Legacy Systems*, May 1997.

[CF01] Aniello Cimitile and Anna Rita Fasolino. *Legacy Systems Assessment to Support Decision Making*. July 09 2001.

[RSW98] Jan Ransom, Ian Sommerville, and Ian Warren. *A Method for Assessing Legacy Systems for Evolution*. 1998.

[Sne91] H. M. Sneed. Economics of software re-engineering. *Journal of Software Maintenance: Research and Practice*, 3(3):163–182, September 1991.

Kapitel 9

Ein Kostenberechnungsmodell für Softwarewartung und Softwareevolution

Arno Faßbender

Inhaltsverzeichnis

9.1 Einleitung

In dieser Seminarausarbeitung soll ein Kostenmodell für Softwarewartung und -evolution basierend auf einer Trennung von variablen und fixen Kosten vorgestellt werden. Schon seit längerer Zeit besteht ein Problem bei der Unterscheidung zwischen den Wartungsarbeiten, die durch eine übliche Wartungsgebühr abgedeckt werden und solchen die dem Kunden gesondert in Rechnung gestellt werden müssen. Eine klare Unterscheidung zwischen diesen zwei Arten von Kosten ist bei jeder Wartungsmaßnahme erforderlich um die Kostenentwicklung unter Kontrolle zu halten. Außerdem konzentrieren sich viele Berechnungsmodelle auf die Eigenschaften des reinen Softwaresystems. Im weiteren soll ein Überblick über ein Berechnungsmodell gegeben werden, dass die Trennung von Wartung und Evolution ermöglicht und gleichzeitig weitere Faktoren, wie z.B. Eigenschaften des Wartungsprozess mit in die Berechnung einbezieht. Gleichzeitig sollen die zu seiner Anwendung nötigen Informationen und deren Gewinnung besprochen, sowie ein Vergleich mit etablierten Verfahren zur Kostenberechnung gezogen werden.

9.2 Klassische Berechnung von Softwarewartungs- und Softwareevolutionskosten

Es existieren viele Modelle zur Berechnung von Softwareentwicklungskosten, wie z.B. *COCOMO* oder *Function-Point*. Innerhalb dieser Modelle wird Softwarewartung und Softwareevolution als Weiterführung der Entwicklung betrachtet. Entsprechend stellen diese Modelle keine speziellen Berechnungsmethoden für Wartung und Evolution zur Verfügung. Die zu erwartenden Kosten nachfolgender Versionen werden auf Grundlage der Eckdaten vorangegangener Versionen berechnet. Des weiteren findet bei der Kostenberechnung keine explizite Unterscheidung zwischen Wartung und Evolution statt. Der Anteil des Aufwands für Wartung wird allgemein, ausgehend von Erfahrungswerten, mit 38% des entstehenden Gesamtaufwands nach der Abnahme- und Einführungsphase angegeben, der für Evolution mit 62% [Bal01].

9.2.1 Berechnung von Softwarewartungs- und Softwareevolutionskosten mit *Function Point*

Die *Function Point*-Methode geht davon aus, dass der Aufwand zur Erstellung eines neuen Projekts vom Umfang und vom Schwierigkeitsgrad das Produkts abhängt. Der Umfang wird hierbei nicht wie bei anderen Methoden in *LOC* (*Lines OfCode*) ausgedrückt, sondern aus den Produktanforderungen ermittelt. Jede Produktanforderung wird einer von fünf Kategorien zugeordnet:

- Eingabedaten

- Abfragen
- Ausgaben
- Datenbestände
- Referenzdaten

Anschließend wird jede Produktanforderung unter Zuhilfenahme von Tabellen, Richtlinien und Beispielen in eine der Klassen *einfach*, *mittel* oder *komplex* eingeordnet. Je nach Kategorie und Klasse erhält jede Produktanforderung eine Gewichtung. Anschließend wird über alle gewichteten Produktanforderungen aufsummiert. Die endgültigen *Function Points* erhält man, indem man sieben Faktoren, die einen wesentlichen Einfluss auf die Entwicklung haben, entsprechend ihrer Einwirkung nach vorgegebenen Richtlinien mit Zahlen bewertet. Diese Bewertungsfaktoren werden dann aufsummiert und mit der Summe der gewichteten Produktanforderungen verrechnet, so dass sich zwischen 70 und 130% der ursprünglich berechneten Punkte ergeben. diese Spanne wird mit empirisch gewonnen Untersuchungsergebnissen begründet. Mit Hilfe dieser Gesamtpunkte und einer entsprechenden Tabelle mit Wertepaaren von *Function Points* und Personenmonaten wird dann ein Wert für den zu erwartenden Aufwand gewonnen. Diesen Schritt kann man nur durchführen, wenn man vorher für eine größere Anzahl abgeschlossener Entwicklungen aus dem gleichen Umfeld die bewerteten *Function Points* ermittelt hat [Bal01].

Zur Berechnung des zu erwartenden Aufwands für Softwarewartung und -evolution werden die gewichteten Produktanforderungen nach folgender Formel modifiziert:

$$EFP = (ADD + CHG + CFP) \cdot VAFa + (DEL \cdot VAFb)$$

Hierbei bezeichnet *EFP* die *Function Points* für die Produkterweiterung, *ADD* die hinzugefügten, *CHG* die geänderten, *DEL* die entfernten *Function Points* und *CFP* die unangepassten *Function Points*, die durch den Umbau entstehen. *VAFa* und *VAFb* sind Anpassungsfaktoren, die für das System vor und nach dem Umbau gelten sollen [ASKK03]. Man beachte, dass die individuelle Produktivität nur durch Verwendung geeigneter Wertepaare von *Function Points* und Personenmonaten ausgedrückt wird und eine explizite Unterscheidung zwischen Softwarewartung und -evolution nicht gegeben ist.

9.2.2 Berechnung von Softwarewartungs- und Softwareevolutionskosten mit *COCOMO-II*

Neben der *Function Point*-Methode ist das *COCOMO-II*-Schätzverfahren in der Industrie weit verbreitet. Aufbauend auf den Erfahrungen des 1981 vorgestellten *COCOMO*-Verfahrens (*Co*nstructive *Co*st *Mo*del) wurde 1995 *COCOMO-II* vorgestellt, das ein iteratives Vorgehen und die objektorientierte Entwicklung unterstützt. Als Eingabe dienen *KLOC* (*Kilo Source Lines Of Code*), die über einige Formeln mit 22 Einflussfaktoren verrechnet werden. Es wird keine explizite firmenspezifische Produktivitätsrate

verwendet, um den Gesamtaufwand zu ermitteln. Die Produktivitätsrate wird indirekt durch die Einschätzung der entsprechenden Einflussfaktoren modelliert.

Diese Einflussfaktoren decken ein wesentlich breiteres Spektrum an Einflüssen als bei der *Function Point*-Methode. Sie sind unterschiedlich gewichtet und wirken sich stark auf das Schätzergebnis aus. Die Fähigkeiten der Analytiker und Programmierer zwischen schlechtester und bester Bewertung wirken sich beispielsweise mit ca. 300% auf den Gesamtaufwand aus. Folgende Faktoren werden berücksichtigt:

Produktbezogen:

- Komplexität
- Erforderliche Zuverlässigkeit
- Erforderliche Wiederverwendbarkeit
- Datenbankgröße
- Anforderungen an die Dokumentation

Personalbezogen:

- Anwendungserfahrung des Personals
- Erfahrung mit Sprachen und Werkzeugen
- Qualität der Analytiker
- Qualität der Programmierer
- Kontinuität der Mitarbeiter
- Erfahrungen mit der Plattform

Plattformbezogen:

- Stabilität der Plattform
- Randbedingungen bezogen auf die Ausführungszeit
- Randbedingungen bezogen auf den Arbeitsspeicher

Projektbezogen:

- Zulässige Entwicklungszeit
- Entwicklung an mehreren Standorten

- Einsatz von Werkzeugen

Skalierungsfaktoren:

- Erstmaligkeit
- Entwicklungsflexibilität
- Zusammenhalt der Projektbeteiligten
- Klärung Architektur/Risiken
- Reife des Prozesses

Bei der Aufwandsberechnung für Wartungsaufgaben wird auch hier nicht explizit zwischen Softwarewartung und -evolution unterschieden. Es findet lediglich eine Anpassung der Eingabegröße in *KLOC* statt, gemäß folgender Formel:

$$Size = BaseCodeSize \cdot MCF \cdot MAF$$

BaseCodeSize beschreibt dabei die Größe des vorhandenen Systems, *MCF* den *Maintenance Change Factor* und *MAF* den *Maintenance Change Factor*. Letzterer ergibt sich zu:

$$MCF = \frac{SizeAdded + SizeModified}{BaseCodeSize},$$

also dem Verhältnis von betroffenem Code zur Gesamtgröße des vorhandenen Systems. Der *MAF* wird eingesetzt um die Eingabegröße an Faktoren wie Verständnis der Software und Gewöhnungseffekte anzupassen.

$$MAF = 1 + (\frac{SU}{100} \cdot UNFM)$$

SU bezeichnet hierbei den *Software Understanding Factor*, welcher Werte zwischen 10 und 50 annehmen kann und *UNFM* den *Programmer Unfamiliarity Factor* mit einem Wertebereich zwischen 0,0 und 1,0. Diese Faktoren können die angepasste Größe also um bis zu 50% erhöhen. Beide Faktoren sind vom Anwender der *COCOMO-II*-Methode unter Zuhilfenahme gewisser Richtlinien selbst einzuschätzen [Bal01][COC].

9.3 Gründe für ein eigenes Kostenberechnungsmodell für Softwarewartung und Softwareevolution

Praktische Studien im Bereich der Kostenberechnung haben gezeigt, dass Wartungskosten nicht im Zusammenhang mit den Entwicklungskosten stehen müssen. Die Wartungskosten für Systeme, die aus vorgefertigten Komponenten zusammengestellt werden, können die Wartungskosten von kompletten Neuentwicklungen um bis zu 40 % übersteigen, obwohl letztere in ihren Entwicklungskosten weit höher liegen [RBC03].

Auch hat sich gezeigt, dass bei Systemen mit vergleichsweise geringen Entwicklungskosten mit höheren Wartungskosten zu rechnen ist [Sne97]. Die Kostenberechnung für Wartung und Evolution auf die Entwicklungskosten zurückzuführen, führt also im Allgemeinen nicht zu zutreffenden Ergebnissen.

Des weiteren existieren Ansätze zur Kostenberechnung auf Basis der Charakteristiken von Software. Diese versuchen die Komplexität des Codes, die Kopplung oder die Qualität der Softwarearchitektur zur Kostenberechnung heranzuziehen. Diese Ansätze sind nützlich um Kostenfaktoren zu identifizieren, sind für sich genommen in ihrer Aussagekraft aber zu begrenzt, da sie sich auf einzelne Charakteristiken der Software beschränken und wichtige Faktoren außerhalb des Systems unberücksichtigt lassen.

Die Berechnung von Softwarewartungskosten ist ein Problem mit vielen Dimensionen und die Software an sich ist ist nur eine davon. Weitere Einflussfaktoren liegen im Wartungsprozess selbst, der verwendeten Umgebung, dem Personal und den verwendeten Werkzeugen. Diese Faktoren haben einen genauso großen Einfluss auf die zu erwartenden Wartungskosten, wie die Software selbst. Eine Vorhersage der Wartungskosten allein basierend auf den Eigenschaften der Software kann nur in einer Umgebung in der alle anderen Einflussfaktoren neutralisiert werden können zu den gewünschten Ergebnissen führen. Für reale Projekte müssen auch die oben genannten Einflüsse berücksichtigt werden.

9.4 Vergleich von Softwarewartung und Softwareevolution

Die Uneinigkeit darüber, was genau Wartung von Weiterentwicklung unterscheidet, reicht bis in die Anfänge der elektronischen Datenverarbeitung zurück. In einer frühen Veröffentlichung des American General Accounting Office - GAO - wurde Wartung als Gesamtheit aller Änderungsaktivitäten nach Auslieferung eines Softwareprodukts definiert [Mar83]. Dies hatte den Vorteil, dass die entstanden Kosten eindeutig bestimmten Budgets zugeordnet werden konnten. Alle Kosten vor dem Auslieferungstermin ware Entwicklungskosten, die dem Entwicklungsbudget zugeordnet werden konnten, alle danach Wartungskosten, die durch eine jährliche Wartungsgebühr abgedeckt werde mussten.

Während der 80er und frühen 90er Jahre wurde diese Definition von Entwicklung und Wartung allgemein anerkannt. Das in dieser Zeit veröffentlichte *IEEE Lebenszyklusmodell* sah Wartung als die letzte Phase im Lebenszyklus einer Software, in der Fehler behoben und kleinere Verbesserungen gemacht wurden [BD91]. Wie sich jedoch später herausstellte, belief sich der Aufwand für diese Phase im Allgemeinen auf ein doppelt so großes Aufwandsvolumen, wie für alle vorherigen Phasen zusammengenommen [Boe88].

Das Problem dieser Definition von Entwicklung und Wartung liegt darin, dass Industriemanager meist eine ganz andere Vorstellung von dem Begriff Wartung haben. Sie betrachten die Aktivitäten in diesen Bereichen nach dem Kriterium, welche Aktivität

welchem Budget zugeordnet werden muss. Reverse Engineering, Erneuerungen, Migrationen, Integration und funktionale Erweiterungen sind typische Projekte, die einer eigenen Finanzierung bedürfen. Fehlerbeseitigung, Optimierung und Anpassungen wiederum sind Aktivitäten, die aus einer jährlichen Wartungsgebühr finanziert werden müssen.

Ähnlich wie bei der Softwarewartung, beschäftigt sich Softwareevolution mit der Frage, was mit einem System passiert, nachdem es ausgeliefert wurde, allerdings mit einem anderen Schwerpunkt. Fehlerbehebung wird als selbstverständlich und Teil eines kontinuierlichen Verbesserungsprozesses angesehen. Ebenso werden Systemwechsel als Teil diese Prozesses angesehen. Die Sicht bezieht sich also hauptsächlich auf Software als ein wachsendes System, was sich auch in ihrer Klassifizierung nach Wachstumsraten widerspiegelt. Systeme mit geringen Wachstumsraten, wie z.B. Büroanwendungen, werden als statisch, Systeme mit hohen Wachstumsraten, wie z.B. Web-Applikationen, als dynamisch bezeichnet [BL76]. Im Rahmen evolutionärer Entwicklung wird ein System Release für Release entwickelt, was voraussetzt, dass ein Release durch die offenen Änderungsanfragen und das Releaseintervall bestimmt wird.

Aus betriebswirtschaftlicher Sicht ist evolutionäre Entwicklung ein Reise ins Unbekannte. Niemand kann genau sagen, wie das endgültige Ergebnis aussehen wird und wie hoch die Kosten sein werden. Dies macht eine Planung, insbesondere im Bereich der Finanzierung natürlich extrem schwierig. Verständlicher Weise haben Manager ein zwiespältiges Verhältnis zu evolutionären Entwicklungsmethoden, wie z.B. *Extreme Programming*. Eine klare Abgrenzung zwischen Entwicklung und Wartung macht für sie die Berechnung und Kontrolle von Kosten einfacher.

9.5 Abgrenzung von Softwarewartung und Softwareevolution durch das *Zustandsmodell*

Durch Einführung des *Zustandsmodells* haben Rajlich und Bennet eine Kompromisslösung zur Aufhebung des Konflikts zwischen Softwarewartung und -evolution gefunden [BR01]. Ausgehend von diesem Modell existieren fünf Phasen im Lebenszyklus eines Softwaresystems:

- Prototypphase
  Erstellung eines Prototyps und einer Durchführbarkeitsstudie
- Entwicklungsphase
  Entwicklung und Erstellung des Systems
- Evolutionsphase
  Schwerpunkt: Weiterentwicklung des System durch funktionale Erweiterungen
- Wartungsphase
  Schwerpunkt: Fehlerkorrektur und kleinere Anpassungen

- Rückzugsphase
  Stilllegung des Systems

In diesem Modell stellen Wartung und Evolution keine konkurrierenden Aspekte einer einzelnen Phase dar, sondern klar von einander abgegrenzte Zeitabschnitte im Lebenszyklus.

Die Prototypphase stellt eigentlich eine Durchführbarkeitsstudie bezüglich der Kosten und des zu erwartenden Nutzen der Software dar. In dieser Phase bietet sich *Extreme Programming* an. Das Budget sowie der der zur Verfügung stehende zeitliche Rahmen werden im Vorfeld festgelegt.

Ausgehend von den Erfahrungen, die in der Prototypphase gemacht wurden, ist es möglich die Kosten und den benötigten Zeitaufwand für die Entwicklungsphase relativ genau voraus zu berechnen. Hierfür können konventionelle Berechnungsmethoden Anwendung finden, da das System bereits definiert wurde. Auch die Entwicklungsphase hat eine feste Deadline, nach der das System frei gegeben wird. Alle nachfolgenden Aktivitäten fallen in die Evolutionsphase.

Die anfallenden Kosten für die gesamte Evolutionsphase können nicht im Vorfeld berechnet werden, wohl aber die für jedes einzelne Release. Für jeden dieser Releases wird ein gewisser Anteil des zu erwartenden Aufwands der Korrektur des vorherigen Release zugeschrieben, ein weiterer der Entwicklung des nachfolgenden Release. Die Entwicklungsaufgaben können in funktionaler Erweiterung, Erneuerung oder kleineren Anpassungen liegen. Falls letztere überwiegen, ist dies ein Zeichen dafür, dass sich die Evolutionsphase ihrem Ende nähert.

Über die Zeit nimmt die Wachstumsrate des Systems ab. Dies kann unterschiedliche Gründe haben, z.B. wachsende Kosten bei abnehmendem Nutzen. Zu einem bestimmten Zeitpunkt werden weitere Investitionen in das System unwirtschaftlich und die Entwicklung wird eingestellt. Hier beginnt die Wartungsphase, in der nur noch Fehlerkorrekturen und unvermeidliche Änderungen durchgeführt werden.

Letztendlich wird das System irgendwann veraltet sein oder der Unterhalt des Systems zu teuer. Die kann an wachsender Komplexität bei gleichzeitig abnehmender Qualität liegen, meist liegt der Grund aber in einem Verlust an Wissen über das System. Oft ist das Personal, dass das System entwickelt hat und über das nötige Wissen über den Arbeitsbereich der Software verfügt, nicht mehr verfügbar. Es ist auch möglich, dass das Arbeitsumfeld des Systems nicht länger zur Verfügung steht. Manchmal ist es möglich die Software in ein neues Arbeitsumfeld zu portieren, in anderen Fällen können nur die vorhandenen Daten gesichert werden und die Software wird ersetzt. In beiden Fällen wird das System stillgelegt [Sne04a].

Das hier im weiteren beschriebene Modell zur Kostenberechnung ermöglicht eine klare finanzielle Trennung dieser Phasen.

9.6 Das *MainCost*-Modell

Das hier vorgestellte *MainCost*-Modell von Harry M. Sneed basiert auf dem oben genannten Zustandsmodell. Es geht von einer Evolutionsphase aus, in der Weiterentwicklungen und Wartungsaufgaben parallel durchgeführt werden und einer klar dazu abgegrenzten Wartungsphase, in der ausschließlich Wartungsaufgaben durchgeführt werden. Ziel des Modells ist es, sowohl die Kosten für Wartung als auch Evolution plan- und kontrollierbar zu machen [Sne04a].

Prinzipiell existieren zwei Ansätze, um Kosten für Softwareänderungen zu berechnen. Einerseits durch eine Analyse der nötigen Änderungen auf Basis einzelnen anstehenden Aufgaben. Hierbei werden die von den Änderungen betroffenen Komponenten identifiziert und anschließend der Umfang der Änderungen in den einzelnen Komponenten berechnet. Der für einen Arbeitsbereich anfallende Änderungsumfang wird dann an die entsprechenden Werte für Qualität und Komplexität des Bereichs angepasst. Dieser Ansatz einer Analyse auf Mikro-Ebene wird ausführlich von Sneed beschrieben [Sne95].

Bei dem hier beschriebenen *MainCost*-Modell handelt es sich um einen Ansatz auf Makro-Ebene. Hierbei werden auf Grundlage von Daten voran gegangener Releases die Kosten für die Korrektur und Anpassung des nächsten geplanten Release vorausberechnet. Zur Durchführung einer Berechnung auf Grundlage diese Modells sind mehrere Parameter notwendig, die sich aus folgenden Analysemethoden ergeben:

- Statische Softwareanalyse
  Liefert Metriken zu den Eigenschaften der Software
- Dynamische Softwareanalyse
  Liefert Daten zu den Testfällen
- Fehleranalyse
  Liefert Daten zu auftretenden Fehlern
- Produktivitätsanalyse
  Liefert Daten zum Aufwand für Änderungsaktivitäten

Die statische Softwareanalyse liefert Metriken bezüglich der Größe, Komplexität und Qualität des Codes, der Spezifikation sowie der Testfälle. Aus der dynamischen Softwareanalyse erhält man Daten bezüglich der Codeabdeckung der Testfälle und die Fehleranalyse gibt Anzahl und Art der auftretenden Fehler sowie die Fehlerdichte an. Die Produktivitätsanalyse wiederum liefert Daten bezüglich des Aufwands für Fehlerberichte, Änderungsanfragen, Reengeniering- und Entwicklungsaufgaben. Ohne diese Informationen ist es kaum möglich die Wartungskosten im Voraus zu berechnen, da die Software allein für eine Modellbildung nicht ausreichend ist.

Mit Hilfe der aus den oben aufgeführten Analysemethoden gewonnen Daten lässt sich der Aufwand für folgende Wartungs- und Entwicklungsaufgaben berechnen:

- Fehlerkorrektur
  Behebung von syntaktischen und logischen Fehlern
- Routineänderungen
  Planbare Änderungen von geringer Dringlichkeit
- Funktionale Erweiterungen
  Erweiterung des Systems um neue Funktionen und Anwendungsbereiche
- Technische Erneuerung
  Optimierung der Architektur und Effizienzsteigerung

Das Modell beruht auf der Annahme, dass kein universeller Ansatz zur Berechnung von Wartungs- und Evolutionskosten existiert, sondern jede Aktivität ihre eigene Analysemethode erfordert. Ziel ist es, die verschiedenen Arten von Wartungs- und Evolutionsaufgaben zu identifizieren und einen entsprechenden Ansatz für jede einzelne Aufgabe zu finden.

9.6.1 Die Analysemethoden

Die oben genannten Methoden zur Analyse der Software sollen hier im weiteren genauer vorgestellt werden. Die mit ihnen gewonnen Ergebnisse bilden die Grundlage zur Berechnung der zu erwartenden Kosten für Softwarewartung und -evolution.

9.6.1.1 Statische Softwareanalyse

Techniken zur statischen Softwareanalyse haben in der Literatur bereits viel Aufmerksamkeit erfahren, insbesondere Methoden der Codeanalyse [HW02]. Allerdings besteht ein Softwaresystem aus mehr als nur Code. Es umfasst zusätzlich die Anforderungsspezifikation, Entwurfsdokumente, Testfälle und Datenbankentwurf. In so weit, als auch diese Artefakte gewartet und weiterentwickelt werden müssen, sollte auch sie Gegenstand der statischen Analyse sein. Um anfallende Kosten berechnen zu können, muss die statische Analyse die Größe, Komplexität und Qualität dieser Softwareartefakte messen. Es gibt fünf verschiedene Arten von Artefakten, die zu analysieren sind:

- Anforderungsspezifikation
- Entwurfsdokumente
- Sourcecode
- Datenbankentwurf
- Testfälle

Bei der statischen Analyse der Anforderungsspezifikation werden die Anzahl der Testfälle, Funktionen, Geschäftspläne, Daten, Attribute, Benutzerschnittstellen und Systemschnittstellen gezählt um eine Quantität des Entwurfs zu erhalten. Die Komplexität des Entwurfs leitet sich aus den Beziehungen zwischen diesen logischen Einheiten ab. Die Qualität der Anforderungsspezifikation hängt von ihrer Vollständigkeit, Konsistenz und formalen Korrektheit ab.

Die statische Analyse der Entwurfsdokumente bezieht sich auf die Anzahl der Diagramme, deren Verbindungen zu einander und ihrer formalen Korrektheit. UML Diagramme werden in einer Datenbank festgehalten, von wo aus sie automatisch analysiert werden können. Falls es keinen direkten Zugriff auf diese Datenbank gibt, können die exportierten Dokumente analysiert werden. Seit Einführung der modellbasierten Entwicklung gewinnt diese Form des Messens von Eigenschaften eines Entwurfs durch Metriken für dessen Größe, Komplexität und Qualität zunehmend an Bedeutung [Sel03].

Die Sourcecodeanalyse liefert eine Reihe von Metriken, die zu einem einheitlichen Maß für Komplexität und Qualität und einem oder mehreren Maßen für die Größe, wie z.B. *Function Points*, *Object Points*, *Statements* oder *LOC*, zusammengefasst werden müssen. Wichtig ist hierbei, dass die Maße für Qualität und Komplexität auf einen Bereich zwischen 0 und 1 normalisiert werden. Diese Normalisierung erlaubt eine Verschmelzung der Metriken für den Code mit denen der anderen Artefakte und eine Anpassung der Größe der Software [Dro95].

Die statische Analyse des Datenbankentwurfs liefert die Anzahl der Tabellen und Attribute, sowie deren Beziehungen untereinander. Das Ergebnis besteht aus Metriken für Größe, Komplexität und Qualität des Datenbankentwurfs.

Der letzte Teil der statischen Analyse bezieht sich auf die Testfälle. Diese können in unterschiedlichen Formaten vorhanden sein, wie z.B. Exceltabellen, CSV Dateien oder in relationalen Datenbanken. In jedem Fall kann aber ihre Anzahl und die ihrer Attribute bestimmt werden, ihre Komplexität gemessen und ihre Qualität auf Grundlage des gemessenen Grades ihrer Automation bestimmt werden [Sne04b].

9.6.1.2 Dynamische Softwareanalyse

Das Ziel der dynamischen Analyse besteht darin, die Abdeckung jedes einzelnen Release durch die Testfälle zu bestimmen. Zu diesem Zweck muss entweder der Sourcecode oder der Bytecode herangezogen werden oder die Analyse findet auf einer höheren Ebene statt. Zu diesen können die Ebene der Methoden oder die der Verzweigungen gehören. Die gewählte Ebene sollte der zu testenden Software angemessen sein. Des weiteren kann es nötig sein, den Code in Abschnitte zu unterteilen um Teile auszuschließen, die von einer spezifischen Anwendung nicht benutzt werden [MB03].

Unabhängig davon werden zur Testzeit Profiltabellen erstellt, um festzuhalten, welche Schlüsselstellen wie oft durchlaufen wurden. Im Allgemeinen wird für jede Kompo-

nente bzw. jedes Modul eine eigene Tabelle geführt. Nach Durchführung der Tests werden die Tabellen zusammengefasst, um eine Gesamtabdeckungsrate für das gesamte System zu erhalten. Diese Rate zeigt, welcher Anteil des relevanten Codes durch die Testfälle abgedeckt wurde. Mit Hilfe dieser Rate, der durchschnittlichen Fehlerrate und der Größe des neuen Release kann die erwartete Fehlerrate berechnet werden.

9.6.1.3 Fehleranalyse

Während der Testphase werden alle gefundenen Fehler festgehalten. Diese Fehler werden dann nach Fehlerquelle, Fehlerart und Schwere des Fehlers kategorisiert. Dazu bietet sich ein Fehlerverfolgungssystem mit angeschlossener Datenbank an.

Zwischen den einzelnen Releases werden außerdem Fehler durch die Anwender gemeldet, die in der selben Datenbank wie die durch Tests entdeckten Fehler festgehalten werden, aber separat gekennzeichnet werden. Zusätzlich wird vermerkt, in welcher Version diese Fehler auftraten. Dadurch ist es möglich nachzuvollziehen, welche Fehler in welcher Version auftraten [KMFA00].

Am Ende einer Releasephase wird die Fehlerdatenbank analysiert und die Fehler über Art und Release aufsummiert. Danach werden diese in Verhältnis zur Größe des Sourcecode gesetzt, um eine Fehlerdichte pro Anweisung, *Function Point* und *Object Point* zu erhalten. Diese Werte dienen dann zur Berechnung der Fehlerrate des nächsten Release.

9.6.1.4 Produktivitätsanalyse

Der wichtigste Faktor zur Vorausberechnung von zu erwartendem Aufwand ist der Produktivitätsfaktor, welcher an die lokalen Gegebenheiten angepasst werden muss. Ein weit verbreiteter Fehler in der Berechnung von Softwarekosten liegt in der Übernahme eines fremden Produktivitätsindex, wie z.B. der IBM *Function Point*-Produktivitätsrate, und dessen Anwendung auf eine Kostenberechnung. Die Wahrscheinlichkeit zu Aussagekräftigen Ergebnissen zu kommen ist minimal, da die Unterschiede zwischen den verwendeten Entwicklungsumgebungen, Werkzeugen und Personal meist viel zu groß sind. Im besten Fall erreicht man eine Annäherung an das gewünschte Ergebnis [Sne04a].

Auch sollte unter keinen Umständen die Produktivität in der Entwicklung auf die in Wartung und Evolution übertragen werden. Es ergeben sich sogar gravierende Unterschiede zwischen Wartung und Evolution. Im Allgemeinen erfordert es mehr Aufwand, einen Fehler zu finden und zu beheben, als eine neue Methode in eine bereits existierende Klasse einzufügen, oder eine neue Klasse in eine bereits existierende Komponente. Dies führt zu einer eigenen Produktivitätsrate für Reengineering-Aufgaben. Als Ergebnis dieser Beobachtung ist es notwendig unterschiedliche Produktivitätsraten für unterschiedliche Aufgaben, wie Fehlerbeseitigung, Änderungen, funktionale Erweiterungen und Erneuerungen, festzuhalten.

Es ist weiterhin Aufgabe der organisatorischen Strukturen genauestens über den für die einzelnen Aufgaben aufgebrachten Aufwand Buch zu führen. Dies erfolgt über ein Zeiterfassungssystem, in dem die Mitarbeiter ihre Arbeitszeit den entsprechenden Wartungs- und Evolutionsaufgaben zuordnen müssen. Dadurch kann dieser in Bezug zu den erbrachten Leistungen kategorisiert nach Codevolumen, *Function Points*. *Object Points* oder anderen Größenfaktoren gesetzt werden. Dadurch ergibt sich eine Produktivitätsrate für jeden einzelnen Faktor.

9.6.2 Berechnug von Softwarewartungskosten

Die oben vorgestellten Analysemethoden liefern die notwendigen Daten um die Kosten für Softwarewartung und Softwareevolution voraus zu berechnen. Hier soll zunächst die Softwarewartung betrachtet werden, welche in Fehlerbeseitigung und Adaption unterteilt wird.

9.6.2.1 Berechnung der Fehlerbeseitigungskosten

Die Kosten zur Beseitigung schwerwiegender Fehler ergeben sich aus der den Kosten zur Fehlerbeseitigung aus vorangegangenen Releases. Dazu werden folgende Daten benötigt:

- Anzahl schwerwiegender Fehler im letzten Release
- Durchschittlicher Aufwand zur Fehlerbeseitigung im letzten Release
- Angepasste Größe des letzten Release
- Angepasste Größe des folgenden Release
- Testfallabdeckung des letzten Release
- Testfallabdeckung des nächsten Release

Die zu bestimmenden Größen sind die Anzahl der Fehler im nächsten Release und der durchschnittlich nötige Aufwand zu ihrer Beseitigung. Daraus ergibt sich der nötige Aufwand zur Fehlerbeseitigung. Die Anzahl der erwarteten Fehler ergibt sich aus der Anzahl der Fehler im letzten Release unter Berücksichtigung des Unterschieds in der Testfallabdeckung. Der nötige Aufwand zur Fehlerbeseitigung ergibt sich aus dem Aufwand pro Fehlerkorrektur im letzten Release unter Berücksichtigung des Unterschieds in Größe, Komplexität und Qualität zwischen letztem und nächstem Release.

Unter der Annahme, dass ein Zusammenhang zwischen der Testabdeckung und der Anzahl der nicht gefundenen Fehler besteht, wie z.B. Fehler die durch die Anwender gemeldet werden, ergibt sich die Fehlerdichte des letzten Release aus den gemeldeten Fehlern relativ zu dem nicht durch Testfälle abgedeckten Teil des Release.

$ErrorDensity = \frac{ErrorsReorted}{Size \cdot (1 - TestCoverage)}$

Die Anzahl der zu erwartenden verbleibenden Fehler nach dem Test des nächsten Release ergibt sich aus dem nicht durch Testfälle abgedeckten Teil des Release multipliziert mit der Fehlerdichte des letzten Release.

$NumberOfExpectedErrors = NewSize \cdot (1 - NewTestCoverage) \cdot ErrorDensity$

Im nächsten Schritt wird der Aufwand zur Beseitigung dieser Fehler berechnet. Dazu wird der benötigte Aufwand pro Fehler im letzten Release unter Berücksichtigung des Unterschieds in Größe, Komplexität und Qualität zwischen letztem und nächstem Release. Die Größe jedes einzelnen Release muss an die Komplexität und Qualität angepasst werden.

$AdjustedSize = Size \cdot \frac{Complexity}{Quality}$

Im letzten Schritt wird die Anzahl der erwarteten Fehler mit der dem durchschnittlichen Aufwand zur Fehlerbeseitigung multipliziert, um den zu zu erwartenden Gesamtaufwand zu bestimmen.

9.6.2.2 Berechnung der Adaptionskosten

Die Kosten zur Umsetzung von Änderungsanfragen werden auf ähnliche Weise wie die Kosten zur Fehlerbeseitigung berechnet, nur mit dem Unterschied, dass hier andere Parameter Anwendung finden. Folgende Parameter finden bei der Berechnung der Adaptionskosten Anwendung:

- Anzahl der Änderungsanfragen im letzten Release
  Wie oft wurden von den Kunden Änderungswünsche angefragt und umgesetzt
- Reduktionsrate der Änderungsanfragen
  Wie stark sind die Änderungsanfragen zurückgegangen
- Durchschnittlicher Aufwand pro Änderung im letzten Release Wie hoch war der Aufwand für eine im letzten Release umgesetzte Änderungsanfrage im Schnitt
- Angepasste Größe des letzten Release
  Nach Komplexität und Qualität gewichtete Größe des letzten Release (vgl. 9.6.2.1)
- Angepasste Größe des nächsten Release
  Nach Komplexität und Qualität gewichtete Größe des nächsten Release (vgl. 9.6.2.1)

Die grundsätzlichen Fragen sind die nach der Anzahl der Änderungsanfragen und welcher Aufwand im Schnitt nötig sein wird, um diese umzusetzen. Daraus ergibt sich der Gesamtaufwand zur Adaption des Systems. Die Anzahl der erwarteten Änderungsanfragen ergibt sich aus der Anzahl der Änderungsanfragen für das letzte Release unter

Berücksichtigung der Reduktionsrate der Änderungsanfragen. Der Aufwand um diese umzusetzen ergibt sich aus dem durchschnittlichen Aufwand pro Änderungsanfrage des letzten Release unter Berücksichtigung der Unterschiede in Größe, Komplexität und Qualität zwischen dem letzten und dem nächsten Release.

Die Reduktionsrate der Änderungsanfragen ergibt sich aus der durchschnittlichen Differenz der Änderungsanfragen seit dem ersten Release. Dabei kann sich eine ansteigende oder abfallende Wachstumsrate ergeben. Empirische Studien haben gezeigt, dass die Anzahl der Änderungsanfragen pro Release im Allgemeinen zuerst ansteigt und im weiteren Verlauf abfällt. Nach dem dritten oder vierten Release ist mit einer Abnahme um 10 % pro Release zu rechnen.

Die Daten über den durchschnittlichen Aufwand pro Änderungsanfrage im letzten Release können dem Zeiterfassungssystem entnommen werden (vgl. 9.6.1.4). Dieser liegt erfahrungsgemäß zwischen 2 und 4 Personentagen [Sne04a].

Zur Berechnung des erwarteten Aufwands für Änderungsanfragen im nächsten Release muss nun lediglich die erwartete Anzahl von Änderungsanfragen mit dem angepassten durchschnittlichen Aufwand pro Änderungsanfrage multipliziert werden.

9.6.3 Berechnung von Softwareevolutionskosten

Softwareevolution, wie sie hier definiert wird, besteht aus zwei Bereichen: funktionale Erweiterung und technische Optimierung. Diese müssen klar von einander getrennt betrachtet werden. Dem zufolge erfolgt auch die Berechnung der entsprechenden Kosten in separaten Teilschritten, die hier im Weiteren näher betrachtet werden sollen.

9.6.3.1 Berechnung von Kosten für funktionale Erweiterung

Die Kosten für funktionale Erweiterung werden in recht ähnlicher Weise wie die für Neuentwicklungen berechnet. Zuerst werden die Anforderungen analysiert und in entsprechende Metriken wie z.B. *Function Point* oder *Object Point* übertragen. Dieses Größenmaß wird dann an die Komplexität und Qualität des Systems angepasst und durch die Produktivitätsrate dividiert. Je nach Größenmaß ist es auch möglich, dass es durch bestimmte Einflussfaktoren modifiziert wird. Für die *Funktion Point*-Methode ergibt sich:

$$Effort = \frac{FunctionPoints \cdot \frac{Complexity}{Quality} \cdot InfluenceFactor}{ScaledProductivity}$$

Bei der *COCOMO-II*-Methode wird die angepasste Größe der Erweiterung in eine Gleichung eingesetzt, in der sie mit Faktoren für den Systemtyp, dem Produkt aus 16 verschiedenen Einflussfaktoren und einem Skalierungsfaktor multipliziert wird.

$$Effort = SystemType \cdot \frac{Size}{Productivity} \cdot ScalingFactor \cdot InfluenceFactor$$

mit

- $SystemType \in [0.5, 4.0]$
- $ScalingFactor \in [0.92, 1.23]$
- $InfluenceFactor = \prod_{i=1}^{16} Influence_i$

Dabei ist es wichtig verschiedene Methoden zur Bestimmung der Größe der Erweiterung zu verwenden und die verschiedenen angepassten Größen der Erweiterung in Aufwandsmaße basierend auf den vorhandenen Produktivitätsraten pro Größeneinheit umzurechnen. Die so erhaltenen verschiedenen Aufwandsmaße geben dann einen Überblick über den zu erwartenden Aufwand.

9.6.3.2 Berechnung von Kosten für technische Optimierung

Technische Optimierung wird hier die Gesamtheit aller Aktivitäten bezüglich technischer Perfektionierung des Systems inklusive Optimierungs- und Erneuerungsaufgaben. Falls diese vom begrenztem Umfang sind, werden sie in die normalen Aufgaben zur Erstellung eines neuen Release eingebunden. Sollten sie jedoch von von größerem Umfang sein, werden sie einem separat zu berechnenden Reengineering-Projekt zugewiesen. In beiden Fällen jedoch, müssen die zu erwartenden Kosten auf andere Art und Weise als bei einer funktionalen Erweiterung berechnet werden.

Der große Unterschied liegt darin, dass sich der Umfang einer funktionalen Erweiterung, ähnlich wie bei einer Neuentwicklung, aus der Anforderungsspezifikation ergibt, während sich der Aufwand einer technischen Optimierung aus dem bereits existierenden Code ableitet. Um den Umfang eines Reengineering-Projekts zu berechnen, ist es zuerst erforderlich alle zu ändernden Teile des Sourcecodes, inklusive der betroffenen Komponenten, Datenbankschemata und Benutzerschnittstellen zu identifizieren. Die Größe jedes betroffenen Bereichs muss an seine Qualität und Komplexität angepasst werden. Die gesamte Größe ergibt sich als die Summe aller angepassten Größen der betroffenen Bereiche. Als zu verwendende Metrik sollte hier entweder die Anzahl der Statements und Deklarationen, oder die Anzahl der unkommentierten *LOC* verwendet werden.

Wenn die angepasste Größe des betroffenen Sourcecode bekannt ist, kann auf diese die *COCOMO-II*-Methode angewendet werden, um basierend auf der Produktivität vorangegangener Reengineering-Projekte den zu erwartenden Aufwand zu berechnen. Größenmaße basierend auf der Anforderungsspezifikation, wie z.B. *Funktion Point*, sind sowohl für technische Optimierung als auch funktionale Erweiterungen ungeeignet, da Reengineering-Aufgaben sich direkt auf den Code beziehen und daher entsprechende Metriken erfordern. Im Falle eines Redesign allerdings, wäre die Verwendung einer entwurfsbezogenen Metrik wie *Object Point* angebrachter, die Klassen, Methoden, Attribute, Schnittstellen, Vererbungen und Assoziationen berücksichtigt [Sne04a] [Hug00].

Beim Ansetzen der Produktivitätsrate im Bereich Reengineering sollte die Produktivität jeder einzelnen betroffenen Abteilung oder Organisation herangezogen werden, da die Produktivität zwischen diesen sehr unterschiedlich sein kann. Dies gilt im Bereich Reengineering im Besonderen, da hier spezielles Wissen und spezielle Werkzeuge erforderlich sind. Die verwendeten Werkzeuge haben dabei einen größeren Einfluss auf die Produktivitätsrate als die eigentliche Produktivität im Bereich Entwicklung [Sne91].

9.7 Erfahrungen mit dem *MainCost*-Modell in der Praxis

Die hier beschriebenen Methoden fanden über fünf Jahre lang Anwendung, um sowohl die Wartungs- als auch die Evolutionskosten der Software *GEOS* zu berechnen. GEOS ist ein IT-Abwicklungs- und Verwaltungssystem von Wertpapieren mit Schwerpunkt auf Straight Through Processing (STP). Die Hauptkomponenten von GEOS (Verwaltung von Finanzinstrumenten und Aufträgen sowie Positionsführung) decken die gesamte Wertschöpfungskette im Wertpapiergeschäft - von der Wertpapierorder über Wertpapier-Clearing und Settlement bis zur Verarbeitung von Corporate Actions (z.B. Zahlung einer Dividende) - ab [GEO].

Während es in der Anfangsphase ein Releaseintervall von sechs Monaten angesetzt wurde, ergab sich nach zwei Jahren ein Releaseintervall von drei Monaten, woraus insgesamt 16 Releases resultierten. In der Berechnung der Kosten für die ersten vier Releases ergaben sich Abweichungen zwischen 25 und 40%, was hauptsächlich auf eine Unterschätzung der erwarteten Fehler und Änderungsanfragen zurückzuführen ist. Nach Ablauf der ersten vier Jahre wurden die Berechnungen für die Wartungskosten genauer, da die Fehlerrate von 350 Fehlern pro Release konstant blieb. Während der letzten drei Jahre blieb die Fehlerrate konstant bei 0,18 Fehlern pro Personentag. Die Anzahl der Änderungsanfragen konnte ebenso genauer vorausgesagt werden und stieg von 156 beim ersten Release auf 281 beim letzten Release.

Ein weiterer konstanter Faktor, der genaue Voraussagen ermöglichte, war der Aufwand für die Fehlerkorrektur. Dieser stieg in den gesamten 5 Jahren nur leicht von 2,6 Personentagen pro Fehler im ersten Jahr, auf 3,5 Personentagen pro Fehler im letzten Jahr. Der durchschnittliche Aufwand für Änderungsanfragen wuchs ebenso kontinuierlich von 3,4 Personentagen im ersten Jahr auf 4,2 Personentage im letzten Jahr. Im letzten Jahr musste ein Aufwand von 24,5 Personenjahren zur Fehlerbeseitigung und 23,5 Personenjahren für adaptive Wartung betrieben werden. Diese 48 Personenjahre machten insgesamt einen Anteil von 26% der gesamten Personalkosten aus.

Während der gesamten fünf Jahre wurden ungefähr 85 Personenjahre für Reengineering-Projekte zur Verringerung der Komplexität und Verbesserung der Adaptilität investiert. Dies erforderte 9% der gesamten Personalkosten. Die Kostenberechnungen für die ersten Projekte dieser Art waren nicht sehr genau, da sich Unterschiede von bis zu 40% bezüglich der erwarteten und den tatsächlichen Kosten ergaben. Als aber erst einmal die genaue Produktivität für Reengineering-Projekte bekannt war, verbesserten sich

die Ergebnisse rapide. Die Kosten für ein späteres Projekt zur Internationalisierung des Systems durch Erweiterung um verschiedene Sprachen wurden lediglich um 3% überschätzt. Daraus zeigt sich, dass die Berechnungen im Bereich Reengineering sehr zuverlässig werden, wenn erst einmal die Werkzeuge, Methoden und Produktivität in diesem Bereich bekannt sind.

Die Berechnungen für funktionale Erweiterungen waren die fehleranfälligsten, da durch Gewinnung neuer Kunden der Bedarf an unterschiedlicher Funktionalität unerwartet anstieg. Dies führe zu einer Verdopplung des Systemumfangs von anfänglich 92 000 *Function Points* im ersten Jahr auf 185 000 *Function Points* im fünften Jahr. Dieser Anstieg der Systemgröße um 101% erforderte einen Aufwand von 6435 Personenmonaten. Daraus ergab sich ein Median von 14,5 *Function Points* pro Personenmonat über die gesamten fünf Jahre.

Probleme bei der Berechnung der Produktivität ergaben sich aus der Tatsache, dass diese von Projekt zu Projekt stark schwankte. Während einige Projekte den Median der Produktivität um bis zu 70% bei 24 *Function Points* pro Personenmonat übertrafen, erreichten andere nur knapp die Hälfte des Medians mit 7 *Function Points* pro Personenmonat. Diese Differenz war auf unterschiedliche Ursachen zurückzuführen. Eine Ursache lag in schwachem Teammanagement, eine andere in mangelnder Erfahrung und ein dritter Grund lag in der Verteilung des Projekts. Externe Teams außerhalb des Stammsitzes der Firma hatten eine geringere Produktivität von 8 bis 16 *Function Points* pro Personenmonat.

In einem kritischen Projekt zur Erweiterung des Systems, nämlich der Entwicklung einer Schnittstelle zur Schweizer Börse, welches an eine Tochterfirma vergeben wurde, ergab sich eine Unterschätzung der Kosten um mehr als 50%, obwohl die Größe korrekt vorausgesagt wurde. Es wurde von 485 *Function Points* ausgegangen und umfasste letztendlich 472 *Function Points*. Die Einschätzung des Aufwands auf 24 Personenmonate basierte auf einer Produktivität von 20 *Function Points* pro Personenmonat. Bis das Teilsystem jedoch einsatzbereit war, mussten 52 Personenmonate mit einer Produktivität von 9 *Funktion Points* pro Personenmonat aufgewendet werden.

Daraus ergibt sich die Schlussfolgerung, dass die Produktivität einer der Hauptfaktoren bei der Berechnung von Kosten ist und deren signifikate Schwankung zwischen einzelnen Personen und Organisationen berücksichtigt werden muss. Ist diese für die betroffenen Personen oder Organisationen nicht bekannt, dann ist das Risiko einer Fehlberechnung groß, besonders im Bereich der Weiterentwicklung. Unabhängig davon, wie genau die Vorhersage der Größe des Projekts ist, führt eine falsche Annahme bezüglich der Produktivität zu unbrauchbaren Ergebnissen [Sne04a] [JS02].

9.8 Vergleich des *MainCost*-Modell mit klassischen Berechnungsmodellen

Das hier vorgestellte *MainCost*-Model zur Berechnung von Softwarewartungs- und -evolutionskosten unterscheidet sich von klassischen Berechnungsmodellen wie *Function Point* oder *COCOMO-II* insbesondere dadurch, dass es durch eine klare Trennung von Softwarewartung und -evolution, ausgehend von dem Zustandsmodell, eine genaue Zuordnung von Kosten zu entsprechenden Budgets von fixen und variablen Kosten erlaubt. Des weiteren kann man durch eine genauere Unterteilung in Einzelaktivitäten, Berücksichtigung der für jeden einzelnen Teilbereich relevanten Einflussfaktoren, auch jenseits des eigentlichen Produkts, und durch individuelle Berechnungsmethoden für jeden Teilbereich, auf ein genaueres Ergebnis hoffen. Eine Studie mit einem direkten Vergleich der genannten und evtl. weiterer Verfahren wäre diesbezüglich wünschenswert. Allerdings haben die Erfahrungen mit dem *MainCost*-Modell in der Praxis gezeigt, dass erst eine gewisse, dem Projekt entsprechende Datenbasis vorhanden sein muss, bevor *MainCost* zuverlässige Ergebnisse liefert. Im Gegensatz dazu sind *Function Point* oder *COCOMO-II* mehr oder weniger direkt und unabhängig von solchen empirischen Daten einsetzbar. Hierbei ist natürlich zu beachten, dass auch hier für individuelle Organisationen erhobene Daten zu besseren Ergebnissen führen. Allen Modellen gemein scheint der große Einfluss der Einschätzung der Produktivität auf die Güte des Ergebnisses zu sein. Bei der *Function Point*-Methode erfolgt dies durch die Wahl einer geeigneten Tabelle zur Umrechnung von *Function Points* in Personenmonate. Bei der *COCOMO-II*-Methode durch die richtige Wahl der entsprechenden Einflussfaktoren und bei der *MainCost*-Methode durch direktes Berechnen einer projektbezogenen Produktivitätsrate, basierend auf während des Projekts gewonnener Daten. [Bal01][Sne04a].

9.9 Zusammenfassung

In dieser Seminarausarbeitung wurde kurz die Prinzipien der Kostenberechnungsmodelle *Function Point* und *COCOMO-II* vorgestellt und etwas genauer auf den Aspekt Behandlung von Wartung und Evolution durch diese Modelle eingegangen. Des weiteren wurde ein Kostenberechnungsmodell für separate Berechnung von Softwarewartungs- und -evolutionskosten vorgestellt, welches auf dem ebenso beschriebenen *Zustandsmodell* beruht. Die Erfahrungen mit diesem Modell in der Praxis wurden ebenso beschrieben, wie auch ein Vergleich zwischen den oben genannten klassischen Berechnungsmodellen und dem neuen Modell gezogen.

Literaturverzeichnis

[ASKK03] Y. Ahn, J. Suh, S. Kim, and H. Kim. The software maintenance project effort estimation model based on function points. *Journal of Maintenance and Evolution: Research and Practice*, 15:71 – 85, 2003.

[Bal01] H. Balzert. *Lehrbuch der Softwaretechnik*. Spektrum Akademischer Verlag, 2001.

[BD91] E. Bersoff and A. Davis. Impacts of life cycle models on software. *Comm. of ACM*, 34(8):104, 1991.

[BL76] L Belady and M. Lehmann. A model of a large program developement. *IBM Systems Journal*, 15(3):225, 1976.

[Boe88] B. Boehm. A spiral model of software developement and enhancement. *IEEE Computer*, page 61, May 1988.

[BR01] K. Bennett and V. Rajlich. Software maintenance and evolution - a staged model. ICSE-2000, pages 73–89, Limerick, 2001. IEEE Press.

[COC] http://sunset.usc.edu/research/COCOMOII/.

[Dro95] D. Dromey. A model for evaluating software product quality. *IEEE Trans on S.E.*, 21(2):146–152, 1995.

[GEO] http://www.geos.biz.

[Hug00] B. Hughes. *Practical software measurement*, page 143. McGraw-Hill, London, 2000.

[HW02] M. Harrison and G. Walton. Identifying high maintenance legacy software. *Journal of Software Maintenance*, 14(6):429, 2002.

[JS02] M. Jörgensen and D. Sjöberg. Impact of experience on maintenance skills. *Journal of Software Maint.*, 14(2):123–146, 2002.

[KMFA00] M. Kajko-Mattson, S. Forsannander, and G. Anderson. Software problem reporting and resolution process at abb. *Journal of Software Maint.*, 12(5):255–286, 2000.

[Mar83] W. Martin, R.J.and Osbourne. Guidance of software maintenance. NBS Pub., pages 100–129. U.S. Nat. Bureau of Standards, December 1983.

[MB03] M. Maare and A. Bertolio. Using spanning sets for test coverage measurement. *IEEE Trans on S.E.*, 29(11):974, 2003.

[RBC03] B. Reifer, D.and Boehm, V. Basili, and B. Clark. Maintaining cots systems - 8 lessons learned. *IEEE Software*, page 69, September 2003.

[Sel03] B. Selic. The pragmatics of model-driven developement. IEEE Software, page 19, September 2003.

[Sne91] H. Sneed. The economics of software reengineering. *Journal of Software Maint.*, 3(3):163–182, 1991.

[Sne95] H. Sneed. Estimating the cost of software maintenance tasks. Proc. of Int. Conf. on Software Maint., pages 168–181, Opio, France, October 1995. IEEE Computer Society Press.

[Sne97] H. Sneed. Measuring the performance of software maintenance departments. Proc. of first European Conf. on Software Maint. And Reengeneering, page 119, Berlin, March 1997. IEEE Computer Society Press.

[Sne04a] H. Sneed. A cost model for software maintenance and evolution. ICSM. IEEE Press, 2004.

[Sne04b] H. Sneed. Test case analysis. Proc of CSMR-2004, Tampere, March 2004. IEEE Computer Society Press.

Chapter 10

Service Perspective on Maintenance

Martin Herbort, BSc.

Contents

10.1 Abstract

In this paper, the process of software maintenance is investigated from a service-oriented point of view as introduced by Niessink and van Vliet [11].

First of all, some notions are clarified: What distinguishes a software product from a service related to the maintenance of that product? What are the key criteria? How to judge and improve quality of services? Therefore, activities are described that involve both the customer, i.e. the user of a software system, and the maintaining organization. Accordingly, the GAP model introduced by Parasuraman et al. is presented. It shows different ,,perception gaps“ that do typically occur within the process of maintenance service provision.

The latter part of this work discusses approaches to implement a service-oriented software maintenance process. This also involves the consideration of software tools that provide means for effective and efficient service provision.

10.2 Introduction

Comparing Products and Services A software product is always the outcome of a software development process. Although there are a lot of artifacts along the development cycle, its main representation is still the executable actual program code. That is, software is to some extent tangible: By reading the program code you can indeed *see* the product. By contrast, the definition of a service is somewhat more complicated. There are many different views on the notion of a service. A quite plausible definition is given by Grönroos [3]. He defines the service as ,,an essentially intangible set of benefits or activities that are sold by one party to another“ [11]. Accordingly, services cannot be experienced as directly as it is possible with products: What you may see or feel is the *effect of a benefit* or the *result of an activity* but not the benefit or activity itself. Intangibility is the first aspect of the differences between products and services.

The second aspect is heterogeneity. If a customer needs a software product to fulfill a particular task, he typically writes down (in a defined manner of course) a set of requirements. When the product is delivered, he can judge its quality by simply trying out whether or not the product fulfills the desired task. It is important to recognize that the product, being program code that gets executed on a machine, works in a defined (here: deterministic or *homogeneous*) manner. Services do not. They are delivered in terms of activities that are performed by humans. That is, communication issues are far more involved. Consequently, services ,,work“ in a more *heterogeneous* manner. Section 10.3 provides more information on this topic.

The third aspect is simultaneous production and consumption. Typically, the processes of production and consumption can be separated for products but not for services. There are two extreme *production/consumption models* (p/c models) one can think of:

1. Production completely before consumption

2. Production and consumption completely parallel

Imagine an automobile company. A car gets produced in a fabric and sold (hence consumed) a few months *later* in a subsidiary. This relates to the first p/c model. By contrast, a transportation service, e.g. providing a bus connection from A to B, needs to be delivered *while* being consumed. This relates to the second p/c model. For software products, such a clear classification cannot be given in general. The reason is that the software product lifecycle may involve a maintenance concept that indeed results in partly concurrent production and consumption. Two of those concepts are *perfective maintenance* and *enhansive maintenance* [12]. Nevertheless, some points distinguish products from services related to production and consumption. For example, customers of a product do not affect each other, but customers of a service potentially do. That is, multiple customers can use their respective local copy of a software product independently. On the other hand, the overall maintenance service quality may sink with a growing number of customers, if the number of responsible employees of the maintenance company is fixed. This, of course, must be avoided!

The fourth and last aspect is perishability. „Services cannot be saved or stored. Consequently [...] services cannot be returned or resold“ [11]. This again reveals the more dynamic character of a service compared to a product.

How to classify software maintenance? We take into account the widely adopted four types of maintenance as given by Lientz and Swanson [8]:

1. Corrective maintenance

2. Adaptive maintenance

3. Perfective maintenance and

4. Preventive maintenance

They all have in common that they focus on activities „aimed at keeping the system usable and valuable for the organization“ [11]. That is, the benefits for the customer lie in the activities. Together with the aspects discussed above, software maintenance can generally be seen as a service. But it may have some product-like augmentations. Consider a service pack to a large software system. If it is delivered on CD, then this CD can be seen as such a product-like augmentation. However, that service pack does not represent the total outcome of the maintenance service. Section 10.3 and 10.4 give you a flavour what the outcome of the software maintenance service can consist of.

How to judge quality of products and services? If a customer evaluates the quality of a software product he will refer to the requirements document he and the development organization agreed upon. As requirements play a key role within product evaluation, the two fundamental categories are pointed out here:

Functional Requirements Describe *what* functions the product provides and *what* tasks it can fulfill. For example: „There is an online payment module that supports the major credit cards“.

Non-Functional Requirements Describe different aspects of the product. Popular representatives include performance, reliability and security. They say something about *how* the product fulfills certain tasks. For example: „The online payment module is safe from man-in-the-middle attacks“.

Note, that the question whether or not a product complies with its requirements can only be decided *after* delivery, that is *after* production

The *what* and *how* dimensions can also be found when evaluating services. Here, technical and functional quality are the main categories [3]:

Technical Quality Determined by the outcome of the service, „*what* the customer is left with when the service has been delivered“ [11].

Functional Quality Determined by the way the customer gets the service. That is, *how* the service is delivered.

So far, the differences of products and services have been discussed (with focus on software products and related maintenance services). But because maintenance services also have some product-like aspects (cp. service pack example) and because all maintenance organizations sell their services as a product, you can define a more general notion of a *product* that is suitable both for software products (in terms of code) *and* the software maintenance service (see figure 10.1). One of the main differences between these two interpretations of products is the way of how the product is evaluated by the customer.

Section 10.3 describes, what difficulties arise when providing software maintenance services and what activities are entailed by these difficulties. Subsequently, section 10.4 takes a closer look at two approaches of implementing such activities, namely the IT Infrastructure Library (ITIL) and the IT Service Capability Maturity Model (IT Service CMM). Section 10.5 contains some notes on the applicability of software tools that aim at facilitating the provision of software maintenance services. Section 10.6 draws the conclusions of this work.

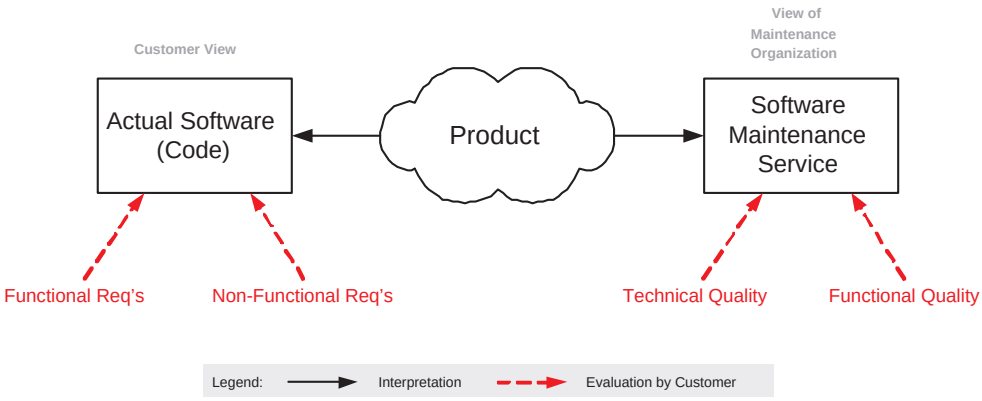


Figure 10.1: General notion of a product

10.3 The GAP Model and Deduced Best Practices

As already stated in the introduction, the process of bringing software maintenance services to the customer heavily relies on communication between the customer and the service providing organization. Hence, you can think of a *perception gap* that results from the fact that the service as expected by the customer differs from the service the service provider believes the customer would expect. This and other perception gaps constitute the GAP model (see figure 10.2).

Possible reasons for this first gap (GAP 1 in the figure) are „inadequate market research, lack of communication between contact employees and management and insufficient relationship focus“ [11]. For example, consider the adaptive activity of software maintenance. It is about adapting a software system to a changing environment. Sneed et al. state that there exist even two different environments [12]:

Business Environment Laws, policies, use cases and business processes.

Technical Environment Technical fundaments like computers, operating systems and database systems a software product uses.

Now assume that a customer and a maintenance organization agree on the term „changing environment“ where the customer thinks about the business environment and the maintainer has the technical environment in mind. Then the perception gap is perfect in a negative sense! As a consequence, there is a need for clear service agreements. This is typically done by setting up a special document (see below).

GAP 2 denotes the deviation of „the service specification as used by the service provider from the expected service as perveiced by the service provider“ [11]. It may result from a lack of defined methods to plan service activities - or there exist defined but inflexible methods. For example, assume that the maintainer wants to guarantee 99,9 percent system availability. But instead of focusing on a quick system restart after failure, the standard procedure used requires a detailed crash analysis first.

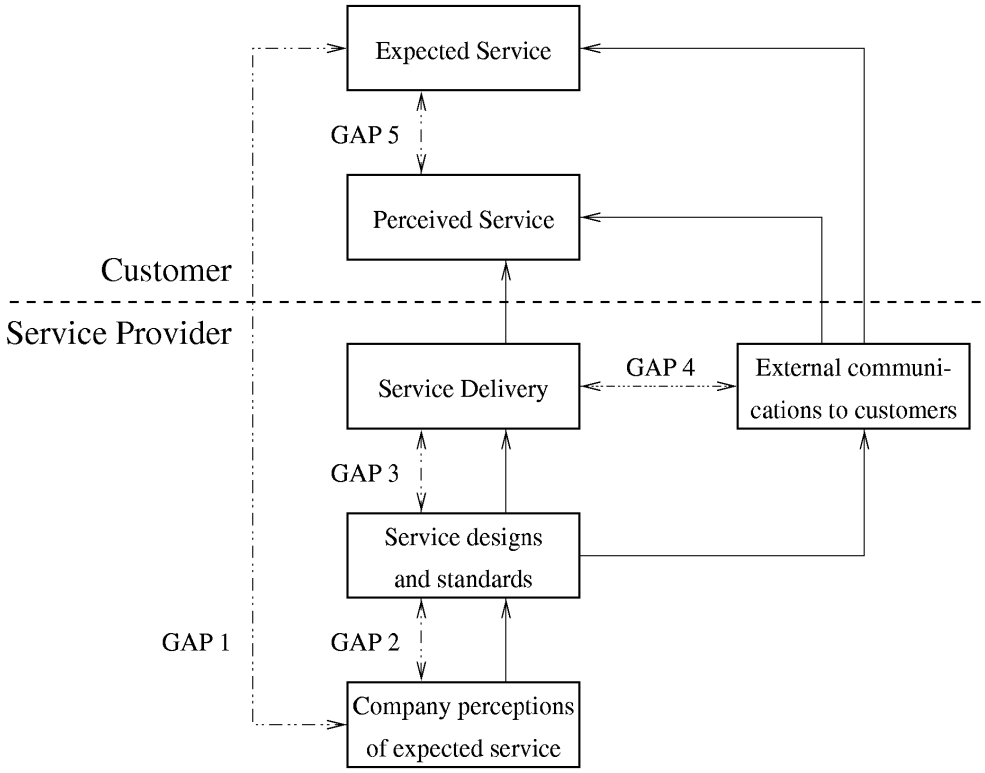


Figure 10.2: The GAP model of service quality (Parasuraman, Zeithaml and Berry, 1985)

GAP 3 involves the interface between the customer and the maintaining organization. Typically, human resources management is the reason why the service delivered in terms of activities differs from the service specification. For example, there might be not enough employees available to perform a particular activity (e.g. a user helpdesk). Additionally, you can think of customers ,,not fulfilling their role“ [11]. For example, assume a user of a software product who wants to report a bug. But instead of using the intended trouble ticket system (see section 10.5) he might have got the phone number of the product manager and call him directly. Hence, in order for the maintainer to keep up-to-date on the service delivery, maintenance activity tracking is very important.

GAP 4 again relates to communication issues. It describes the difference between the communication of a service and its actual delivery. For example, assume a ready-to-download service pack (that incorporates several bugfixes) the customer is not notified about. The presence of a service pack can be seen as an *event*. As there are many other types of events and because events typically occur very often, there is a need for efficient event management.

GAP 5 is the ,,net difference“ between what service quality the customer perceives and what service quality he expects. It is caused by the GAPs 1-4.

Closing the GAPs After having identified the perception gaps, the need for techniques to close these gaps becomes obvious. The following paragraph summarizes such techniques as given by Niessink and van Vliet [11].

1. Instead of using vague formulations use more formal methods to translate customer expectations with respect to maintenance into clear service agreements.
2. Use these service agreements as a basis for planning and implementing maintenance activities.
3. Ensure that maintenance is done according to planning and procedure.
4. Manage the communication about the maintenance activities carried out.

The following paragraph introduces Service Level Agreements (SLA) as a widely used measure to keep expectations of the customer and the service provider congruent. Section 10.4 analyzes two more wholistic approaches with respect to how they take these paradigms into account.

Service Level Agreement (SLA) As service provision in general and software maintenance in particular takes place between two parties, namely the customer on one side and the service organization on the other side, it is important to ,,work together towards the specification of *relevant and realistic* maintenance service commitments“ [11]. The outcome of this process is a *contract* in terms of a document containing clear statements about the maintenance services to be delivered. Such a document is called *Service Level Agreement*, or SLA. As a requirement, committed service levels have to be *measurable*. This facilitates controlling and reporting of maintenance activities. In order to address GAP 1 mentioned above, the service commitments contained in the SLA should be derived from the needs of the customer rather than just from the capabilities of the maintenance organization. This implies, that an SLA has to be reviewed periodically to ensure that the service commitments are still in line with possibly changing maintenance needs.

An SLA should at least specify the following:

- Actual maintenance services, e.g. a bug reporting service
- Agreed service levels, e.g. availability, maximum reaction time
- Customer obligations, e.g. what input to provide in case of a bug, document formats to be used
- Consequences of a service level breach, e.g. penalty of non-performance
- Reporting behaviour, e.g. when and what will be reported to the customer
- Review planning, e.g. when and how the SLA will be reviewed

- Exeptional circumstances under which service provision is not guaranteed as usual

10.4 Approaches to Realize Service-Oriented Software Maintenance

The preceding section has outlined the need for wholistic approaches towards customer-focused IT service provision. The mere setup of an SLA is not enough for high-quality IT services. The specific characteristics of software maintenance (e.g. user support, release management) imply procedures that cover user-related, development-related and economic issues and potentially more. Accordingly, two approaches are presented in the following that involve suitable processes. Besides the already widely adopted IT Infrastructure Library (ITIL), the emerging IT Service Capability Maturity Model is explained on an introductory level.

10.4.1 IT Infrastructure Library (ITIL)

The IT Infrastructure Library (ITIL) is a customizable framework of best practice recommendations that target at optimizing service provision of IT service organizations. That is, ITIL is an approach to IT service management in a rather general sense. „ITIL addresses the organizational structure and skill requirements for an IT organization by presenting a comprehensive set of management procedures with which an organization can manage its IT operations“ [13]. It is a publicly available de-facto industry standard in the area of IT service management that is maintained by the UK's Office of Government Commerce (OGC). The framework focuses on matching end-user expectations [1]. Therefore, the field of the processes described also covers activities that are very close to the customer (e.g. provision of a *service desk*, see below). Note that ITIL makes statements about *what* must be done to assure some level of service quality. It does dictate *how* this goal must be reached.

Relationship with Software Maintenance Although ITIL is a comprehensive framework going far beyond software maintenance, there are several parts in this standard that have a very close relationship with the process of software maintenance. Particularly, ITIL deals with *incidents*, *problems* and *known errors*. These notions relate to *error*s, *faults* and *failures* respectively as defined by Liggesmeyer [9]. Both sets of notions actually mean the same, namely some kind of unexpected state or behaviour. But *incident*, *problem* and *known error* in the ITIL sense emphasize the service aspect, whereas *error*, *fault* and *failure* are more focused on the product. See figure 10.3 for the concrete relationship.

As far as software systems are concerned, an incident is an unexpected event with unknown root cause, that negatively influences the operation of the software system.

For example, the fact that the print functionality of a document processor does no longer work, is considered an incident. After having identified the root cause, the incident becomes a problem. Referring to the preceding example, the root cause may be a programming bug. When a work-around (here: correct code) is *known* and not yet implemented, the problem becomes a known error.

Structure ITIL is organized in a set of *books* each of which covers one topic, i.e. a particular aspect. Each book is further devided into *disciplines* that correspond to more concrete activities. The core framework consists of 10 processes (on an operational level) that are described in two books as follows:

Service Support

- Incident Management: Rapid restoration of a normal service operation with respect to agreed service levels.
- Problem Management: Comprehensive analysis of recurring incidents.
- Change Management: Process of resolving known errors. Therefore, *Requests for Change* (RFCs) are generated.
- Release Management: Planning and controlling of distribution and installation of the system.
- Configuration Management: Tracking of all configuration items in the system (hardware, software) and the relationships between them.

Service Delivery

- Service Level Management: Negotiation of clear service level agreements with the customer.
- IT Financial Management: Identification and controlling of the costs of the services used by the customer.
- Capacity Management: Resource and investment management aiming at rapid response to customer inquiries.
- Availability Management: Cost-effective fulfillment of availability requirements.
- IT Continuity Management: Identification of vulnerabilities and risks within information resources. Also includes the decision process on appropriate countermeasures.

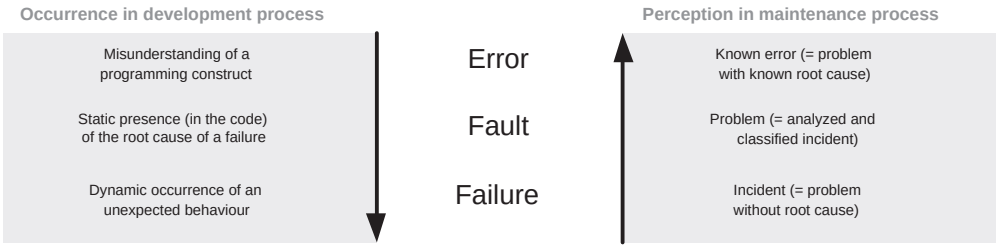


Figure 10.3: Tracking of unexpected behaviours

According to ITIL, the *service desk* is the main point of contact between the IT service organization and their customers. The task of the service desk is to resolve or at least record user inquiries that can be of different type. By contrast to similar concepts, e.g. help desk or call center, the service desk in the ITIL sense covers a significantly wider spectrum. Additionally to the handling of incidents, problems and questions, it also ,,provides an interface for other activities such as customer change requests, maintenance contracts and software licenses“ [13]. Therefore, the service desk can be seen as an ,,implementation“ of the activities of the Service Support book. Figure 10.3 shows how unexpected behaviours, dealing with is part of the activities of the service desk, are tracked compared to their occurrence within the development process.

The independent IT Service Management Forum (ITSMF) states that today, the books represent only about 1% of all ITIL related products and enabling services sold worldwide [6]. These products and services aim at helping a particular organization, e.g. a software maintenance organization, to implement the ITIL standard and therefore be able to bring optimized services to the customer. This situation makes is harder to investigate what is *really defined* by the standard and what is sold by the industry as *being compliant* with the standard. The following paragraph provides an overview of all ITIL books together with a brief description.

1. Service Support: Allows for efficient provision of IT services (consider service desk).

2. Infrastructure Management: Defines organization and initiation of an enabling infrastructure.

3. Service Delivery: Defines processes to match requirements between customer and service organization (typically involves SLAs, see section 10.3).

4. Security Management: Deals with information security.

5. Application Management: Defines how to integrate existing and new software applications. Involves the development lifecycle.

6. Planning to Implement Service Management: Describes analysis of service optimization potentials (in ITIL sense) and adaption of processes to the resulting (changed) requirements.

7. Business Perspective: Explains the key principles and requirements of the service organization in order to implement ITIL.

10.4.2 IT Service Capability Maturity Model (IT Service CMM)

The IT Service Capability Maturity Model (IT Service CMM) is a maturity growth model derived from the SEI's Software Capability Maturity Model (IT Software CMM) version 1.1 to fit in the context of IT service provision. As opposed to the latter one, the IT Service CMM claims to be applicable to maintenance-only organizations as well. It has two main goals [10]: First, it enables IT service providers „to assess [i.e. measure] their capabilities with respect to the delivery of IT services". Second, it specifies „directions and steps for further improvement of service capability". The measurement of the capabilities of the IT service processes is done on a five level ordinal scale. Each maturity level prescribes certain key processes that have to be established before the service providing organization belongs to that level. Each key process „implements a set of related activities that, when performed collectively, achieve a set of goals". Before investigating the structure of the IT Service CMM, a basic knowledge about the different levels is necessary. They are very similar to the levels of the IT Software CMM and are defined as follows [11]:

1 - Initial The IT service delivery process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics.

2 - Repeatable Basic service management processes are established to track cost, schedule and performance of the IT service delivery. The necessary discipline is in place to repeat earlier successes on projects with similar services and service levels.

3 - Defined The IT service processes are documented, standardized, and integrated into standard service processes. IT services are delivered using approved, tailored versions of the organization's standard service processes.

4 - Managed Detailed measurements of the IT service delivery process and service quality are collected. Both the service processes and the delivered services are quantitatively understood and controlled.

5 - Optimizing Continuous process improvement is enabled by quantitative feedback from the processes and from piloting innovative ideas and technologies.

The question now is, what a high score according to this levels is good for. The specification promises a well-performing organization to be able to... [10]

- deliver quality IT services, tailored to the needs of its customer,
- do so in a predictable, cost-effective way,

- combine and integrate different services, possibly delivered by different service providers, into a consistent service package and to
- continually and sustainably improve service quality in a customer-focused way.

Structure and Relationship with Software Maintenance As mentioned above, the advancement from one level to the next one requires several *key processes* (also: *key process areas*) to be implemented. For the sake of simplicity, the explanation of the structure of the IT Service CMM is limited to exactly one of the processes needed to reach the repeatable level (level 2).

In section 10.3, the need for clear service agreements has been shown. The related IT Service CMM key process that serves this purpose is *Service Commitment Management*. It defines several *goals* to be achieved. One of these is:

> Service commitments are documented and agreed to by the customer and the IT service provider.

Besides goals, each key process is is also related to *commitments*. Commitments say something about what is necessary to get the process started. This typically involves organizational policies and leadership. Here, one particular commitment is:

> A service manager is designated to be responsible for negotiating service commitments.

In order conduct necessary *activities*, a set of *abilities* (i.e. preconditions) has to be demonstrated first. By contrast to commitments, abilites typically involve resources, organizational structures and training. An example for an ability is:

> Adequate resources and funding are provided for developing the service commitments.

A particular actual activity is outlined by (and described in the specification in a stepwise manner):

> The service commitments are documented.

As one of the primary goals of the IT Service CMM is the assessment of capabilities, *measurements* are defined. Related to *Service Commitment Management*, this means:

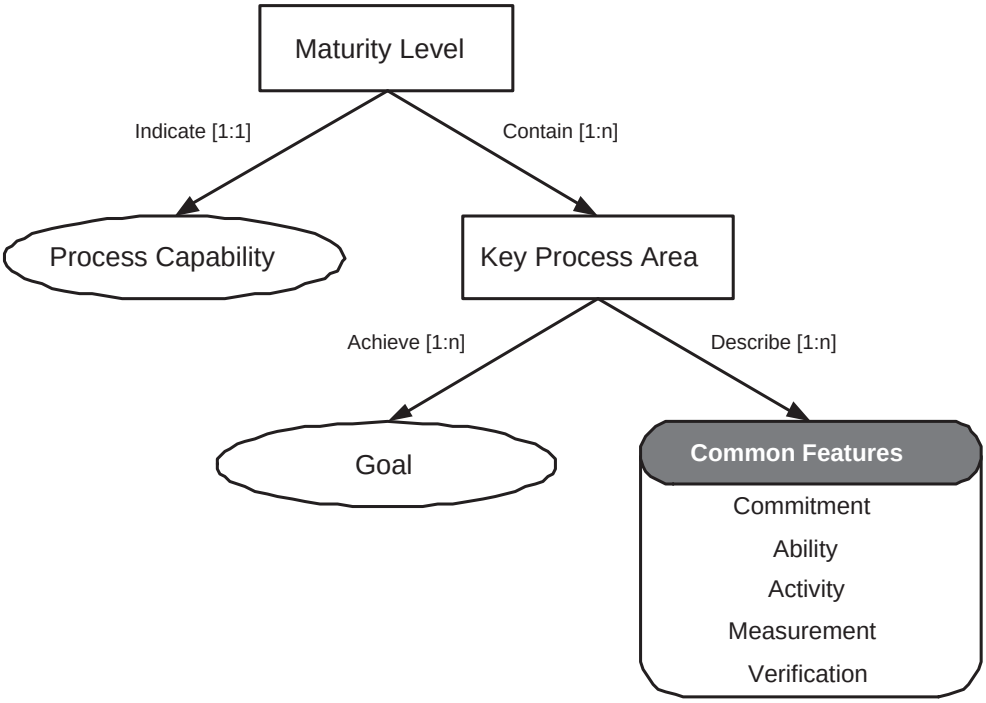


Figure 10.4: Structure of the levels associated with the IT Service Capability Maturity Model

> Measurements are made and used to determine the status of the service commitment management activities, e.g. *work completed*, *effort expended* and *funds expended in the service commitment management activities compared to the plan*.

To verify implementation of the key process, *verifications* are specified. Here, a particular verification is:

> The service commitment management activities are reviewed with senior management on a periodic basis.

Commitments, abilities, activities, measurements and verifications are different types of process-specific *common features*. Figure 10.4 visualizes the structure of the maturity levels according to the specification.

10.4.3 Evaluation

The evaluation of ITIL relies on publicly available information on the structure and content of the framework, but also on business reports and academic publications. Hochstein, Zarnekow and Brenner have conducted a formal assessment of the ITIL

framework on the basis of established criteria according to the ,,principles for orderly modeling“ [5]. Particularly, the principles ,,Adequacy of Construction“, ,,Adequacy of Language“ and ,,Economy“ were considered important. Therefore, they analyzed the published ITIL documents and conducted four case studies. According to the assessment, the main deficits of ITIL include a lack of formalisation (thus implying a significant risk of inconsistencies) and a lack of minimalism. Further, ITIL is claimed to be rather a *common-practice* model and not a *best-practice* model. That is because ,,innovative insights based on a well-founded theory, a requirement for best-practice models, are not emphasised by ITIL“. Nevertheless, the four case studies have shown that ITIL has a positive Return-on-Invest (ROI). This observation is corroborated by a study conducted by the German Competence Center for Business Process Management: 34.3% of all companies in the German-speaking area (multi-class businesses, large group of financial services providers) use ITIL [7]. Hochstein and Hunziker draw very similar conclusions [4]. They indicate further that the OGC has developed an ITIL-specific process model. This and the generic characteristics of ITIL aim at facilitating adaption to different scenarios. Additionally, the large international community around ITIL helps that the framework will remain up-to-date and relevant.

Because of the literature known and the number of ,,ITIL-compliant“ products, ITIL is seen as a rather pragmatic approach. By contrast, the focus of the IT Service CMM is the complete service organization [10]. The specification itself states that ,,the IT Service CMM focuses more on the what and the ITIL more on the how“. It is still work-in-progress but there are a couple of assessments presented on the IT Service CMM website. Niessink and van Vliet state that all disadvantages of the original Software CMM also hold for the IT Service CMM [11]. One of these is the fact that ,,the massive attention of organizations to obtain CMM-like certification holds the danger that focus shifts from developing (and maintaining) software to developing processes“. But good processes do not imply high-quality software products and maintenance.

The practice-based ITIL, obviously being well-accepted throughout the industry, seems to be ahead of the process-oriented IT Service CMM. The latter one is still under development and therefore, did not have the chance to prove itself, yet.

10.5 Tool Support

In respect of software maintenance, there are several areas, where software tools can obviously be applied. Two of these are configuration management and request management. The strong relationship with the service perspective is shown particularly in section 10.4.

Configuration Management ,,Almost all IT services concern the management, operation or maintenance of hardware and software components“ [11]. In general, this implies the need for putting all these components under configuration control. Part of this is a configuration management database (CMDB) that documents all relevant re-

lationships between the components. Therefore, the CMDB provides detailed knowlegdge about... [1]

- what software is installed on what server,
- what IT components an IT service consist of and
- what components are covered by what maintenance contracts.

In the field of software maintenance, it is important to have an instance of the software system available as it is running on the customer's side. More concrete, a software maintenance organization must manage nearly as many software configurations (often in terms of releases) as it has customers running that software. There is a big number of tools that facilitate just this and a lot more. The span is from open-source tools that focus on source code management to more sophisticated and expensive but powerful solutions that cover automatic builds, baselines and integration with other tools.

Request Management As described in section 10.4, there are several types and also a significant number of user inquiries, e.g. bug reports, user interface problems and change requests. A typical concept that is incorporated by some tools is the concept of a *ticket*: If a particular user of a software product has a problem, he uses a special system provided by the maintenance organization. Such a system is called *ticket request system*. The name results from the fact that once a request is triggered by the user, this request becomes a ticket that represents all (current and succeeding) service and communication actions taken to deal with the request. A user inquiry traverses service hierarchy just a like piece of luggage that travels from A to B where its current state is always referred to by a ticket. In the world of software maintenance service provision, this allows for detailed service delivery measurement and controlling. Again, there is a big number of appropriate tools. It is noticable, that the community around the *Open Source Ticket Request System* (OTRS) has started promising ITIL certification activities that will probably lead to a basic certification level in 2006 [2].

10.6 Conclusions

This work has compared products and services to establish a service perspective on software maintenance. It was shown that the quality of products and services is judged by the customer differently. In both cases a *what* and a *how* dimension is present.

The GAP model has explained different perception gaps that may exist between the customer (i.e. the user of a software system) and the maintenance organization. In particular, the proper definition of Service Level Agreements (SLAs) was considered an appropriate measure to manage service expectations.

The IT Infrastructure Library (ITIL) and the IT Service Capability Maturity Model (IT Service CMM) represent two different approaches to realize high-quality IT services. They can also be used as a basis for efficient software maintenance service provision. ITIL has turned out to be (so far) more popular and more cost-effective compared to the IT Service CMM that is still under development.

Further, service provision can be supported by software products. Configuration and request management are two areas, where the application of tools is definitely reasonable.

Bibliography

[1] Glenfis AG. Itil cobit mapping — basis für das unternehmensweite integrale risikomanagement, 2005.

[2] OTRS GmbH. News — otrs wird itil-konform, 2006. [Online; accessed 18-March-2006].

[3] Christian Grönroos. *Service Management and Marketing: managing the Moments of Truth in Service Competition*. Lexington, Mass., 1990.

[4] Axel Hochstein and Andreas Hunziker. Serviceorientierte referenzmodelle des it-managements. In *Strategisches IT-Management*. Walter Brenner and Andreas Meier and Rüdiger Zarnekow, 2003.

[5] Axel Hochstein, Dr. Rüdiger Zarnekow, and Prof. Dr. Walter Brenner. Itil as common practice reference model for it service management: formal assessment and implications for practice. In *Proceedings of the IEEE International Conference e-Technology, e-Commerce and e-Service*, pages 704–710, 2005.

[6] IT Service Management Forum (itSMF). itsmf website, 2006.

[7] Sabine Koll. Geschäftsprozesse sind fest im griff der it. *Computer Zeitung*, (10), 2006.

[8] B. P. Lientz and E. B. Swanson. *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations*. Addison-Wesley, 1980.

[9] Peter Liggesmeyer. *Software-Qualität*. Spektrum, 2002.

[10] Frank Niessink, Viktor Clerc, Ton Tijdinka, and Hans van Vliet. *The IT Service Capability Maturity Model*, 2005. Version 1.0, Release Candidate 1.

[11] Frank Niessink and Hans van Vliet. Software maintenance from a service perspective. *Journal of Software Maintenance: Research and Practice*, 12(2):103–120, March/April 2000.

[12] Sneed, Hasitschka, and Teichmann. *Software-Produktmanagement*. dpunkt Verlag, 2004.

[13] Wikipedia. Information technology infrastructure library — wikipedia, the free encyclopedia, 2006. [Online; accessed 9-March-2006].

Kapitel 11

Management der Software-Wartung

Katja Gruber

Inhaltsverzeichnis

11.1 Einleitung

Softwaresysteme durchleben regelmäßig einige Veränderungen, sei es durch Fehlerbehebung, Funktionsverbesserungen oder durch die Erstellung einer neuen Version. All diese Veränderungen zählen zur Software-Wartung. Die Wartung gehört jedoch zu einem der am wenigsten-strukturierten Probleme der Softwareentwicklung.
In dieser Ausarbeitung wird ein Rahmenwerk vorgestellt, dass bei der Analyse von Wartungsentscheidungen behilflich sein soll. Das Rahmenwerk betrachtet dabei verschiedene Einflussfaktoren des Software-Marktes, beispielsweise die Qualität und den technologischen Fortschritt. In Kapitel 11.2 werden die wichtigsten dieser Einflussfaktoren vorgestellt. Für jede Analyse wird ein Modell benötigt, daher wird in Kapitel 11.3 das für das Rahmenwerk notwendige Upgrade-Cycle-Modell vorgestellt. In Kapitel 11.4 werden die für das Modell verwendeten Wartungs-Strategien eingeführt. In Kapitel 11.5 wird ein Beispiel aufgestellt und anschließend mit dem Rahmenwerk analysiert. Das Rahmenwerk liefert dann eine Bewertung der Einflussfaktoren und eine Optimalitätsaussage über die Wartungs-Strategien.

Aber nicht nur die Analyse der Wartung sondern auch die Organisation der Wartung sind für das Software-Wartungs Management wichtig. In Kapitel 11.6 wird ein Überblick über die verschiedenen Möglichkeiten der Software-Wartungsorganisation gegeben.
Es scheint offensichtlich, dass es keine eindeutige und einheitliche Lösung für das Management der Software-Wartung gibt, denn je nach den Zielen des Unternehmers, werden einige Faktoren mehr hervorgehoben als andere. Mit einer Zusammenfassung wird diese Ausarbeitung dann in Kapitel 11.7 beendet.

11.2 Die Grundlagen

In diesem Kapitel wird auf die Grundlagen der Software-Wartung eingegangen. In Abschnitt 11.2.1 werden die Bereiche der Software-Wartung sowie die verschiedenen Einflussgrößen des gesamten Software-Marktes erläutert. Die Einflussgrößen lassen sich in interne und externe Faktoren unterteilen. In dem Abschnitt 11.2.2 wird auf die internen und externen Faktoren des Software-Marktes eingegangen. In Abschnitt 11.2.3 wird auf die Kundenerwartung eingegangen, die als externer Faktor betrachtet wird. Die Kundenerwartung wird unter anderem durch den technologischen Fortschritt bestimmt, deshalb wird in Abschnitt 11.2.4 auf diesen externen Einflussfaktor eingegangen.
Es sei darauf hingewiesen, dass die Begriffe „Unternehmer“, „Entwickler“ und „Anbieter“ in diesem und den folgenden Kapiteln 11.3, 11.4 und 11.5 synonym verwendet werden.

11.2.1 Bereiche der Software-Wartung

Bei der Wartung von Software handelt es sich um alle Veränderungen, die nach der Auslieferung an der Software vorgenommen werden. Da es verschiedene Arten von Veränderung an einer Software gibt, wird die Software-Wartung in unterschiedliche Kategorien geteilt. Swanson [Swa76] und Arthur [Art88] unterteilen die Wartung beispielsweise in die korrektive, adaptive und perfektive Wartung. Es sind aber auch weit mehr Kategorien denkbar, wie beispielsweise in [Reu81] oder [KS99] zu sehen.
Für diese Ausarbeitung werden folgende drei Wartungskategorien [SZ01b] definiert, die bei der späteren Einführung des Upgrade-Cycle-Modells benötigt werden.

- **Gewährleistung:** Umfasst die Maßnahmen, die der Entwickler vornimmt, damit die Software die geforderten Anforderungen erfüllt. Es handelt sich also um die Gewährleistung der vom Kunden gewünschten Anforderungen.
- **Wartung:** Umfasst die Maßnahmen, die der Entwickler vornimmt, um die Funktionalität der Software zu verbessern. Bei der Wartung wird also die Implementierung der Software verbessert.
- **Erweiterung:** Umfasst alle neuen Funktionalitäten oder Features, die zusätzlich zur bereits existierenden Software hinzugefügt werden.

Unter einer Gewährleistungsmaßnahme (warranty) wird die Korrektur eines Fehlers verstanden, dessen Existenz die Erfüllung der Anforderungen verhindert. Eine Wartungsmaßnahme (maintenance) kann auch Fehler entfernen, jedoch wird durch eine Wartungsmaßnahme in erster Hinsicht die Funktionalität der Software verbessert. Es folgt also, dass eine Wartungsmaßnahme eine Gewährleistungsmaßnahme enthalten kann, jedoch nicht umgekehrt. Eine Erweiterungsmaßnahme (upgrade) [SZ01b] schließt neue Programmteile ein und stellt somit eine überarbeitete Version dar.
Mit einer Gewährleistungs-, Wartungs- oder Erweiterungsmaßnahme reagiert der Anbieter somit auf eine Gewährleistungs-, Wartungs- oder Erweiterungsmöglichkeit. Um im Folgenden die Aufzählung der drei Maßnahmen bzw. Möglichkeiten abzukürzen, wird allgemein von Wartungsmaßnahmen bzw. Wartungsmöglichkeiten gesprochen. Das bedeutet, Wartungsmaßnahmen bzw. Möglichkeiten stehen für alle drei Kategorien. Wird hingegen von einer Wartungsmaßnahme bzw. Möglichkeit gesprochen, handelt es sich um die spezielle Wartungskategorie.

11.2.2 Der Software-Markt

Das Rahmenwerk zur Analyse von Software-Wartung berücksichtigt neben den eingeführten Gewährleistungs-, Wartungs- und Erweiterungsmaßnahmen auch verschiedene Marktansichten [SZ01b].
Es wird davon ausgegangen, dass die Software für den Massenmarkt entwickelt wird. Um das Ziel, die Maximierung des Gewinns, zu erreichen, ist der Anbieter gezwungen seinen Marktanteil und seinen Kundenbestand beizubehalten. Mit Gewährleistungs-

und Wartungsmaßnahmen wird versucht den Marktanteil zu behalten. Mit Erweiterungsmaßnahmen wird hingegen versucht den Kundenbestand zu erhöhen und somit auch eine Erhöhung des Marktanteils zu erreichen. Dies erzielt der Anbieter durch eine hohe Qualität der Software, ein gutes Preis-Leistungsverhältnis und vor allem durch eine hohe Kundenzufriedenheit. Die Qualität drückt sowohl die Systemfunktionalität als auch die angebotene Güte an Dienstleistungen aus. Das heisst, die Qualität einer Software wird durch verschiedene Einflussfaktoren bestimmt. Dabei wird zwischen internen und externen Faktoren unterschieden [SZ01b]. Bei internen Einflussfaktoren handelt es sich um Faktoren, die der Anbieter selbst beeinflussen kann, wie beispielsweise die Gestaltung und die Implementierung der Software. Externe Einflussfaktoren hingegen können nicht vom Anbieter beeinflusst werden, wie beispielsweise der technologische Fortschritt.

Einer der wichtigsten Faktoren für die Qualität einer Software ist die Kundenzufriedenheit. Sie wird durch den „index of customer satisfaction“ ausgedrückt oder auch als „American Customer Satisfaction Index (ACSI)“ bezeichnet [SZ01b, Bre94]. Der ACSI ist ein unabhängig erhobener Messwert für die Kundenzufriedenheit eines Unternehmens, der vierteljährlich von der Universität Michigan bestimmt wird. Der ACSI liegt zwischen 0 (niedrigster Wert) und 100 (höchster Wert). Er stellt eine Beziehung zwischen zahlreichen Unternehmensdaten und der Kundenzufriedenheit eines Unternehmens her. Denn bei Anstieg des ACSI steigt auch der Gewinn des Unternehmens an. Natürlich hängt der Anstieg des ACSI mit der jeweiligen Reaktion des Anbieters auf Wartungsmöglichkeiten zusammen. Reagiert der Anbieter auf vom Kunden geäußerte Fehlermeldungen oder andere Wartungswünsche nicht, wird folglich der ACSI des Unternehmens sinken.
Ein schlechter Entwurf bietet häufig die Möglichkeit zur Wartung. Eine fehlerhafte Implementierung hingegen führt zu vielen Fehlern und daher zur Möglichkeit der Fehlerbehebung. Die Qualität der Implementierung wird somit durch die Wahrscheinlichkeit des Auftretens einer Gewährleistungsmöglichkeit in einer Periode ausgedrückt, die Qualität des Design hingegen durch die Wahrscheinlichkeit des Auftretens einer Wartungsmöglichkeit in einer Periode. Die jeweiligen Wahrscheinlichkeiten [SZ01b] lassen sich dabei entweder objektiv, durch Beobachten der Vorkommen von Gewährleistungs- und Wartungsmöglichkeiten, oder subjektiv, durch ein Expertenurteil, bewerten. Aber auch die Beschwerden von Kunden werden bei der Beurteilung der Wahrscheinlichkeiten berücksichtigt, denn sie geben Auskunft über die Häufigkeit einer Wartungsmöglichkeit. Die Möglichkeit für eine Erweiterungsmaßnahme setzt voraus, dass der Anbieter die Fähigkeit besitzt, neue Technologien in die bereits existierende Software zu integrieren. Die vorhandenen Ressourcen sind somit ausschlaggebend für die Bestimmung der Wahrscheinlichkeit einer Erweiterungsmaßnahme. Auch diese Wahrscheinlichkeit lasst sich ähnlich wie die Wahrscheinlichkeiten von Gewährleistungs- und Wartungsmaßnahmen beurteilen.

Der Anbieter ist verantwortlich für die Gewährleistung, Wartung und Erweiterung eines Systems und stellt somit den Entscheidungsträger dar. Eines der Hauptentscheidungskriterien ist die Kostenfrage. Also wie viel eine Gewährleistungs-, Wartungs- oder Erweiterungsmaßnahme den Anbieter kosten würde. Für das Beispiel in Kapitel 11.5 werden die Kosten als Durchschnitt aller Kosten der jeweiligen Wartungska-

tegorie angegeben.
Es wurde beobachtet, dass die Verkaufsanzahl einer Software mit ihrem Alter ansteigt [PS94]. Die Begründung dafür liegt an der Innovationsausbreitung [Gri57, GK82]. Bringt ein Anbieter ein neues Produkt auf den Markt, gelten die ersten Käufe dieses Produkts als eine Besonderheit. Doch das neue Produkt wird irgendwann zum Marktstandard. Immer mehr Kunden wechseln zu diesem Produkt und somit wird der gesamte Markt ausgebaut. Dieser Effekt der Innovationsverbreitung wird oft mit dem Lerneffekt verglichen [PS94]. Denn je mehr Kunden das Produkt verwenden, um so schneller tritt der Lerneffekt ein und die Verkäufe dieses Produktes steigen.

11.2.3 Die Kundenerwartung

Im Abschnitt 11.2.2 wurde bereits erwähnt, dass die Reaktionen eines Anbieters mit dem Ansteigen oder Sinken des „index of customer satisfaction“ zusammenhängt. In diesem Abschnitt wird auf die verschiedenen Möglichkeiten der Reaktion eines Anbieters auf Wartungsmöglichkeiten eingegangen.
Grundsätzlich werden beim Arbeiten mit Software irgendwann Defizite oder Mängel festgestellt und es ist wünschenswert, dass die Funktionalität der Software gesteigert wird. Es wird dabei angenommen, dass diese Gelegenheiten rein zufällig auftreten und der Kunde eine entsprechende Reaktion des Anbieters darauf erwartet. Es ist offensichtlich, dass die Zufriedenheit des Kunden mit der Software sinkt, wenn der Anbieter nicht auf die Gelegenheit reagiert. Somit sinkt auch der ACSI. Auf Grund der drei Wartungskategorien (siehe Abschnitt 11.2.1) gibt es auch beim Abfall des ACSI drei Parameter, die den Abfall beeinflussen. Es handelt sich dabei um den Gewährleistungs-, Wartungs- und Erweiterungsabfall.

Bei einer Reaktion des Anbieters auf eine Gewährleistungs- oder Wartungsmaßnahme wird davon ausgegangen, dass der ACSI gleich bleibt [SZ01b]. Denn bei Auslieferung der Software wurde davon ausgegangen, dass sie fehlerfrei ist. Würden Gewährleistungs- oder Wartungsmaßnahmen zu einer Erhöhung des ACSI führen, würden alle Anbieter fehlerhafte Software ausliefern, um beim anschließenden Korrigieren dieser Fehler eine Erhöhung des ACSI zu erhalten. Doch es ist offensichtlich, dass das auf lange Sicht keine brauchbare Strategie wäre. Darum wird angenommen, dass Gewährleistungs- und Wartungsmaßnahmen den derzeitigen ACSI nicht verändern. Mit Erweiterungsmaßnahmen versucht der Anbieter hingegen den ACSI bis auf sein Maximum zu steigern. Aber nicht nur die Reaktion des Anbieters, auch der Software-Markt auf dem die Software angeboten wird, ist ausschlaggebend für das Steigen oder Fallen des ACSI. Auf einem gesättigten Markt werden Kunden nicht so empfindlich auf eine Nicht-Reaktion des Anbieters reagieren, wie beispielsweise auf einem Markt, auf dem die sofortige Reaktion des Anbieters überlebensnotwendig für Kunden ist.

Zur Bewertung der drei Abfallparameter gibt es mehrere Ansätze. Einer ist beispielsweise die direkte Kundenbefragung. Das heisst, der Anbieter befragt den Kunden beim Auftreten eines Fehlers, wie er das System mit und ohne diesen Fehler bewertet. Eine solche Abschätzung könnte beispielsweise von „Behebung dieses Fehlers ist sehr

wichtig“ bis hin zu „Behebung des Fehlers ist unwichtig“ formuliert werden. (Vergleichbar mit den Vorlesungsfragebögen am Semesterende.) Der Unterschied der beiden Abschätzungen (mit Fehler und ohne Fehler) gibt eine Schätzung des Gewährleistungsparameters. Die Wartungs- und Erweiterungsparameter können in einer ähnlichen Weise mittels Kundenbefragung abgeschätzt werden (siehe [SZ01b]). Eine andere Möglichkeit zur Abschätzung der drei Parameter ist, den Unterschied der Indizes kurz vor einer Gewährleistungs-, Wartungs- oder Erweiterungsmaßnahme und nach der jeweiligen Maßnahme zu berechnen. So erhält der Anbieter eine Schätzung, die keinen Einfluss von technologischem Fortschritt aufweist. Weitere Möglichkeiten zur Parameterabschätzung können aus der Literatur von Sahin und Zahedi entnommen werden [SZ01b].

11.2.4 Der technologische Fortschritt

Neben der Kundenerwartung und der Kundenzufriedenheit zählt auch der technologische Fortschritt zu einem externen Einflussfaktor. Außerdem beeinflusst der technologische Forschritt die Kundenzufriedenheit, denn ständiger Fortschritt steigert die Erwartungen bei den Kunden. Neue Technologien treiben den Anbieter dazu, seine Produkte auf diese Technologien umzustellen, damit es zu keinem Verlust der Marktanteile kommt. Es ist offensichtlich, dass nur durch Erweiterungsmaßnahmen auf einen technologischen Fortschritt reagiert werden kann, denn nur mit der Erweiterungsstrategie (siehe Abschnitt 11.2.1 und [SZ01b]) werden neue Funktionalitäten oder Features in das Produkt integriert.
Die Auswirkung des technologischen Fortschritts auf den ACSI wird durch einen Technologieabfall [SZ01b] ausgedrückt. Dieser lässt sich durch den Unterschied des ACSI zweier aufeinanderfolgenden Perioden abschätzen. Dabei sei zu beachten, dass in den zwei aufeinanderfolgenden Perioden keine Gewährleistungs-, Wartungs- oder Erweiterungsmaßnahme stattgefunden hat. Es folgt, dass ein Sinken des ACSI vier Ursachen haben kann: Gewährleistungsabfall, Wartungsabfall, Erweiterungsabfall und Technologieabfall. Das bedeutet aber auch, ein Sinken des ACSI in Perioden, in denen es keine Gewährleistungs-, Wartungs- oder Erweiterungsmöglichkeit gab, entsteht allein durch den technologischen Fortschritt.

11.3 Das Upgrade-Cycle-Modell

Im vorherigen Kapitel 11.2 wurden bereits die verschiedenen Einflussfaktoren vorgestellt, die der Anbieter für seine Reaktion auf Wartungsmöglichkeit berücksichtigen sollte. Der Anbieter hat also in jeder Periode abzuwägen, ob er eine Gewinnsteigerung oder einen Gewinnverlust in Kauf nehmen kann. Reagiert er auf die Wartungsmöglichkeit, dann sind damit zwar Kosten verbunden, aber dafür bleibt der ACSI erhalten. Reagiert er nicht auf die Wartungsmöglichkeit, dann spart er zwar Kosten ein, nimmt aber ein Sinken des ACSI und damit des Marktanteils in Kauf.

Der Entscheidungshorizont des Anbieters ist ein Upgrade-Cycle. Ein Upgrade-Cycle ist die Zeitspanne zwischen zwei aufeinanderfolgenden Erweiterungen, das heisst der Upgrade-Cycle beginnt mit der Vollendung einer Erweiterung und endet, wenn die Software eine erneute Erweiterung durchlebt. Ein Upgrade-Cycle-Modell stellt ein konkretes Maß dar, das zudem für einen Vergleich zwischen verschiedenen Strategien benutzt werden kann. Die Länge eines Upgrade-Cycle stellt eine Zufallsvariable dar. Es ist zu beachten, dass die Länge des Upgrade-Cycle-Modells vom jeweiligen Software-Markt, den jeweiligen Maßnahmen des Anbieters während einer Zeitspanne und den internen Einflussfaktoren abhängt. Der ACSI kann sich in vier verschiedenen Bereichen befinden. In der

1. Do nothing - Region,
2. only Warranty - Region,
3. Warrant und maintain - Region
4. Warrant, maintain und upgrade - Region

Die Grenzwerte W, M und U stehen für die Grenzwerte des ACSI für Gewährleistungs- (Warranty), Wartungs- (Maintain) und Erweiterungsmaßnahmen (Upgrade). W, M und U bilden die Obergrenze der jeweiligen Regionen und schlagen geeignete Reaktionen des Anbieters vor. Dabei verändert sich die Maßnahme des Anbieters beim Sinken des ACSI unter jeden einzelnen Grenzwert. Die Grenzwerte W, M und U sollen so gewählt werden, dass der erwartete Gewinn des Anbieters im Upgrade-Cycle maximiert wird. Das gewählte Optimierungskriterium [SZ00, SZ01a]entscheidet, ob es sich beim Gewinn um den Gesamtgewinn oder den Durchschnittsgewinn handelt.

Die Abbildung 11.1 stellt einen solchen Upgrade-Cycle dar. Es wird angenommen, dass die Wahrscheinlichkeit für das Auftreten einer Gewährleistungs-, Wartungs- und Erweiterungsmöglichkeit während des Upgrade-Cycles sich nicht ändert [SZ01b]. Des Weiteren wird angenommen, dass der Anbieter bei Nicht-Reaktion auf Wartungsmöglichkeiten an Wohlwollen von Kunden verliert und diesen Verlust mit späteren Maßnahmen auch nicht wieder ausgleichen kann. Diese Annahme ist durch die Existenz von Konkurrenten [SZ01b] auf dem Software-Markt gerechtfertigt. Aus Abbildung 11.1 ist zu erkennen, dass der Upgrade-Cycle beginnt, wenn der ACSI seinen maximalen Wert erreicht hat. In der ersten Periode fällt der ACSI wegen des Technologieabfalls, denn eine Gewährleistungsmöglichkeit tritt nicht ein. In der zweiten Periode hingegen tritt eine Gewährleistungsmöglichkeit auf, doch der Anbieter unternimmt nichts, weil sich das System noch in der „Do nothing-Region“ befindet. Der ACSI fällt demnach weiter auf Grund des Gewährleistungs- und Technologieabfalls. In der dritten Periode fällt der ACSI wieder nur auf Grund des Technologieverfalls und unterschreitet dabei den Grenzwert W. Das System befindet sich am Ende der Periode in der „only Warranty-Region“. Das bedeutet, dass der Anbieter auf jede Gewährleistungsmöglichkeit reagiert. Weil keine Wartungs- oder Erweiterungsmöglichkeiten auftreten, sinkt der ACSI jedoch weiter auf Grund des Technologieabfalls.

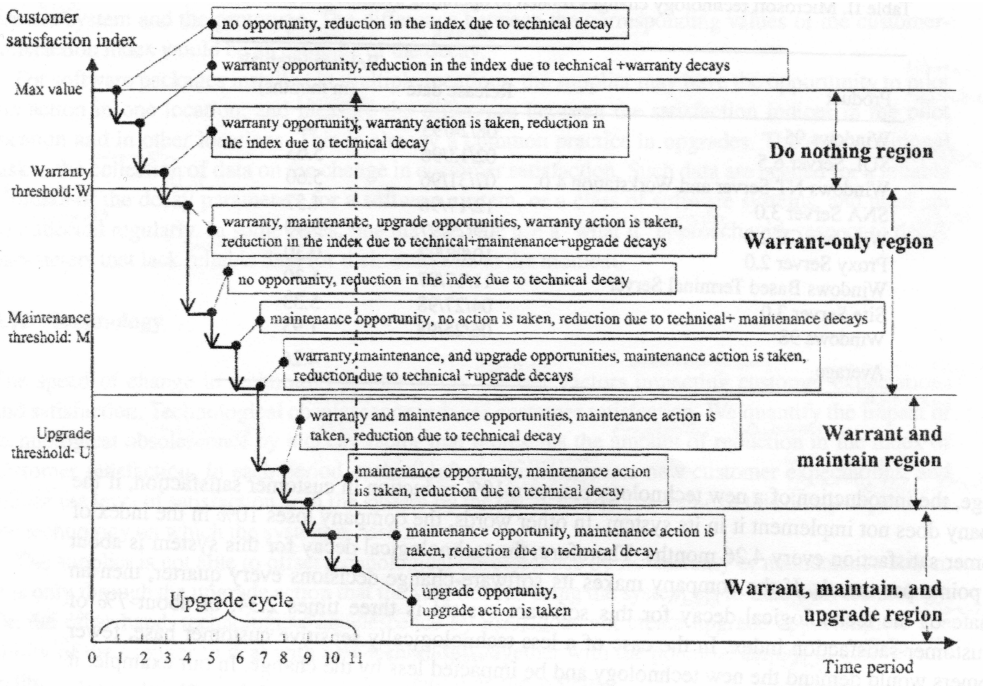


Abbildung 11.1: Ein Beispiel des Upgrade-Cycle-Modells

Unterschreitet der ACSI dann den Grenzwert M, befindet sich das System in der „Warrant und maintain-Region“. Der Anbieter reagiert also auf Gewährleistungs- und Wartungsmöglichkeiten. Auf Erweiterungsmöglichkeiten wird nicht reagiert, so dass diese ein weiteres Sinken des ACSI zur Folge haben. Sobald der ACSI den Grenzwert U unterschritten hat und sich somit in der „Warrant, maintain und upgrade-Region“ befindet, reagiert der Anbieter auf alle Möglichkeiten. Nimmt der Anbieter eine Erweiterungsmaßnahme vor, steigt der ACSI wieder auf sein Maximum und der Upgrade-Cycle ist beendet.

Die Annahme, dass es eine „do Nothing-Region“ gibt, wird auch in der Praxis vorgefunden. Denn kommt eine neue Software auf den Markt oder wurde erweitert, dann existiert eine Zeitspanne in der keine Veränderungen an ihr vorgenommen werden. Diese Zeitspanne wird von der reinen Fehlerkorrektur (Gewährleistung) und anschließender Gewährleistung und Verbesserung der Implementierung (Wartung) gefolgt. Bis die Software letztendlich erneuert wird (Erweiterung).

11.4 Die Baseline-Strategien

In diesem Kapitel wird das Schema einer Strategie zum Upgrade-Cycle-Modell eingeführt. Dazu werden die „Baseline-Strategien“ eingeführt und erläutert.
Eine Baseline-Strategie verweist auf die Werte der Gewährleistungs-, Wartungs- und

Strategie	**Beschreibung**	**empfohlene Maßnahme**
WMU Baseline	W, M und U Grenzwert gleich W, M und U Baselines	immer Gewährleistungsmaßnahme durchführen, warten und erweitern
WM Baseline	W und M Grenzwerte gleich W und M Baseline; U Grenzwert unter U Baseline	immer gewährleisten und warten; nur erweitern wenn ACSI unter U Grenzwert fällt
WU Baseline	W und U Grenzwert gleich W und U Baselines; M und U Grenzwert gleich U Baseline	immer gewährleisten und erweitern; nur warten wenn ACSI unter U Baseline fällt
MU Baseline	M und U Grenzwert gleich M und U Baseline; W und M Grenzwert gleich M Baseline	immer warten und erweitern; nur gewährleisten wenn ACSI unter M Baseline fällt
W Baseline	W Grenzwert gleich W Baseline; M und U Grenzwert gleich und liegen unter U Baseline	immer gewährleisten; warten und erweitern wenn ACSI unter U Grenzwert fällt
M Baseline	M Grenzwert gleich M Baseline; W und M gleich M Baseline; U Grenzwert unter U Baseline	immer warten; nur gewährleisten wenn ACSI unter M Baseline fällt; erweitern wenn ACSI unter U Grenzwert fällt
U Baseline	W, M und U Grenzwert gleich U Baseline	immer erweitern; nur gewährleisten und warten wenn ACSI unter U Baseline fällt
U only	W, M und U Grenzwert gleich und liegen unter U Baseline	nie gewährleisten, warten oder erweitern bis ACSI unter U Grenzwert fällt

Tabelle 11.1: Die 8 verschiedenen Baseline-Strategien

Erweiterungsgrenzwerte, die beim Eintreten einer Wartungsmöglichkeit angeben, welche Maßnahme ergriffen wird. Die in Kapitel 11.3 eingeführten Grenzwerte W, M und U werden als solche baselines betrachtet. Die W baseline in Abbildung 11.1 ergibt sich aus der Differenz des maximalen ACSI und dem Technologieabfall plus 1. Die Addition von 1 am Ende der Formel [SZ01b] resultiert aus der Tatsache, dass eine Wartungsmaßnahme empfohlen wird, wenn der ACSI unter den entsprechenden Grenzwert fällt und nicht, wenn der ACSI den Grenzwert erreicht (d.h. wenn der ACSI in einer der vier Wartungs-Regionen liegt.).
Um die Berechnungen der baselines zu verdeutlichen, wird angenommen, dass der maximale ACSI bei 100, der Technologieabfall bei 2%, der Wartungsabfall bei 6% und der Erweiterungsabfall bei 20% liegt. Dann ergeben sich folgende W, M und U Baselines:

- W = 100 - 2 + 1 = 99
- M = 100 - 2 - 6 + 1 = 93
- U = 100 - 2 - 6 -20 + 1 =79

Das bedeutet, dass bei einem Wartungsgrenzwert von M = 93 und Erweiterungsgrenzwert von U = 79, der Anbieter immer auf eine Wartungsmöglichkeit und Erweiterungsmöglichkeit reagiert, obwohl der ACSI noch sehr hoch ist.

Sahin und Zahedi [SZ01b] haben ein Experiment (in Kapitel 11.5 wird auf dieses Experiment näher eingegangen) mit einem selbst-entwickelten Softwarepaket durchgeführt, um den Erfolg der Baseline-Strategien zu erforschen. In ihrer Analyse ist festgestellt worden, dass fast alle optimalen Strategien die W und M Baseline enthalten und die U Baseline hingegen nur relativ selten. Werden die W, M und U Baseline kombiniert, ergeben sich 8 optimale Baseline-Strategien. Ein Überblick über die 8 verschiedenen Baseline-Strategien kann aus Tabelle 11.1 entnommen werden. Sahin und Zahedi [SZ01b] haben bei genauerer Betrachtung der Baseline-Strategien folgende interessanten Eigenschaften entdeckt:

- Der W Grenzwert ist entweder gleich der W Baseline oder gleich dem M Grenzwert, denn ein W Grenzwert der zwischen der W Baseline und dem M Grenzwert liegt, ist niemals optimal.
- Der M Grenzwert ist entweder gleich der M Baseline oder gleich dem U Grenzwert, denn ein M Grenzwert der zwischen der M Baseline und dem U Grenzwert liegt, ist niemals optimal.
- Der U Grenzwert ist entweder gleich der U Baseline oder liegt unterhalb der U Baseline.

Die erste Eigenschaft bedeutet für den Anbieter, dass er entweder auf alle auftretenden Gewährleistungsmöglichkeiten reagiert, oder alle Gewährleistungsmaßnahmen aufschiebt bis der M Grenzwert unterschritten wurde. Die zweite Eigenschaft bedeutet, dass er entweder auf alle auftretenden Wartungsmöglichkeiten reagiert, oder alle Wartungsmaßnahmen aufschiebt bis der U Grenzwert unterschritten wurde. Die dritte Eigenschaft bedeutet, dass er entweder auf alle auftretenden Erweiterungsmöglichkeiten reagiert, oder die nächste Erweiterung aufschiebt bis der ACSI den optimalen U Grenzwert unterschritten hat.

11.5 Ein Beispiel einer Strategie-Analyse

In diesem Kapitel wird in Abschnitt 11.5.1 ein Beispiel vorgestellt, in Abschnitt 11.5.2 und 11.5.3 werden die entsprechenden Einflussfaktoren und die Baseline-Strategien analysiert. Das Beispiel verwendet die in den vorherigen Kapiteln 11.2, 11.3 und 11.4 erläuterten Maßnahmen und Strategien.

11.5.1 Vorstellung des Beispiels

Das Beispiel wurde jeweils unter dem Kriterium Gesamtgewinn und Durchschnittsgewinn betrachtet. Als interne und externe Einflussfaktoren wurden unten aufgelistete Komponenten verwendet. Es wurden lineare und nicht-lineare Einkommensfunktionen zur Repräsentation verschiedener Software-Märkte gewählt. Um die Empfindlichkeit

des optimalen Gewinns bei Variation der Gewährleistungs-, Wartungs- und Erweiterungskosten zu ermitteln, wurden zwei weitere Experimente durchgeführt. Einmal mit 50% geringeren und einmal mit 50% höheren Kosten. (Die entsprechenden Werte befinden sich in Klammern hinter den Kosten.) Bei den Gewährleistungs-, Wartungs- und Erweiterungskosten handelt es sich um Durchschnittskosten. Das Experiment enthält 8 Kombinationen für den Gewährleistungs-, Wartungs- und Erweiterungsabfall und der Technologieabfall variiert zwischen 2% und 8%. Die Prozentwerte drücken den prozentualen maximalen Wert des ACSI aus.
Ein Markt auf dem Volatilität herrscht, ist durch ein hohes Maß an Wettbewerb, also durch viele Konkurrenten, gekennzeichnet. Um eine Dominanz auf dem Markt zu erhalten, werden ständig neue Funktionalitäten in die Software eingebaut. Somit ist ein Markt auf dem Volatilität herrscht auch durch ein hohes Maß an „jungen“ Produkten gekennzeichnet. Die verwendeten Komponenten des Beispiels sind:

- 2 lineare und 4 nicht-lineare Einkommensfunktionen
- Gewährleistungskosten: $300 ($150, $450)
- Wartungskosten: $1000 ($500, $1500)
- Erweiterungskosten: $10 000 ($5000, $15 000)
- Qualität: hoch (HQ), gering (LQ)
- Wahrscheinlichkeit Gewährleistungsmöglichkeit: 0.2, 0.4
- Wahrscheinlichkeit Wartungsmöglichkeit: 0.1, 0.2
- Wahrscheinlichkeit Erweiterungsmöglichkeit: 0.03, 0.01
- Markt-Volatilität: hoch (HV), gering (LV)
- Gewährleistungsabfall: 2%, 6%
- Wartungsabfall: 6%, 12%
- Erweiterungsabfall: 12%, 20%
- Technologieabfall: 0%, 2%, 4%, 6%, 8%

11.5.2 Analyse des Beispiels

In diesem Abschnitt werden die Ergebnisse des in Abschnitt 11.5.1 eingeführten Beispiels unter dem Durchschnittsgewinn-Kriterium betrachtet. Die Ergebnisse des Gesamtgewinn-Kriteriums werden aus Umfangsgründen nicht angesprochen, können aber aus Sahin und Zahedi [SZ01b] entnommen werden.
Die Abbildung 11.2 zeigt den Durchschnittsgewinn einer linearen Einkommensfunktion. Aus dieser Abbildung ist die Einteilung des Graphen in vier Bereiche zu erkennen. Es handelt sich dabei um die folgenden Bereiche: hohe Qualität - geringe Volatilität

(HQ-LV), hohe Qualität - hohe Volatilität (HQ-HV), geringe Qualität - hohe Volatilität (LQ-HV) und geringe Qualität - geringe Volatilität(LQ-LV).
Die Analyse zeigt, dass die Qualität unter allen Umständen zählt, denn eine höhere Qualität im Design und der Implementierung führt zu einem höheren Durchschnittsgewinn. Dies wird durch Vergleichen der Gewinnwerte der Bereiche HQ-LV und LQ-LV bzw. HQ-HV und LQ-HV deutlich. Diese Tatsache ist aber nicht überraschend, wenn angenommen wird, dass bessere Design- und Implementierungsqualität mit einer geringeren Wahrscheinlichkeit von Gewährleistung- und Wartungsmöglichkeit verbunden wird. Außerdem ist bei einer hohen Qualität der Durchschnittsgewinn unempfindlicher gegenüber einem Kundenverlust, der durch die Gewährleistungs-, Wartungs- und Erweiterungsabfälle ausgedrückt wird. Es ist folglich zu erkennen, dass Qualität zählt und es durch Verwendung des Upgrade-Cycle-Modells möglich ist, die Einwirkung der Qualität auf die Wirtschaftlichkeit des Entwicklers zu messen.

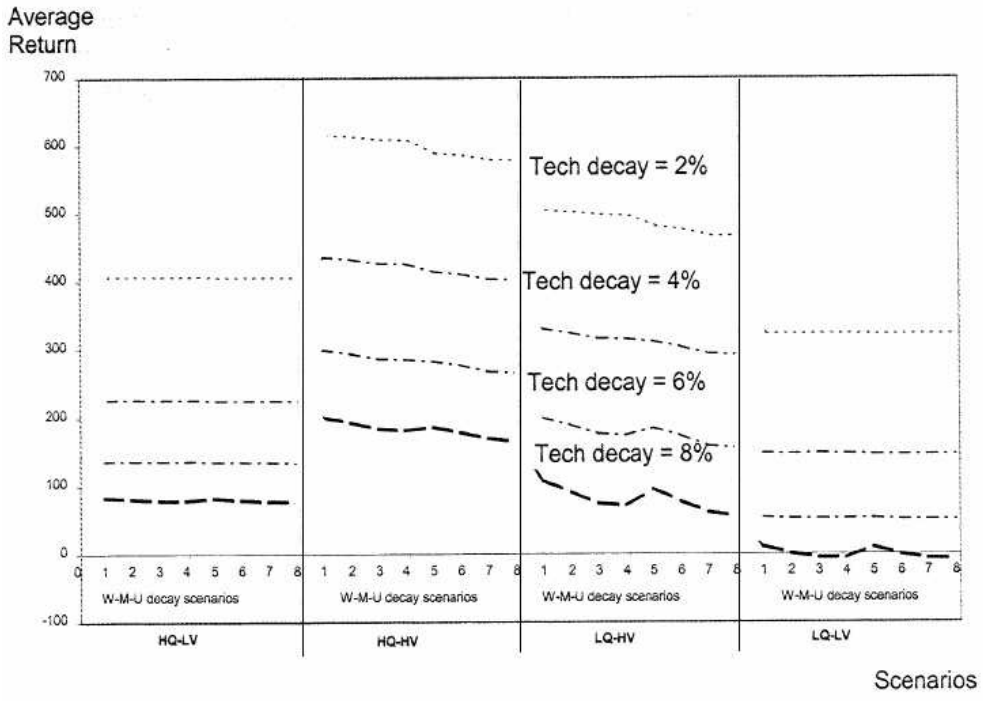


Abbildung 11.2: Der Durchschnittsgewinn einer linearen Einkommensfunktion

Die Abbildung 11.2 verdeutlicht zusätzlich, dass ein eher sprunghafter Markt dem Anbieter mehr Erweiterungsmöglichkeiten bietet. Nutzt der Anbieter diese Möglichkeiten, hat das Auswirkungen auf den Durchschnittsgewinn, denn der Durchschnittsgewinn ist höher. Beim Vergleich der Bereiche HQ-LV, LQ-LV (geringer Volatilität) mit HQ-HV, LQ-HV (höhere Volatilität) ist die Auswirkung auf den Durchschnittsgewinn deutlich zu erkennen. Außerdem ist auf einem Markt mit hoher Volatilität der Gewinn empfindlicher auf einen Qualitätsabfall, weil wesentlich mehr Konkurrenz besteht.
Besteht eine hohe Kundenempfindlichkeit auf technologischen Fortschritt, dann spiegelt sich auch das in einem geringeren Durchschnittsgewinn wieder. Diese Behauptung

wird in Abbildung 11.2 durch die erheblichen Abstände zwischen den vier Kurven mit unterschiedlichen technologischen Abfallfaktoren (von 2% bis 8%) untermauert. Dieses Ergebnis wurde für alle verwendeten Einkommensfunktionen vorgefunden. Ein schnelles Sinken des ACSI in einer Periode begrenzt somit die Wirtschaftlichkeit eines Entwicklers.
Die Analyse hat ergeben, dass eine hohe Qualität die Grundlage für eine Gewinnoptimierung ist. Denn durch hohe Qualität entstehen weniger Gewährleistungs- und Wartungsmöglichkeiten für den Anbieter und er reagiert unempfindlicher gegenüber einem Kundenverlust. Hohe Qualität verschafft dem Anbieter also mehr Marktanteile. Nutzt der Anbieter jedoch nicht die Erweiterungsmöglichkeiten, verliert die Software an Qualität und der Anbieter demzufolge an Gewinn.

11.5.3 Analyse der Baseline-Strategien

Nachdem die einzelnen Einflussfaktoren betrachtet wurden, werden nun die Baseline-Strategien auf ihre Optimalität untersucht. Bei der Durchführung des Experiments [SZ01b] hat sich die WM Baseline als die am häufigsten optimale Strategie herausgestellt. Sie ist für 51% der Beispielfälle optimal. Auf einem Markt mit hoher Volatilität und einer geringen technologischen Veralterung erweist sich die WM Baseline als optimal. Mit einem Anstieg der technologischen Veralterung erweist sie sich aber ebenfalls für einen Markt mit geringer Volatilität als optimal. Die WMU Baseline erweist sich mit 26% als die zweitbeste Strategie. Sie ist unter geringer technologischer Veralterung und geringer Volatilität optimal. Die WM und WMU Baselines erweisen sich also mit 77% als die wirksamsten Strategien. Es ergeben sich laut [SZ01b] folgende Ergebnisse: U-only Baseline (10%), W Baseline (7%), M Baseline (3%) und alle anderen Baselines (3%). Die U-only, W und M Baseline erweisen sich bei hoher technologischer Veralterung und hoher Volatilität als optimal.
Steigt der Technologieabfall an, dann verschieben sich die WMU oder WM Baseline zur W oder M Baseline und anschließend zur U-only Baseline. Bei hoher Volatilität entstehen größere Schübe.
Das ergibt also folgendes Ergebnis: auf einem stabilen Markt (geringer Technologieabfall und geringe Volatilität) sind die WMU oder WM Baseline-Strategien die Optimalsten. Bei einem instabilen Markt (hoher Technologieabfall und hohe Volatilität) werden die WMU oder WM Baseline-Strategien durch die W, M oder U-only Baseline-Strategien abgelöst.

Fazit:
Die Analyse des Beispiels hat die allgemeine Erkenntnis gebracht, dass Qualität in jedem Fall zählt, egal unter welchen Marktbedingungen oder Kundenerwartungen. Außerdem wurde herausgefunden, dass die Markt-Volatilität für die Einnahmen des Anbieters bedeutsam sind. Denn Anbieter, die auf einem Markt mit hoher Volatilität arbeiten, weisen einen höheren Durchschnittsgewinn auf, als jene, die auf einem Markt mit niedriger Volatilität arbeiten. Es ist zu erkennen, dass die Qualität, die Kundenzufriedenheit und die Volatilität in einer starken Beziehung zueinander stehen.
Bei der Bewertung der Baseline-Strategien ist zu erkennen gewesen, dass die Markt-

Volatilität und die technologische Veralterung für eine Optimalitätaussage eine entscheidende Rolle spielen. Denn je nach Konstellation von Volatilität und technologischer Veralterung ergeben sich die optimalen Baseline-Strategien. Es ist beispielsweise nachvollziehbar, dass auf einem instabilen Markt eine Reaktion auf jede Wartungsmöglichkeit nicht ratsam wäre, denn durch das hohe Maß an Konkurrenz ist ein Schwanken des ACSI vorprogrammiert. Bei sofortiger Reaktion auf Wartungsmöglichkeiten leidet irgendwann die Qualität des Produkts, weil keine Zeit (und kein Geld) für Erweiterungen übrig bleibt. Wird hingegen in regelmäßigen Abständen gewartet und erweitert, bleibt die Qualität des Produkts erhalten und eine gewisse Stabilität des ACSI stellt sich ein.
Es gibt viele Einflussfaktoren in der Software-Wartung und es ist zu erkennen gewesen, dass das vorgestellte Modell nur einige davon berücksichtigt. Die Organisation der Wartung wird vom Modell beispielsweise gar nicht betrachtet. Daher wird im folgenden Kapitel 11.6 auf diesen weiteren Einflussfaktor bei der Wartung eingegangen. Abschließend soll nochmal erwähnt werden, dass das Modell nur eine Möglichkeit zur Analyse von Wartungsentscheidungen darstellt.

11.6 Die Organisation der Software-Wartung

In diesem Kapitel wird ein Überblick über die verschiedenen Möglichkeiten der Software-Wartungsorganisation gegeben. Die Wartung wird nicht nur durch die Entscheidungen „Wann soll der Entwickler auf eine Wartungsmöglichkeit reagieren?“ oder „Mit welcher Maßnahme soll der Entwickler auf eine Wartungsmöglichkeit reagieren?“ beeinflusst, sondern auch durch die Infrastruktur der Wartung [HST04]. Mit der Infrastruktur der Wartung ist die Wahl des Wartungsteams gemeint, die eine organisatorische Entscheidung der Wartung darstellt. Das heisst, es muss die Frage geklärt werden, ob das Entwicklungsteam auch als Wartungsteam fungiert oder ob ein separates Wartungsteam für die Wartung beauftragt wird. Am Ende des Kapitels wird versucht, durch eine Gegenüberstellung der Vor- und Nachteile beider Möglichkeiten, eine Entscheidungshilfe für diese Frage zu geben.

Im Unterschied zur Software-Ersterstellung ist die Software-Wartung durch starke Einschränkungen beeinflusst [HST04]. Eine triviale Einschränkung, die jedoch zu der Wichtigsten zählt, ist die Tatsache, dass die Software schon existiert. Somit muss das Wartungsteam darauf achten, dass Änderungen vollständig sind und durch sie nichts funktionierendes fehlerhaft wird [HST04]. Die Wartung ist zudem viel dienstleistungsbezogener als die Ersterstellung, das heisst, eine ständige Kundenbetreuung ist notwendig. Denn durch Kundenbetreuung wird es ermöglicht, eventuelle Fehler und Defizite zu erkennen.

Stellt sich also die Frage, nach welchen Kriterien wird das Wartungsteam zusammengestellt. Da ein enger Zusammenhang zwischen der Produkt und der Organisationsstruktur herrscht [HST04], muss entschieden werden, wie das Wartungsteam strukturiert wird. Bei dieser personellen Entscheidung gibt es zwei Möglichkeiten [HST04]:

- tätigkeitsbezogenes Team: Alle Mitarbeiter in einem Team zusammenfassen, die die gleiche Tätigkeit ausüben.

- teilsystembezogenes Team: Alle Entwickler eines Teilsystems in einem Team zusammenfassen.

In den meisten Fällen wird ein teilsystembezogenes Wartungsteam gewählt, weil den Mitgliedern dort eine Tätigkeitsvielfalt angeboten wird. Somit ist die Motivation der Mitarbeiter höher und die Arbeitsplanung fällt leichter [HST04]. Es besteht aber die Gefahr, dass die Mitarbeiter überfordert sind, weil sie ein breitgefächertes Know-how über viele verschiedene Werkzeuge besitzen müssen. Daher ist es ratsam, starke Querschnittsfunktionen [GT03] zu haben. Unter Querschnittstellen werden die Spezialistenteams für beispielsweise die Architekturgestaltung, das Konfigurationsmanagement oder das Qualitätsmanagement verstanden [HST04, GT03].
Aber auch die Verantwortlichkeiten der Einzelelemente spielt bei der Teamstruktur eine Rolle. Es gibt zwei Zuständigkeitsmodelle:

- **kollektive Zuständigkeit:** „jeder darf jedes Element ändern“
  Die kollektive Zuständigkeit ermöglicht es, eine schnelle Reaktion auf Wartungsmöglichkeiten zu gewährleisten. Denn es muss nicht auf die zuständige Person gewartet werden. Als Folge dessen ergibt sich, dass ein Ändern der Elemente durch nicht-spezialisierte Mitarbeiter die Qualität leidet und die Ausrichtung des Produktes eventuell verändert wird [HST04].

- **individuelle Zuständigkeit:** „jedem Element wird ein Zuständiger zugewiesen“
  Eine Ausprägungsform von der individuellen Zuständigkeit ist die „Chefprogrammiererorganisation“. Es gibt einen Hauptverantwortlichen und die Anderen arbeiten ihm zu [GT03]. Der Vorteil dieser Zentralisierung ist, dass wenig Kommunikationsaufwand betrieben werden muss und außerdem gewährleistet wird, dass die Linie des Produkts eingehalten wird [HST04]. Der Nachteil hingegen ist die Abhängigkeit von Personen. Daher ist es ratsam, die Verantwortung auf mehrere Personen zu verteilen.

Durch das unkontrollierbare Auftreten von Wartungsmöglichkeiten, wird die laufende Arbeit der Mitarbeiter unterbrochen und sie müssen Wartungsmaßnahmen vornehmen. Durch die ständigen Unterbrechungen wird die Produktivität negativ beeinflusst, daher ist es hilfreich das Wartungsteam in Teilteams zu teilen. Ein Team ist für die Instandhaltung (Gewährleistung und Wartung) und das andere für die Weiterentwicklung (Erweiterung) zuständig. Mit diese Entscheidung ergeben sich beispielsweise die Vorteile eines einheitlichen Berichtswesens und einem einheitlichen Konfigurationsmanagements (weitere Vorteile können aus [HST04] entnommen werden). Als Nachteile dieser Entscheidung ergeben sich jedoch beispielsweise 2 Ansprechpartner für jede Funktion (höherer Kommunikationsaufwand) und die Entwickler bekommen den Anreiz, die Änderbarkeit und Testbarkeit weniger zu berücksichtigen (weiter Nachteile in [HST04]).

Die Infrastrukturentscheidung des Unternehmers muss eine Menge Faktoren berücksichtigen. Diese Entscheidung fällt nicht leicht, aber ein Gegenüberstellen der Vor- und Nachteile erleichtert die Entscheidung meistens. Aber neben der Frage „wie soll das Wartungsteam aufgebaut sein?“ (also teilsystembezogen mit kollektiver Zuständigkeit oder anders), stellt sich auch die Frage „bildet das Entwicklerteam das Wartungsteam oder wird ein separates Wartungsteam gebildet?“. Wird das Entwicklerteam auch als Wartungsteam verwendet, ist ein hohes Maß an Know-how geboten, denn die Teammitglieder besitzen ein gutes Wissen über das System. Außerdem werden die Personalkosten klein gehalten [Bom05]. Hingegen wird die Planung der Projekte durch die unkontrollierten Wartungsmöglichkeiten schwierig und es ist eventuell kein genauer Ansprechpartner gegeben. Wird ein separates Wartungsteam eingesetzt, gibt es zwar klare Ansprechpartner, aber das Team besitzt nur das vom Entwicklerteam dokumentierte Wissen über das System. Das Team hat also ein geringeres Know-how über das System. Jedoch kann sich das Team, beispielsweise durch Einsatz von Werkzeugen zur Fehlersuche und zum Testen, ein „Wartungswissen“ aneignen. Zudem steigen die sowieso schon vorhandenen Kosten des Projekts [Bom05]. Es ist auch eine Mischform aus beidem denkbar. Das Team besteht dabei aus einem Wartungsmanager und dem Entwicklerteam. So hilft das Entwicklerteam bei der Wartung, wird aber nicht ständig bei der eigentlichen Arbeit unterbrochen. Zusätzlich ist ein klarer Ansprechpartner vorhanden [Bom05].

11.7 Zusammenfassung

In dieser Ausarbeitung wurde ein Rahmenwerk vorgestellt, dass bei der Analyse von Software-Wartungsentscheidungen behilflich ist. Es verwendet die verschiedenen internen und externen Einflussfaktoren einer Software und erstellt mit Hilfe dieser Faktoren ein Upgrade-Cycle-Modell. Dieses Modell hilft bei der Wahl einer Wartungsmaßnahme. Bei der Verwendung des Upgrade-Cycle-Modells wurde deutlich, dass es verschiedene Wartungsbereiche gibt, nämlich den Gewährleistungs-, Wartungs- und Erweiterungsbereich. In jedem dieser Bereiche werden dem Wartungsmitarbeiter entsprechende Maßnahmen zum Umgang mit einer eintretenden Wartungsmöglichkeit angeboten. Des Weiteren wurden die auf Grund des Upgrade-Cycle-Modells entstehenden Strategien vorgestellt. Das Upgrade-Cycle-Modell bietet 8 verschiedene Baseline-Strategien an. Jede Baseline-Strategie setzt dabei andere Grenzwerte für die drei Wartungsbereiche und liefert somit andere Reaktionsmaßnahmen.

Anhand eines Experiments wurden die vorgestellten Einflussfaktoren und Baseline-Strategien mit Hilfe des Rahmenwerks analysiert. Auch eine Optimalitätsaussage über die Baseline-Strategien wurde vorgenommen und ergab dabei, dass eine WM und WMU Baseline auf einem stabilen Markt die optimalen Strategien sind. Auf einem instabilen Markt hingegen werden sie durch die W, M und U-only Baseline abgelöst.

Aber nicht nur die Analyse der Wartung spielt eine Rolle für das Wartungsmanagement, sondern auch die Infrastruktur der Wartung. Daher wurde in der Ausarbeitung auch auf die verschiedenen Möglichkeiten einer Teamzusammenstellung Stellung ge-

nommen. Es existiert die Möglichkeit einer Teamzusammenstellung nach tätigkeits- oder teilsystembezogenen Aufgaben. Wie aus den jeweiligen Namen schon erkennbar ist, handelt es sich bei einem tätigkeitsbezogenen Team um die Zusammenstellung aller Mitarbeiter, die die gleiche Tätigkeit im Projekt ausüben. Bei einem teilsystembezogenen Team handelt es sich hingegen um die Zusammenstellung aller Mitarbeiter, die an einem Teilsystem arbeiten. Aber nicht nur die Teamzusammenstellung sondern auch die jeweiligen Verantwortlichkeiten sind entscheidend bei der Wartungsorganisation. Hier kann zwischen der kollektiven und individuellen Zuständigkeit gewählt werden. Es wurde festgestellt, dass jedes der beiden Zuständigkeitsmodelle seine Vor- und Nachteile aufweist. Letztendlich ist festzustellen, dass die Wahl der Organisation mit entsprechenden Zuständigkeiten im Ermessen des Unternehmers liegen. Denn je nach finanziellen Möglichkeiten, Produktqualitätsziel und Arbeitsumfeld hängt seine Wahl des Wartungsteams ab.

Das Risikomanagement spielt in der gesamten Softwareentwicklung eine Rolle, denn Risiken sind immer vorhanden. Da viele Risiken sowohl in der Entwicklung als auch in der Wartung auftreten können, ist es nicht sinnvoll das Risikomanagement der beiden Bereiche separat zu betrachten. Einige vielleicht unbewusste Risiken werden beispielsweise durch das Beheben von Implementierungsfehlern oder durch Einführen neuer Funktionalitäten hervorgerufen. Denn bei der Fehlerbehebung könnten neue Fehler erzeugt werden, die ungeahnte Folgen haben. Die Einführung neuer Funktionalitäten und/oder eine Erweiterung der Software könnten von den Anwendern nicht akzeptiert werden, genauso wie neue Technologien. Aber auch die Wahl des Wartungsteams stellt Risiken dar, beispielsweise durch falsche Wahl der Zuständigkeiten. Bei einer kollektiven Zuständigkeit birgt die Behebung von Implementierungsfehlern größere Risiken, denn die Fehlerbehebung von nicht-spezialisierten Mitarbeitern führt eher zu neuen Fehlern als von Spezialisten. Es ist zu erkennen, dass in der Wartung eine Menge Risiken vorhanden sind. Daher ist es notwendig das Risikomanagement in der Software-Wartung einzubinden.

In dieser Ausarbeitung ist deutlich geworden, dass das Software-Wartungsmanagement eine Menge Einflussfaktoren zu berücksichtigen hat und das hier vorgestellte Rahmenwerk eine Möglichkeit zur Analyse bietet. Je nach den Zielen und den Einflussfaktoren werden dem Unternehmer optimale Strategien angeboten. Es wurde aber auch verdeutlicht, dass ein solches Modell niemals alle Einflussfaktoren berücksichtigen kann, wie beispielsweise die Organisation der Wartung. Aber diese Einflussfaktoren lassen sich separat betrachten und beurteilen. Außerdem muss auf das Risikomanagement geachtet werden, denn auch die Wartung birgt eine Menge an Risiken. Das Software-Wartungsmanagement wird also hauptsächlich durch die Ziele des Unternehmers (bspw. Gewinnmaximierung, Erhöhung des Marktanteils) beeinflusst.

Literaturverzeichnis

[Art88] LJ. Arthur. Software evolution: A software maintenance challenge. John Wiley and Sons: New York, 1988.

[Bom05] C. Bommer. Wege aus dem würgegriff der software-wartung. GI-Arbeitskreis Software-Wartung, 04.10.2005.

[Bre94] J. Brecka. The american customer satisfaction index. 27(10), pages 41–44. Quality Progress, 1994.

[GK82] M. Gort and S. Klepper. Time path in the diffusion of product innovations. 92(367), pages 630–653. Economic Journal, 1982.

[Gri57] Z. Griliches. Hybrid corn: An exploration in the economics of technological change. 25, pages 501–522. Econometrica, 1957.

[GT03] P. Grubb and AA. Takang. Chapter 10 - management and organisational. In *Software Maintenance: Concepts and Practice*, 2, pages 204–217. World Scientific Publishing Company, 2003.

[HST04] M. Hasitschka HM. Sneed and MT. Teichmann. Kapitel4 - aufbauorganisation. In *Software-Produktmanagement*, pages 111–119. Dpunkt. Verlag GmbH, 2004.

[KS99] CF. Kemerer and S. Slaughter. An empirical study approach to studying software evolution. 25(4), pages 493–509. IEEE Transactions on Software Engineering, 1999.

[PS94] TJ. Prusa and JA Jr. Schmitz. Can companies maintain initial innovation thrust? a study of the pc software industry. 76(3), pages 523–540. The Review of Economics and Statistics, 1994.

[Reu81] J. Reutter. Maintenance is a management problem and a programmer´s opporunity. 50, pages 343–347. AFIPS Press: Reston VA, 1981.

[Swa76] EB. Swanson. The dimension of maintenance. pages 492–497. IEEE Computer Society Press: Long Beach CA, 1976.

[SZ00] I. Sahin and FM. Zahedi. Optimal policies under risk for changing software systems based on customer satisfaction. 123(1), pages 175–194. European Journal of Operational Research, 2000.

[SZ01a] I. Sahin and FM. Zahedi. Control limit policies for warranty, maintenance, and upgrade of software systems. 33(9), pages 729–745. IIE Transactins, 2001.

[SZ01b] I. Sahin and FM. Zahedi. Policy analysis for warranty, maintenance, and upgrade of software systems. 13, pages 469–493. Journal of Software Maintenance and Evolution: Research and Practice, 2001.

Kapitel 12

Wartungsprozesse

Sebastian Rüsche

Inhaltsverzeichnis

12.1 Einführung

Diese Ausarbeitung beschäftigt sich zunächst mit Prozessen, wie sie in der Softwareentwicklung und Softwarewartung eingesetzt werden. In den darauf folgenden Abschnitten beschäftigt sie sich mit so genannten Reifegradmodellen oder Maturity Models. Diese dienen der Verbesserung, der Reife, von Prozessen und geben daher einen guten Ansatz, was für die einzelnen Prozesse von besonderer Bedeutung ist.

Hierzu werden zunächst die Grundlagen der Reifegradmodelle, wie sie in der Softwareentwicklung eingesetzt werden, erläutert und an Hand eines konkreten Beispiels, des *Capability Maturity Model integration* (CMMi), veranschaulicht. Diese Reifegradmodelle in der Softwareentwicklung sind der Ausgangspunkt für die Entwicklung der anschließend vorgestellten Reifegradmodelle in der Softwarewartung.

Im 3. Kapitel wird dann ausführlicher auf Reifegradmodelle für die Softwarewartung eingegangen. Hier werden zwei Beispiele, das *Software Maintenance Maturity Model* (SM^{mm}) und das *Corrective Maintenance Maturity Model* (CM^3), vorgestellt. Der Schwerpunkt liegt hierbei allerdings auf dem Software Maintenance Maturity Model.

12.2 Prozessmodelle in der Softwarewartung

Mit immer mehr bereits verfügbarer und immer komplexer werdender Software wächst die Notwendigkeit, diese Software zu warten, heute immer weiter an. Daher ist es auch notwendig, diese Prozesse der Softwarewartung in irgendeiner Weise zu organisieren, um sowohl die Qualität der Wartung als auch ihre Kosten im Griff zu behalten.

Zu diesem Zweck gibt es heute einige Prozessmodelle, mit deren Hilfe ein Wartungsprozess für die jeweilige Organisation definiert werden kann. Ein solcher Prozess hat, richtig umgesetzt, den Vorteil, dass jedem Mitarbeiter die in einer bestimmten Situation folgenden Schritte bekannt sind. Die Gefahr, z. B. wichtige Arbeitsschritte einfach zu vergessen, sinkt, wodurch die Qualität des Produktes steigt. Auch ermöglicht ein solcher Prozess dem Management von vornherein eine gewisse Abschätzung, wie viel Zeit und Geld eine bestimmte Wartungsaktivität kosten wird.

Einer der heute verbreiteten Wartungsprozesse ist der Prozess nach dem Standard IEEE 1219. Dieser Standard beschreibt die Softwarewartung nach der Auslieferung mit dem folgenden Wartungszyklus:

Der Prozess wird durch eine Änderungsanfrage eines Benutzers angestoßen. Diese Anfrage wird dann von der Wartungsorganisation erfasst und einer Kategorie zugeordnet. Der Standard definiert hierzu die folgenden drei Kategorien:

Korrigierend Es wird ein Fehler in der Software beseitigt.

Anpassend Die Software wird an veränderte Rahmenbedingungen angepasst, z. B.

eine andere Gesetzesgrundlage oder neue Hardware

Perfektionierend Die Software wird um Funktionen erweitert.

Anschließend erfolgt eine genauere Analyse der Anfrage, ob sie umsetzbar ist und welcher Aufwand mit einer Umsetzung verbunden ist. Nach dieser Analyse entscheidet die Wartungsorganisation ob die Anfrage akzeptiert wird, oder nicht. Eine größere Funktionserweiterung der Software kann z. B. auf das nächste Release verschoben werden.

Auf die Analyse folgen Design und schließlich die Implementierung der Änderung. Mit Hilfe von Systemtests wird anschließend überprüft, ob die Änderung keine unerwünschten Nebenwirkungen mit sich gebracht hat. Sind diese Tests erfolgreich, wird die veränderte Software, nach einem letzten Akzeptanztest durch den Kunden, ausgeliefert.

Um nun genauer zu verstehen, auf welche Aspekte es bei der Umsetzung solcher Prozessmodelle in der Softwarewartung genau ankommt, bietet sich die Betrachtung von Reifegradmodellen an. Diese Modelle beurteilen an Hand verschiedener Gesichtspunkte einen Prozess nach der Reife, die er bereits erlangt hat und geben Anhaltspunkte, wie er verbessert werden kann. Die Kriterien, nach denen die Bewertung erfolgt, können gleichzeitig als Kernpunkte der Prozesse in der Softwarewartung angesehen werden. Da die Reifegradmodelle in der Softwarewartung oftmals aus Modellen in der Softwareentwicklung entstanden sind, soll aber zunächst das Prinzip der Reifegradmodelle in der Softwareentwicklung veranschaulicht werden.

12.3 Reifegradmodelle in der Softwareentwicklung

12.3.1 Grundlagen

Mit dem immer weiter steigenden Wettbewerb im Bereich der Softwareentwicklung wird es für Softwareunternehmen notwendig, ihre Prozesse bei der Entwicklung neuer Software immer weiter zu verbessern, um im Wettbewerb bestehen zu können. Ein Hilfsmittel, diese Verbesserung der Prozesse zu erreichen sind Reifegradmodelle. Mit Hilfe eines Reifegradmodells werden die Prozesse des Unternehmens untersucht und bewertet.

Hierzu werden meist Prozessbereiche (PA, Process Area) betrachtet, also keine einzelnen Prozesse. Diese Bereiche müssen bestimmte im Modell beschriebene Anforderungen erfüllen. Diese Anforderungen sind jeweils für jeden Reifegrad festgelegt und bauen meist aufeinander auf.

Nach einer Bewertung geben die Gründe, warum der nächst höhere Reifegrad nicht erreicht wurde, Ansatzpunkte für eine Verbesserung der Prozesse. Das Ergebnis der

Bewertung kann außerdem verwendet werden, um Wartungsorganisationen miteinander zu vergleichen.

Als Beispiel für ein solches Reifegradmodell soll im folgenden Abschnitt das Capability Maturity Model integration (CMMi) vorgestellt werden.

12.3.2 CMMi

Das CMMi wurde vom Software Engineering Institute der Carnegie Mellon Universität entwickelt. Das CMMi enthält Modelle für verschiedene Disziplinen, unter anderem Softwareentwicklung.

Zu jedem Modell gibt es zwei verschiedene Repräsentationen, kontinuierlich (continuous) und abgestuft (staged). In der kontinuierlichen Repräsentation werden die einzelnen Prozessbereiche einzeln bewertet und können somit auch einzeln in einer, mehr oder weniger, beliebigen Reihenfolge verbessert werden. Außerdem ist diese Repräsentation anderen Standardmodellen, wie EIA/IS 731 oder ISO/IEC 15504 sehr ähnlich, so das eine Umstellung von diesen Modellen auf die kontinuierliche Repräsentation des CMMi einfacher ist.

In der abgestuften Repräsentation werden die Prozesse als Ganzes beurteilt. Es wird dadurch eine Reihenfolge der Verbesserungen in den einzelnen Bereichen vorgegeben. Dies kann besonders dann von Vorteil sein, wenn das Unternehmen noch nicht allzu viel Erfahrung mit Prozessen und ihrer Verbesserung hat. Außerdem ist diese Repräsentation dem Vorgängermodell Software-CMM sehr ähnlich, was eine Umstellung erleichtert.

Es werden zunächst beide Repräsentationen vorgestellt. Im Hinblick auf den Schwerpunkt dieser Ausarbeitung, nämlich das Software Maintenance Maturity Model, wird aber nur die abgestufte Repräsentation näher erläutert, da diese eine der Grundlagen für das Software Maintenance Maturity Model bildet. Es wird immer das Modell für die Softwareentwicklung betrachtet [Ins02].

12.3.2.1 CMMi für Softwareentwicklung

Um die Reife der Softwareentwicklung in einer Organisation zu beurteilen und eventuelle Schwächen aufzuzeigen, werden beim CMMi sogenannte Prozessbereiche betrachtet. Alle ablaufenden Prozesse werden in solche Prozessbereiche eingeteilt. Das Modell für die Softwareentwicklung enthält insgesamt 25 Prozessbereiche, z. B. Organizational Training, Requirements Management oder Process and Product Quality Assurance. Zu jedem Prozessbereich werden außerdem Ziele definiert, die erreicht werden sollten, um über möglichst gute Prozessabläufe in diesem Bereich zu verfügen.

Kontinuierliche Repräsentation Bei der kontinuierlichen Repräsentation werden für jeden Prozessbereich sechs *Capability Level* definiert. Jedes Capability Level ist im Prinzip eine Teilmenge der für den Prozessbereich definierten Ziele. Anschließend wird geprüft, auf welchem dieser Level sich die vorhandenen Prozesse in diesem Bereich innerhalb des Unternehmens befinden. Das Unternehmen kann dann entscheiden, in welchen Bereichen zunächst mit Verbesserungen begonnen werden soll. Die Gründe für die jeweilige Einstufung geben dabei Anhaltspunkte für die Verbesserungen.

Die sechs definierten Capability Level sind:

Incomplete Es ist entweder kein den Anforderungen von *Performed* entsprechender Prozess zur Softwareentwicklung vorhanden, oder er wird nicht vollständig ausgeführt.

Performed Es ist ein grundlegender Prozess vorhanden und dieser wird zur Softwareentwicklung eingesetzt.

Managed Die Durchführung des Prozesses wird geplant und der tatsächliche Stand der Entwicklung immer wieder mit dem erstellten Plan abgeglichen. Falls nötig werden entsprechende Maßnahmen ergriffen, um Plan und Wirklichkeit wieder in Einklang zu bringen.

Defined Es sind organisationsweite Prozesse definiert, die als Ausgangspunkte für jedes Projekt verwendet werden. Hierzu werden nur noch leichte Anpassungen vorgenommen.

Quantitatively Managed Qualität und Effizienz der Prozesse wird mit Hilfe statistischer Methoden ermittelt und bewertet.

Optimizing Das Hauptaugenmerk liegt auf der kontinuierlichen Verbesserung der Prozesse. Die zur Verfügung stehenden Daten und Ressourcen werden genutzt, um den Prozess immer weiter zu verbessern.

Abgestufte Repräsentation Bei der abgestuften Repräsentation werden insgesamt fünf Reifegrade festgelegt. Jeder dieser Reifegrade enthält einen Teil der Prozessbereiche. Eine Organisation hat den entsprechenden Reifegrad erreicht, wenn es alle Ziele der Prozessbereiche erreicht hat, die zu diesem oder einem niedrigeren Reifegrad gehören. Eine Organisation muss also erst Reifegrad 3 erreicht haben, um Reifegrad 4 erreichen zu können.

In diesem Fall nimmt das Modell dem Unternehmen also größtenteils die Entscheidung ab, welche Bereiche zuerst verbessert werden sollten, da die Reihenfolge bereits durch die Gruppierungen zu den einzelnen Reifegraden vorgegeben ist.

Die fünf definierten Reifegrade sind:

Initial Die Prozesse werden in keiner Weise kontrolliert und sind daher auch nicht vorhersagbar.

Managed Die Prozesse sind jeweils für ein Projekt definiert, es findet also eine gewisse Kontrolle statt.

Defined Die Prozesse sind unternehmensweit definiert (es finden nur kleinere Anpassungen an die einzelnen Projekte statt).

Quantitatively Managed Die Prozessabläufe werden kontrolliert und gemessen. Es werden also Daten erhoben, mit denen z. B. zukünftig die Prozesse besser vorhergesagt werden können.

Optimizing Die erhobenen Daten werden auch immer wieder zur Verbesserung der Prozesse eingesetzt.

Um nun z. B. den Reifegrad Managed (2) zu erreichen, müssen in sieben Prozessbereichen die geforderten Ziele erfüllt werden. Diese Prozessbereiche sind: Requirements Management, Project Planning, Project Monitoring and Control, Supplier Agreement Management, Measurement and Analysis, Process and Product Quality Assurance und Configuration Management. Zu jedem dieser Prozessbereiche müssen nun die Anforderungen erfüllt werden. Eine Anforderung aus dem Bereich Project Planning ist z. B., dass Schätzungen über Aufwand und Kosten des Produkts aufgestellt werden.

Da diese Repräsentation immer nur einen Teil der Prozesse betrachtet, ist die letztendliche Bewertung etwas ungenauer. Dafür dürfte diese Variante aber auch einfacher einzusetzen sein, da oftmals nur einen Teil der Prozessbereiche zu betrachten braucht. Außerdem liefert das Modell direkt eine in den meisten Fällen sinnvolle Reihenfolge der Verbesserungen.

12.4 Reifegradmodelle in der Softwarewartung

Im Folgenden geht es nun darum, die vorgestellten Konzepte auch in der Softwarewartung einzusetzen. Die Modelle selbst lassen sich leider nicht immer direkt übertragen.

12.4.1 Motivation

Nachdem nun ein Modell für die Softwareentwicklung vorgestellt wurde, stellt sich die Frage, wie es mit dem Hauptthema, der Softwarewartung, aussieht. Das vorgestellte Modell CMMi lässt sich zwar in einem begrenzten Rahmen auch für die Wartung einsetzen, wartungsspezifische Aspekte werden dann aber nur am Rande oder gar nicht beachtet. Durch dieses Fehlen wartungsspezifischer Aspekte wird die Situation der Organisation unter Umständen im Hinblick auf die Softwarewartung falsch bewertet.

Der Software Engineering Body of Knowledge (SWEBOK) [AM04] gibt eine ganze Reihe solcher wartungsspezifischer Tätigkeiten an. Zu den wichtigsten gehören:

Transition Als Transition bezeichnet man den Prozess, mit dem ein Softwareprodukt von den Entwicklern an die Wartungsmitarbeiter übergeben wird.

Wartungsverträge Für die Wartung eines Softwareprodukts werden mit den Kunden in der Regel Wartungsverträge abgeschlossen. In diesen ist festgelegt, welche Änderungen von der Wartungsorganisation im Rahmen der Wartung durchgeführt werden müssen, in welcher Zeit die Änderungen erfolgen müssen, etc. Diese Verträge müssen bei der Planung der Wartungsaktivitäten berücksichtigt werden.

Help Desk Um die Wartung effizient durchführen zu können, wird in den meisten Fällen ein Help Desk eingerichtet, der im Idealfall den einzigen Kontakt der Kunden zur Wartungsorganisation darstellt. Der Help Desk erfasst dann die Problemberichte, priorisiert sie und leitet sie an die entsprechenden Stellen weiter.

Entscheidung über Änderungsanträge Es muss ein Verfahren geben, dass darüber entscheidet, ob ein Änderungsantrag angenommen oder abgelehnt wird. Auf diesen Entscheidungsprozess haben z. B. auch die Wartungsverträge Einfluss.

Auswirkungsanalyse Bevor mit Veränderungen an der Software begonnen wird, muss ermittelt werden, welche Auswirkungen diese Änderungen auf die anderen Teile der Software haben. Das Ergebnis dieser Analyse kann auch Einfluss darauf nehmen, ob ein Änderungsantrag angenommen oder abgelehnt wird.

Da diese Aktivitäten in den Reifegradmodellen für die Softwareentwicklung kaum auftreten, sind neue oder zumindest erweiterte Modelle notwendig. Hierzu wird im Folgenden die Softwarewartung noch einmal genauer spezifiziert. Anschließend wird das Software Maintenance Maturity Model, welches speziell für die Softwarewartung entwickelt wurde, vorgestellt und auch noch ein Blick auf das Corrective Maintenance Maturiy Model geworfen.

12.4.2 Grundlagen

Bei Softwarewartung unterscheidet man zunächst zwischen vier Kategorien der Wartung [AM04]:

Korrigierende Wartung Von den Anwendern werden Fehler gemeldet, die dann korrigiert werden.

Präventive Wartung Durch z. B. Refactoring-Maßnahmen wird die zukünftige Wartung und Fehlersuche erleichtert.

Adaptive Wartung Das Produkt wird an veränderte Anforderungen angepasst (z. B. neue Hardware, neue rechtliche Rahmenbedingungen).

Perfektionierende Wartung Das Produkt wird um Funktionen erweitert, die Bedienung wird vereinfacht, etc.

	Korrigierend	Verbessernd
Reaktiv	Korrigierend	Anpassend
Aktiv	Präventiv	Perfektionierend

Tabelle 12.1: Wartungskategorien

Die Kategorien können allgemeiner in jeweils zwei Gruppen, wie in Tabelle 12.1, zusammengefasst werden.

Unabhängig von der Kategorisierung treten bei der Wartung, wie bereits erwähnt, Prozesse auf, die während der normalen Softwareentwicklung nicht ablaufen. Dies sind z. B. die Transition des Produkts, also die Übergabe des Produkts von den Entwicklern and die Wartungsmitarbeiter, Prozesse zur Entscheidung, ob ein Änderungswunsch akzeptiert wird, oder die Beachtung der mit den Kunden abgeschlossenen Wartungsverträge.

Einer der Hauptpunkte, der die Wartung von der Entwicklung unterscheidet, ist aber die Verwaltung der Benutzeranfragen bzw. Änderungswünsche. Hier ist es, auch im Hinblick auf die Wartungsverträge, notwendig, dass die Anfragen schnell registriert, priorisiert und an die entsprechenden Mitarbeiter weitergegeben werden.

Diese Unterschiede zwischen Softwareentwicklung und -wartung gehen in die folgenden Modelle mit ein, so dass hier eine bessere Beurteilung und Weiterentwicklung der Wartungsprozesse möglich wird.

12.4.3 SM^{mm}

Ein Reifegradmodell, welches nun versucht, die Reife der Wartungsprozesse in einer Organisation zu beurteilen, ist das *Software Maintenance Maturity Model* (SM^{mm}) von Alain April, Jane Huffman Hayes, Alain Abran und Reiner Dumke. Das SM^{mm} basiert unter anderem auf dem bereits für die Softwareentwicklung vorgestellten Modell CMMi, allerdings ist SM^{mm} im Gegensatz zu CMMi ausschließlich auf die Wartung ausgelegt [AHAD05].

12.4.3.1 Das Modell

Um nun die Reife der Wartungsprozesse beurteilen zu können, werden diese bei SM^{mm}, ähnlich wie bei CMMi, in Prozessbereiche (KPAs, Key Process Areas) eingeteilt. In diesen Bereichen gibt es dann für jeden Reifegrad entsprechende, in der Regel aufeinander aufbauende Anforderungen. Das Modell identifiziert für die Softwarewartung insgesamt 18 Prozessbereiche, welche in 4 Prozess-Domänen zusammengefasst sind.

1. **Prozess Management**: Diese Domäne enthält alle KPAs, die sich hauptsächlich mit der Verwaltung der Softwarewartungsprozesse befassen:

Maintenance Process Focus Hierzu gehören die Planung der Wartungsaktivitäten und die Festlegung der Zuständigkeiten.

Maintenance Process/Service definition Dokumentation und Standardisierung der verwendeten Prozesse, außerdem die Anpassung der Prozesse und Dienste bei Bedarf.

Maintenance Training Dieser Teil umfasst sowohl das Training von neuen Mitarbeitern, wie auch Weiterbildungen und auch die Schulung von Benutzern an der Software. Letzteres kann helfen, den Help Desk deutlich zu entlasten.

Maintenance Process Performance Um eine gewisse Vorhersagbarkeit zu erreichen, sollten Maße für die Wartungsprozesse definiert, diese Maße angewendet und Vorhersagemodellen auf Grund der ermittelten Daten erstellt werden.

Maintenance Innovation and Deployment In diesen Teil fallen die Suche nach Verbesserungen und deren Test in Pilotprojekten, sowie die anschließende Einführung der Verbesserungen und eine Überprüfung, ob die Veränderungen wirklich den gewünschten Erfolg gebracht haben.

2. **Anfragen Management**: Diese Domäne enthält alle KPAs, die sich mit der Verwaltung eingehender Wartungsanfragen, wie z. B. Fehlerberichten oder Änderungswünschen, beschäftigen:

   Event and Service Request Management Hierzu gehören die Verwaltung der eingehenden Änderungsanfragen, sowie die Verwaltung der Kontakte zu den Kunden.

   Maintenance Planning Dieser Teil bezeichnet die längerfristige Planung der Wartungsaktivitäten, die Planung des Übergangs des Produkts in die Wartungsphase und die Planung der Versionen und Updates, aber auch Notfallpläne (*disaster recovery*).

   Monitoring and Control of Service Requests and Events Hierunter versteht man die Nachverfolgung der Aktivitäten mit Blick auf den definierten Prozess, sowie den anschließenden Rückblick, aus dem sich Verbesserungen für den Prozess ergeben können.

   Service Level Agreements and Supplier Agreements Dieser Teil beinhaltet die Erstellung und den Abschluss von Wartungs- und anderen Verträgen, sowie die damit verbundenen Aktivitäten, wie z. B. das erstellen von Rechnungen.

3. **Evolution Engineering**: Diese Domäne enthält alle KPAs, die sich mit der Weiterentwicklung und Verbesserung der Software beschäftigen:

   Pre-Delivery and Transition Dieser Teil beinhaltet die Aktivitäten zur Überwachung und Durchführung der Übergabe des Produkts von den Entwicklern an die Wartungsmitarbeiter, z. B. ein entsprechender Wissenstransfer oder auch die gemeinsame Durchführung von Tests.

   Operational Support Dieser Teil bezeichnet die Unterstützung des Kunden, auch außerhalb der normalen Geschäftszeiten.

Evolution and Correction In diesen Teil fallen die Aktivitäten bei der Veränderung von Software, nämlich die Umsetzung, Tests und die Dokumentation der Änderungen.

Verification and Validation Die durchgeführten Änderungen werden mit Hilfe der Aktivitäten aus diesem Bereich überprüft und es werden Akzeptanztests durchgeführt. Nach bestandenen Tests werden die Änderungen dann in die Produktion übernommen.

4. **Unterstützung des Evolution Engineering**: Diese Domäne enthält alle KPAs, die als unterstützende Prozesse des Evolution Engineering angesehen werden können, wie z. B. Qualitätssicherung oder ein Konfigurations-Management:

   Configuration Management Hierunter versteht man die Verwaltung der einzelnen Änderungen im Bezug auf verschiedenen Versionen.

   Process and Product Quality Assurance In diesen Bereich fallen Aktivitäten zur objektiven Bewertung des Produkts und der Prozesse, z. B. durch Metriken, sowie das Aufzeigen von Fällen, in denen sich das Produkt nicht konform zu den Spezifikationen verhält und die Nachverfolgung der Behebung dieser Fehler.

   Measure and Analysis of Maintenance In diesem Teil werden Maße für die Prozesse definiert, Daten gesammelt und anschließend ausgewertet. Die Ergebnisse werden dann den Mitarbeitern mitgeteilt.

   Casual Analysis and Problem Resolution Es wird nach den Ursachen von Fehlern gesucht, diese analysiert und anschließend werden Lösungen vorgeschlagen, wie diese Fehler in Zukunft vermieden werden könnten.

   Rejuvenation, Migration and Retirement Zu diesem Teil gehören Aktivitäten, wie Redokumentation oder Restrukturierung der Software, aber auch das Ausserbetriebnehmen der Software.

Für jeden dieser KPAs sind nun Ziele für jeden Reifegrad definiert. Das Erreichen dieser Ziele ist gleichbedeutend mit dem Erreichen des entsprechenden Reifegrades. Im Rahmen des SM^{mm} wurden 443 solche Ziele bzw. Vorgehensweisen beschrieben. Einige davon werden im Beispiel im nächsten Abschnitt vorgestellt.

Im SM^{mm} sind die folgenden Reifegrade definiert:

Performed Bei einem Unternehmen dieses Reifegrades erfolgt die Wartung mehr oder weniger unkoordiniert. Die Ergebnisse funktionieren zwar in der Regel, es ist aber im Voraus kaum abzuschätzen, wie gut das Ergebnis wird und wie lang man dafür brauchen wird. Die eingehenden Wartungsanfragen werden irgendwie abgearbeitet.

Managed Bei einem Unternehmen dieses Reifegrades werden zumindest die eingehenden Anfragen zentral registriert und priorisiert. Es können zumindest grobe Aussagen z. B. über die Dauer der Wartungsarbeiten getroffen werden.

Established Es gibt einen definierten Prozess, nachdem Wartungsarbeiten ausgeführt werden. Dieser Prozess ist, in einem gewissen Anpassungen an die jeweiligen Projekte, organisationsweit gleich.

Predictable Die zu erwartende Dauer und die zu erwartenden Kosten können abgeschätzt werden. Dieser Reifegrad ist schon relativ schwierig zu erreichen.

Optimizing Das Unternehmen versucht ständig die Wartungsprozesse zu verbessern.

Die Zuordnung zu einem dieser Reifegrade erfolgt durch die in den einzelnen der zuvor vorgestellten KPAs erreichten Ziele. Zumindest der Reifegrad Optimizing dürfte in der Realität allerdings eher selten anzutreffen sein, da ein Erreichen dieses Grades einen recht hohen Einsatz erfordert.

12.4.3.2 Beispiel

Im Folgenden werden einige Beispiel-Praktiken beschrieben, die eine Bewertung des Prozessbereichs „Event and Service Request Management“ verdeutlichen sollen. In diesem Prozessbereich geht es, wie bereits kurz erwähnt, um die Verwaltung von Benutzeranfragen und anderen Ereignissen. Im Idealfall, gibt es in der Wartungsorganisation einen zentralen Kontakt, über den alle Anfragen abgewickelt werden. Von diesem Kontakt werden die Anfragen dann direkt priorisiert und an die entsprechenden Mitarbeiter weitergeleitet. Die Praktiken beschreiben die Reifegrade 1 bis 3, also Performed bis Established.

Die Praktiken sind im Format Req1.r.p nummeriert, wobei Req1 ein eindeutiger Bezeichner für den KPA ist, r ist der Reifegrad, dem die Praktik zugeordnet ist und p ist eine fortlaufende Nummer innerhalb des KPA und des Reifegrads.

Performed Bei einer Organisation mit Reifegrad Performed werden Anfragen von den einzelnen Wartungsmitarbeitern persönlich und informell abgearbeitet. Es gibt keine standardisierten Verfahren für die Annahme und Aufzeichnung von Anfragen. Im SM^{mm} sind hierfür folgende Praktiken beschreiben:

Req1.1.1 Anfragen und Ereignisse werden informell verwaltet

Req1.1.2 Ein individueller Ansatz zur Verwaltung von Benutzeranfragen und Ereignissen basiert hauptsächlich auf der persönlichen Beziehung zwischen dem Wartungsmitarbeiter und dem Benutzer.

Managed Bei diesem Reifegrad werden die Anfragen durch eine einzige Stelle bearbeitet. Sie werden registriert, kategorisiert und einer Priorität zugeordnet. Die so zentral anfallenden Daten können genutzt werden um die Wartungskosten zumindest grundlegend zu dokumentieren. Einige der verwendeten Praktiken sind:

Req1.2.1 Es gibt eine zentrale Stelle, an der die Benutzer direkte Unterstützung erhalten.

Req1.2.2 Änderungswünsche, Problemberichte, etc. werden registriert und von den Wartungsmitarbeitern als Arbeitsanweisung verwendet.

Established Bei diesem Reifegrad laufen die Wartungsprozesse weitgehend standardisiert ab. Dies führt zu einer effizienten und teilweise auch vorhersagbaren Abarbeitung der Wartungsanfragen. Einige der verwendeten Praktiken sind:

Req1.3.1 Den Benutzern stehen mehrere Möglichkeiten zur Verfügung Unterstützung zu erhalten.

Req1.3.4 Es gibt einen definierten Entscheidungsprozess, wie Anfragen weiter zu bearbeiten sind, ob sie beispielsweise angenommen, weiter analysiert oder abgelehnt werden.

Req1.3.7 Die Pflicht- und optionalen Felder der Formulare für Benutzeranfragen sind (organisationsweit) standardisiert.

Die entsprechenden Praktiken für die Reifegrade 4 und 5, also Predictable und Optimizing, wurden im Paper leider nicht veröffentlicht. Dies sollte erst nach eine Absprache mit den Sponsoren geschehen.

12.4.3.3 Zusammenfassung und Praxis

Das Software Maintenance Maturity Model scheint die bei der Softwarewartung ablaufenden Prozesse zu einem großen Teil abzudecken. Die Definitionen der einzelnen Praktiken und auch der Aufbau des Modells sind, meiner Meinung nach, recht gut verständlich, wodurch das ganze Modell auch ohne allzu großen Aufwand in einem Unternehmen eingesetzt werden kann. Da es eine recht große Ähnlichkeit zu Modellen wie CMMi aufweist, dürfte seine Nutzung in Unternehmen, die CMMi bereits als Reifegradmodell für die Softwareentwicklung einsetzen, leicht umzusetzen sein.

Die vorliegende Version 2.0 des Modells wurde zwischen 1995 und 2005 dreimal mit Hilfe von verschiedenen Unternehmen, zumindest bei den ersten beiden Validierung hauptsächlich Telekommunikationsunternehmen, validiert. Zwischen den Validierungen wurden auch immer wieder andere Standards in die Weiterentwicklung mit einbezogen.

Bei den Tests 2001 wurden bei zwei der drei Unternehmen die Wartungsabläufe mit dem Reifegrad 1 bewertet, die des dritten Unternehmens mit Reifegrad 2.

12.4.4 CM^3

Ein weiteres Reifegradmodell ist das Corrective Maintenance Maturity Model. Es wurde hauptsächlich von Mira Kaijko-Mattsson entwickelt. Wie der Name schon vermuten lässt, beschäftigt sich dieses Modell ausschließlich mit dem Bereich der korrigierenden Wartung. Die Autorin ist der Ansicht, dass die einzelnen Bereiche der Wartung,

also korrigierende, perfektionierende Wartung, etc., zu unterschiedlich sind, um sie alle mit einem Modell abzudecken [KM01].

12.4.4.1 Das Modell

Das CM^3-Modell ist ähnlich den beiden bereits vorgestellten Modellen aufgebaut. Es gibt sieben Kernbereiche:

- Prozesse vor der Auslieferung/einem neuen Release
- Transition/Auslieferung
- Problem-Management
- Testen
- Dokumentation
- Upfront Maintenance
- Schulung und Training der Wartungsmitarbeiter

Die in diesen Bereichen vorhandenen Prozesse sind aber nicht einem Reifegrad zugeordnet, sondern es gibt verschiedene Ausprägungen des jeweiligen Prozesses, die jeweils einem der drei definierten Reifegrade *Initial*, *Defined* oder *Optimal*, entsprechen. Diese Unterteilung der Prozesse beruht auf der Annahme, dass ein Prozess nicht während des Übergangs von einem Reifegrad zum nächsten vollständig ausreifen kann.

Die drei Reifegrade sind folgendermaßen definiert:

Initial Es gibt keinen definierten Prozess. Eine erfolgreiche Wartung, falls vorhanden, beruht in diesem Fall nicht zuletzt auf der Kompetenz und den Überstunden der Wartungsmitarbeiter.

Defined Bei diesem Reifegrad gibt es bereits einen definierten Prozess, der auch eingehalten wird. Der Prozess gibt die grundlegenden Schritte bei der Wartung vor. Die Dokumentation des Prozesses wird bei Änderungen oder Neuerungen ebenfalls aktualisiert. Außerdem wird der Ablauf des Prozesses kontrolliert.

Optimal Der Prozess wird laufend optimiert. Erfahrungen werden in den Prozess mit eingebunden.

In Zukunft soll das Modell CM^3 in ein umfassenderes Modell oder eine Sammlung von Modellen eingebunden werden, die alle Bereiche der Wartung abdeckt. Diese soll dann M^3 genannt werden.

12.4.5 Vergleich der beiden Modelle

Beide Modelle beschäftigen sich mit der Einführung und Verbesserung von Wartungsprozessen. Das Software Maintenance Maturity Model betrachtet dabei alle Bereiche der Wartung, während sich das Corrective Maintenance Maturity Model auf den Bereich der korrigierenden Wartung beschränkt.

Das CM^3 gibt hierbei für die einzelnen Prozessebereiche innerhalb dieses Wartungsbereichs Anhaltspunkte, wie dieser zu verbessern ist. Die Reihenfolge, in der die Prozessbereiche verbessert bis zu welchem Reifegrad werden sollten, wird allerdings nicht vorgegeben. Dies kann ein Vorteil sein, wenn sich die Organisation ausreichend mit den Verbesserungen auseinandersetzt, da dann zunächst die Prozessbereiche, die für das Unternehmen am wichtigsten sind, verbessert werden können. Andernfalls birgt diese Vorgehensweise allerdings auch die Gefahr, dass Prozessbereiche, wie z. B. die Schulung von Wartungsmitarbeitern, vernachlässigt werden, oder eine letztendlich doch eher ungünstige Reihenfolge der Verbesserungen gewählt wird.

Das SM^{mm} hingegen gibt neben detaillierten Anweisungen für die einzelnen Prozessbereiche, durch die Definition der Reifegrade eine mehr oder weniger feste Reihenfolge der Verbesserungen vor. Dies kann vor allem bei Unternehmen, die im Bereich Prozesse, etc., noch nicht so viel Erfahrung haben, durchaus von Vorteil sein.

Ein weiterer positive Punkt des SM^{mm} im Vergleich zum CM^3 ist, dass es alle Bereiche der Softwarewartung abdeckt. Da oftmals auch nicht ganz klar ist, in welche Kategorie von Wartungstätigkeiten eine ganz bestimmte Aktivität nun fällt, ist es, meiner Meinung nach, vorteilhafter, die Wartung als ganzes zu betrachten und zumindest im Bereich der Prozesse keine zu großen Unterscheidungen zwischen den einzelnen Kategorien zu machen.

Unabhängig davon, bleibt beim CM^3 die Frage, wie mit den anderen Wartungsbereichen verfahren werden soll. Dies wird wohl erst mit dem Erscheinen des umfassenderen Modells M^3 geklärt.

Im Vergleich zum CMMi haben beide Modelle den Vorteil, dass sie speziell auf die Softwarewartung zugeschnitten sind. Zwar lässt sich das CMMi prinzipiell auch auf Prozesse in der Softwarewartung anwenden, für viele Unternehmen dürften SM^{mm} und CM^3 aber günstiger sein, da diese auch bewährte Praktiken für die Softwarewartung mit bringen, die bei CMMi zum größten Teil selbst mit eingebracht werden müssten.

Die beiden Modelle stellen also eine sinnvolle Entwicklung für die Softwarewartung dar, wobei CM^3 noch einen Erweiterungsbedarf hat.

12.5 Zusammenfassung

In der Softwareentwicklung werden schon seit längerer Zeit Prozessmodelle zur Steuerung der Abläuft eingesetzt. Dies ist auch in der Softwarewartung möglich. Um einen Eindruck davon zu erhalten, welche Aspekte hier besonders zu beachten sind, lohnt es sich, entsprechende Reifegradmodelle zu betrachten.

In der Softwareentwicklung ist eines der verbreitetsten Reifegradmodelle das Capability Maturity Model Integration (CMMi). Dieses bildet auch eine der Grundlagen für die beiden Reifegradmodelle der Softwarewartung, die in dieser Ausarbeitung vorgestellt werden, das Software Maintenance Maturity Model (SM^{mm}) und das Corrective Maintenance Maturity Model (CM^3) für den Bereich der korrigierenden Wartung.

Diese beiden Modelle geben einen Überblick über die bei der Einführung und späteren Optimierung von Prozessen in der Softwarewartung zu beachtenden Aspekte.

Literaturverzeichnis

[AHAD05] Alain April, Jane Huffmann Hayes, Alain Abran, and Reiner Dumke. Software maintenance maturiy model. *Journal Of Software Maintenance And Evolution*, 17:197–223, 2005.

[AM04] Alain Abran and James W. Moore. *Guide to the Software Engineering Body of Knowledge*. 2004.

[Ins02] Software Engineering Institute. *Capability Maturity Model Integration, Version 1.1*. Carnegie Mellon University, 2002.

[KM01] Mira Kajko-Mattsson. Motivating the corrective maintenance maturity model. *Seventh IEEE International Conference on Engineering of Complex Computer Systems, 2001. Proceedings*, 2001.

Kapitel 13

Evolution von Programmen und Prozessen in Open Source Software und Free Software

Andreas Ganser

Inhaltsverzeichnis

13.1 Motivation

Zweifelsohne sind viele Free and Open Source Programme erfolgreich. So erreicht der Apache Webserver Anfang 2006 eine Marktpräsenz von ca. siebzig Prozent [Net06]. Nun drängt sich die Frage auf, ob dieser Erfolg alleine vom Preis abhängt oder noch andere Faktoren eine Rolle spielen. Diese Frage erübrigt sich, wenn in Betracht gezogen wird, dass auch viele kommerzielle Internetpräsenzen per Apache gehostet werden. Dort sind die Anforderungen an Verfügbarkeit und Zuverlässigkeit von Servern hoch, was ohne qualitativ hochwertige Programme nicht gewährleistet wäre.
Ohne ein Qualitätsurteil zu fällen, lässt sich feststellen, dass Free and Open Source Programme auch im kommerziellen Bereich Einsatz finden. Das beschränkt sich nicht nur auf den Apache Webserver sondern lässt sich auch für den Übersetzer GCC, den Betriebsystemkern Linux und viele weitere Projekte feststellen.
Abseits von der Qualität der Programme geben Mockus et al. eine Liste von Eigenschaften an, die zusätzlich für die Verwendung von Free and Open Source Programmen sprechen [MFH02]. Darunter befindet sich auch die schnelle und flexible Reaktion der Entwickler auf Veränderungen in den Anforderungen und bei Problemen. Diese Eigenschaften sind in den letzten Jahren in mehreren Studien untersucht worden und konnten weitgehend mit Fallstudien bestätigt werden. Doch fällt bei näherer Betrachtung auf, dass für eine schnelle und flexible Reaktion auf Anforderungen auch flexible Entwicklungsprozesse notwendig sind.
Aus dem kommerziellen Bereich ist bekannt, dass Entwicklungsprozesse möglichst gut beschrieben sein sollten. Außerdem sind sie aufgrund von Gewohnheiten schwer änderbar. Deshalb lohnt es die Prozesse und Technologien der Free and Open Source Gemeinschaft genauer zu betrachten. Dabei soll die Evolution von Prozessen und Architekturen im Vordergrund stehen.
Dafür ist eine gewisse Grundlage an Wissen über die Free and Open Source Gemeinschaft und die Verschiedenheit der einzelnen Projekte hinsichtlich Organisation und Prozesse nötig. Des Weiteren benötigt der spätere Vergleich zum kommerziellen Bereich eine kurze Rekapitulation der Erkenntnisse aus dem kommerziellen Bereich; insbesondere der Gesetze von Lehman. Diese Erkenntnisse können dann miteinander verknüpft werden und es können Ähnlichkeiten zu anderen Ideen gesucht werden.

13.2 Open Source und Free Software

Zunächst ist eine Erklärung der Begriffe Open Source Programme und Free Software erforderlich. Zwar werden die Begriffe vorerst abgegrenzt, doch wird später keine scharfe Trennung zwischen Free Software und Open Source vorgenommen. Deshalb wird dann nur die Kurzform „FOSS“ für Free and Open Source Software benutzt.
Von Open Source Programmen wird gesprochen, wenn der Quelltext der Programme vorliegt, dieser genutzt, verbreitet und kopiert werden und in veränderter wie unveränderter Fassung weitergegeben werden darf. Diese Definition basiert auf der „Open Source Definition“, die ursprünglich von Bruce Perens stammt, einem ehemaligen Projetleiter bei Debian GNU/Linux.

Als Free Software werden Programme bezeichnet, deren Lizenz die Nutzung für jeglichen Zweck ausdrücklich erlaubt. Somit müssen diese Programme im Quelltext vorhanden sein, damit sie nach Belieben verteilt, kopiert und verändert werden können. Der Quelltext muss auch nach jeder Veränderung weiterhin zugänglich bleiben, auch die veränderte Version.
Zunächst unterscheiden sich Open Source Programme und Freie Software nicht voneinander, doch lösen sie gewöhnlich unterschiedliche Assoziationen aus. Hier soll nicht weiter auf diese Feinheiten eingegangen werden; etwas detaillierter werden Lizenzen in Abschnitt 13.2.1 betrachtet.
Abzugrenzen von FOSS ist die Closed Source Software. Darunter soll sämtliche proprietäre und kommerzielle Software fallen, deren Quelltext nicht allgemein verfügbar ist.

13.2.1 Lizenzmodelle

Im Bereich der FOSS gibt es eine schier unendliche Anzahl von Lizenzen, unter denen Programme stehen können. Insbesondere Firmen, die Teile ihre Closed Source Programme unter eine Open Source oder Free Software Lizenz stellen, erstellen gerne ihre eigene Lizenz. Populäre Beispiele sind OpenSolaris oder die SAP Datenbank SAP-DB. Doch hier sollen nur die beiden, vermutlich, wichtigsten Lizenzen kurz erwähnt und beschrieben werden.
Bei Programmen, die unter GNU GPL [Fre91] veröffentlicht werden, darf das ursprüngliche Urheberrecht nicht geändert werden. Somit müssen alle Programme, die einmal unter GNU GPL standen auch weiterhin darunter verbleiben. Ganz besonders muss jede Modifikation oder Weiterentwicklung wieder unter GNU GPL stehen.
Liberaler sind die BSD-Lizenzen [Uni98]. Sie erlauben auch die kommerzielle Nutzung der ursprünglichen Programme, solange der Urheber genannt bleibt. Insbesondere schließt das die Möglichkeit ein, die modifizierten Programme als Closed Source Programme zu vertreiben. In Abschnitt 13.2.3 wird als Beispiel auf einen Betriebsystem namens Darwin eingegangen.

13.2.2 Entwicklungsprozesse bei Free and Open Source Software

Oft wird der Entwicklungsprozess bei FOSS als chaotisch und zufällig beschrieben. Ein populärer Artikel von Eric Steven Raymond aus dem Jahr 1998 mit dem Titel „The Cathedral and the Bazaar“ [Ray98] wird in diesem Zusammenhang gerne zitiert und falsch verstanden. Dabei nutzt Raymond die Orte Kathedrale und Bazar, um zwei mögliche Hierarchien bei der Entwicklung von FOSS zu beschreiben. Keiner von beiden Orten ist bei dieser Beschreibung chaotisch. Stattdessen gibt es auf dem Basar eine Art Marktaufsicht und bei der Kathedrale eine strenge Hierarchie, wie sie aus der römisch-katholischen Kirche bekannt ist. Folglich existiert bei diesem Modell ein Bauplan und ein Chef.
Wie sich die interne Organisation der Projekte in der Entwicklung von FOSS darstellt, ist meist historisch gewachsen. Es gibt verschiedene Grade von „Strenge“ in

den Prozessen. Angefangen von einfachen moderierten Projekten bishin zu Projekten, die ähnlich im kommerziellen Umfeld vorkommen, ist vieles auffindbar. So ist der Übersetzer GCC ein sogenanntes „free-speech“ Projekt (siehe auch Abschnitt 13.5.1), bei dem jeder seinen Beitrag direkt einbringen kann [He02]. Der populäre Webserver Apache ist strikter organisiert und zumindest moderiert. Noch strenger organisiert ist das FreeBSD-Projekt. Dort gibt es im weitesten Sinne eine Hierarchie und spezielle Zuständigkeiten. Das Mozilla-Projekt ist noch straffer organisiert.
Insgesamt ist bei allen FOSS-Projekten festzustellen, dass sich die Konzepte und Entwicklungstechniken über die Zeit entwickelt haben. Dabei sind erfolgreiche Programme entstanden, die in puncto Qualität ihrer kommerziellen Konkurrenz nicht nachstehen. Vorzeigeprogramme wie Apache Server, GCC, Linux, Emacs und viele andere mehr sind über mehr als zehn Jahre entwickelt worden. Ohne spezielle Konzepte wäre das in diesem „gesunden“ Maße nicht möglich gewesen. Insgesamt ist die Entwicklung von FOSS Programmen oft sehr gut organisiert und mit klaren Rollen in gut strukturierten Prozessen ausgestattet.
Gestützt werden die Konzepte und Techniken durch Werkzeuge wie das Concurrent Versions System, kurz CVS, und zunehmend dessen Nachfolger Subversion. Doch über die Zeit hinweg müssen sich auch die Prozesse den Werkzeugen angepasst haben. Denn als GCC und Emacs in ihrer ersten Version veröffentlicht wurden, gab es weder Mailinglisten noch das World Wide Web. Deshalb muss ein Zusammenhang zwischen der Veränderung der Prozesse und der Qualität von FOSS existieren. Denn nicht umsonst setzt Apple heute auf den FreeBSD-Systemkern, der Teile seiner Ideen und Quelltexte in der Mitte der Achtziger Jahre findet.

13.2.3 Kommerzialisierung

Mac OS wird bei Apple unter dem Code-/Projektnamen „Darwin“ entwickelt. Apple möchte mit der Benennung nach Charles Darwin auf die Evolutionstheorie von ihm anspielen. Dadurch zeigt Apple, dass sie auf „gesunde“ Evolution setzten und unterstreichen den Erfolg des zugrundeliegenden FreeBSD-Projektes .
Doch die Einflüsse von Firmen in der FOSS-Entwicklung sind inzwischen weitreichender, als einfache Nutzung. Denn viele große FOSS-Projekte werden durch Firmen personell und finanziell gestützt. Die Motivation der Firmen mag unterschiedlich sein, der Direktor des Linux Technology Center, Dan Frye, hat es im November 2001 so formuliert: „The Linux business model for IBM is staightforward: We sell the hardware underneath it, we sell the software on top of ist, and we sell the services all around it. The fact that we don’t sell the thin layer Linux operating system is frankly irrelevant.“ Eine klare Kampfansage an andere Betriebssystemhersteller!

13.2.4 Brückenschlag zu Closed Source Entwicklung

FOSS Programme zeigen viele Erfolge und eine enorme Energie [BP03]. Dennoch finden viele Erkenntnisse aus Wissenschaft und kommerziellen Bereichen keine Beachtung bei der Entwicklung von FOSS. Andersherum beachten Firmen bei der Entwick-

lung von Closed Source Programmen die Erkenntnisse aus Wissenschaft und FOSS-Entwicklung nicht. Die Veröffentlichung von Bauer und Pizka [BP03] versucht hier eine Brücke zu schlagen. Denn in „The Contribution of Free Software to Software Evolution“ versuchen sie die Vorteile der FOSS-Entwicklung aufzuzeigen und stellen zwei Thesen auf, die durch mehrere Fallstudien geprüft wurden ([MFH02] und [DB04]). Im einzelnen wird behauptet:

- Der dynamische Prozess der Entwicklung freier Software hängt eng zusammen mit dem Wachstum der Code-Basen und der stets veränderlichen Projektgröße.

- Die Evolutionsstrategieen in FOSS weisen wegen ihrer liberalen Umgebung starke Ähnlichkeiten zum Wirtschaftsliberalismus. Somit setzt sich das beste Programm durch.

Um die Verknüpfung und Erklärung dieser Thesen wird sich auch der folgende Text bemühen. Als Basis für die Untersuchung und Diskussion sind allerding Kenntnisse über Lehmans Gesetze der Softwareevolution und über Mockus Hypothesen nötig.

13.3 Lehmans Gesetze und Mockus Hypothesen

Bereits in den späten Sechzigern und den frühen Siebzigern untersuchte Lehman die Entwicklung von Programmen über einen langen Zeitraum und über Versionssprünge hinaus. Diese Entwicklung bezeichnet er als Softwareevolution. Ausgangspunkt für seine Untersuchungen waren Programme und Programmsysteme aus dem kommerziellen Bereich. Darunter befindet sich das Großrechnerbetriebssystem OS/360 von IBM, ein Finanzverwaltungsprogramm, zwei Programmsysteme für Telekommunikation und ein Verteidigungssystem. Für die Untersuchung von Softwareevolution schlägt Lehman [LR01] vor, die „Natur“ der Evolution zu untersuchen. Damit meint er besonders ihre Ursachen, Eigenschaften, Charakteristika, Konsequenzen usw. Insgesamt schlägt er acht Gesetze vor:

1. Continuing Change
2. Increasing Complexity
3. Self Regulation
4. Conservation of Organisational Stability
5. Conservation of Familiarity
6. Continuing Growth
7. Declining Quality
8. Feedback System

Eigentlich wurden diese Gesetze im Bereich der Großrechner ermittelt und später angepasst. Doch auch für FOSS treffen sie weitgehend zu. Das ist für Gesetz eins, zwei und sechs trivial; in Abschnitt 13.6.3 werden die restlichen diskutiert.
Zusätzlich sind derzeit die „sieben Hypothesen“ von Mockus Fallstudie über Apache Server und Mozilla [MFH02] in der Diskussion:

1. Open Source Developments will have a core of developers (10-15 people) who control the code base, and will create approximately 80% or more of the new functionality.
2. If a project is so large that more than 10-15 people are required to complete 80% of the code in desired time frame, then other mechanisms, rather than just informal ad hoc arrangements, will be required in order to coordinate the work.
3. In successful open source developments, a group larger by an order of magnitude than the core will repair defects, and yet a larger group will report problems.
4. Open source developments that have a strong core of developers but never achieved large numbers of contributors beyond that core will be able to create new functionality but will fail because of a lack of resources devoted to finding and repairing defects.
5. Defect density in open source releases will generally be lower than commercial code that has only been feature-tested.
6. In successful open source developments, the developers will also be users of the software.
7. Open source developments exhibit very rapid responses to customer problems.

In einer Fallstudie anhand FreeBSD wurden diese Hypothesen untersucht [DB04] und weitgehend bestätigt.
Gerade im Bereich der FOSS-Projekte kann es zusätzlich sinnvoll sein, kulturelle Aspekte sowie Aspekte über Sprache und Wirtschaft bei der Betrachtung zu berücksichtigen. So beschreibt Scacchi [Sca04], dass diese Aspekte in der Free and Open Source Gemeinschaft eine wichtige Rolle spielen. Somit kommt er zu einem grundlegend anderen Ergebnis, als z.B. Hofstede in seinem Buch „Cultures and Organizations - Software of the Mind“ [Hof96]. Allerdings hat Hofstede unter arbeits-psychologischen Aspekten gearbeitet. Dabei hat er die Verteilung von Firmen mit Standorten in vielen verschiedenen Ländern betrachtet und deren Auswirkungen auf die soziale Struktur bei der Zusammenarbeit bei Projekten in diesen Firmen. Insgesamt kommt er zu dem Ergebnis, dass kulturelle Unterschiede sehr wohl eine Barriere für die Zusammenarbeit in Projekten sind.

13.4 Evolution der Entwicklungsprozesse

Nicht selten beginnen Entwicklungen von FOSS im Stil des Rapid Prototyping: nachdem eine Aufgabe identifiziert ist, wird ein Prototyp programmiert, der höchstens die rudimentärsten Anforderungen an Struktur erfüllt. Dieser Prototyp ist und soll nicht mehr sein als ein „Proof of Concept“. In diesem Schritt der Entwicklung ist keinerlei Administration nötig, da das Programm meist ein Ein-Mann-Projekt ist. Projektdetails existieren meist nicht niedergeschrieben, sondern nur „im Kopf“ des Entwicklers – eine Anforderungsspezifikation gibt es also nicht. Da die Idee zu diesem Programm nicht selten in einer Gruppe entstanden ist, wird es deshalb häufig auf einem der Projektserver für FOSS der Öffentlichkeit zur Verfügung gestellt [Sou]. Bekanntheit findet dieses Programm dann über Mundpropaganda und die Zugriffsmöglichkeit auf den Servern.
Findet dieser Prototyp Resonanz und Anwendung, so wird das Programm zu einem

Projekt. Dann können die Werkzeuge der entsprechenden Server administrative Unterstützung liefern. Schnell werden dann Mailinglisten eingerichtet, Fehler-Datenbanken aufgesetzt usw.
Mit der Größe des Projektes müssen sich auch die Entwicklungsprozesse weiterentwickeln [BP03]. In der Tat skaliert nach Bauer und Pizka der Entwicklungsprozess bei freier Software höchst dynamisch mit der darunterliegenden Architektur und den Fähigkeiten der Menschen, die an dem Projekt beteiligt sind. Ist es anfänglich noch möglich ein paar Programmierer über eine Mailingliste zu koordinieren, kann diese Art der Organisation bei zunehmender Größe schnell schwierig werden. Dann können andere, mächtigere Werkzeuge eingesetzt werden und meist muss die technische Architektur geändert werden. Zudem wird ein Archiv nötig, die Kommunikationswege ändern sich usw. Eine Skalierung der Entwicklungsprozesse findet jedoch nur reaktiv statt und ist deshalb stark mit der Komplexität des Projektes verwoben.
Doch viele Werkzeuge, die heute selbstverständlich sind, gab es Mitte der Achtziger Jahre noch nicht. Das ist die Zeit, in der viele FOSS-Projekte entstanden sind, die aus der Perspektive der Softwareevolution interessant sind. Zudem haben sich bei diesen FOSS-Projekten ggf. einerseits die ursprünglichen Ziele und andererseits die Anzahl und Art der Mitwirkenden verändert.
Ein populäres Beispiel ist der Übersetzer GCC. Ursprünglich Mitte der Achtziger von Richard Stallman als schmaler Übersetzer initiiert, ist GCC heute eine sogenannte Compilersuite mit ca. einer Million Quelltextzeilen und Unterstützung von mehr als zweihundert Plattformen [Piz97]. Aufgrund der Projektgrösse reichen deshalb E-Mail und Usenet nicht mehr für Kommunikation und Koordination aus.

13.4.1 Entwicklung der Werkzeuge und Werkzeugunterstützung

Seit die erfolgreichen FOSS-Projekte meist Mitte der Achtziger initiiert wurden, haben sich weitreichende technische Veränderungen ergeben. Zu der Möglickeit via E-Mail, Mailinglisten und Newsgroups zu kommunizieren, haben sich beispielsweise Chatsysteme und das World Wide Web entwickelt oder sind erst entstanden.
Um auf das Beispiel des Übersetzers GCC zurückzukommen, haben Bauer und Pizka in [BP03] folgende Vorteile aus den technischen Fortschritten für die Entwicklungsprozesse identifiziert:

- Automatisches Management von Mailinglisten mit Zugriff auf durchsuchbare Archive und Web-Oberflächen
- CVS-Server mit Web-Oberfläche, die sowohl verschiedene Versionen als auch unabhängige Enwicklungen innerhalb des Projektes verfolgen
- Große Anzahl von *mirror sites*, die die Verfügbarkeit erhöhen
- Einführung und Interesse von neuen Sprachen und Hardwareplattformen
- modernes und automatisiertes Fehlerverfolgungssystem mit weltweitem Zugang
- *compile farms*, die an einem Punkt zentralen Zugang zu mehreren Plattformen bieten und für gewöhnlich durch Firmen gefördert werden, die ein Interesse an Open Source Produkten haben

- Wachsende Anzahl periphärer Projekte, die auf dem GCC aufbauen, aber ihren Fortschritt unabhängig managen

Neben diesen rein technischen Veränderungen, haben sich auch bei den Werkzeugen Fortschritte ergeben. Zu den CVS-Servern kamen immer wieder Programme, die durch die Notwendigkeit der einzelnen Projekte entstanden — streng genommen ist auch CVS so entstanden — so ist z.B. Bugzilla als ein Nebenprodukt beim Mozilla-Projekt entstanden und dadurch zu einem sehr beliebten Fehlerverfolgungssystemen geworden. Notwendig war Bugzilla geworden, da eine ständig steigende Anzahl von Mitwirkenden ihren Beitrag zur Qualität der Mozilla-Suite liefern wollten.

13.5 Evolution der personellen Architektur

Da die Anzahl der Quelltextzeilen meist mit der Anzahl der Beiträge und Mitwirkenden wächst, wird diesbezüglich an bestimmten Schwellenwerten von kritischen Größen gesprochen. Das sind typische Zeitpunkte, an denen die personelle Struktur der FOSS-Projekte verändert werden muss. Diesen Veränderungen geht in der FOSS-Entwicklung meist eine Restrukturierung der technischen Architektur voraus. Das verdeutlicht, dass diese Ebenen eng miteinander verknüpft sind. Deshalb werden hier die Begriffe personelle und technische Architektur verwendet; die technische Architektur wird in Abschnitt 13.6 betrachtet.
Welche personellen Architekturen möglich sind, soll anhand des Übersetzters GCC und des FreeBSD-Projektes gezeigt werden. Diese beiden FOSS-Projekte eignen sich aufgrund ihrer verschiedenen personellen Organisation gut, um zu veranschaulichen, dass es bei der personellen Architektur von FOSS-Projekten keinen „goldenen Weg“ gibt. Stattdessen sind diese Strukturen historisch gewachsen und alle für sich haben zum Erfolg der FOSS-Projekte geführt.

13.5.1 Struktur von Projekten

Alle FOSS-Projekte haben gemeinsam, dass es einen gewissen „Lenkungsausschuss“ gibt. Die Größe dieser Kernteams bewegt sich bei ca. 15 Personen. Diese Zahl ist bemerkenswert, da sie sich bei den genannten FOSS-Projekten unabhängig und über die Zeit entwickelt hat. Tabelle 13.1 zeigt eine Übersicht über die Führungsgremien einiger FOSS-Projekte.
Ein Vergleich mit Untersuchungen zu Erfolg und Scheitern bei kommerziellen Projekten bestätigt diese Zahlen als „gute“ Wahl. So zeigen die CHAOS-Berichte der Standish Group [Int95] und [Int99]: Die Wahrscheinlichkeit, dass ein Projekt scheitert, wächst mit der Größe des Projektes.
Bemerkenswert ist, dass sich die Größe der Führungsgremien der FOSS-Projekte evolutionär auf eine gewisse Größe eingependelt hat und ebenso die Subprojekte eine adäquate Größe haben. Bauer nimmt dieses Indiz als Bestätigung dafür, dass diese Werte in Selbstregulierung durch Wettbewerb und Auslese entstanden sind – eben ge-

rade so, wie es im Wirtschaftsliberalismus typisch ist [BP03]. Somit werden FOSS-Projekte anhand der Notwendigkeit in kleinere FOSS-Projekte aufgeteilt und damit die personelle Zuteilung aufgegliedert. Insgesamt skaliert die personelle Architektur mit der Größe des Projektes.
Doch es soll noch auf ein paar signifikate Punkte einzelner Projekte eingegangen weden:

GCC

Beim Übersetzer GCC besteht das Steering Committee aus zwölf professionellen Entwicklern, die Vollzeitbeschäftigte von z.B. IBM, Novell und Red Hat sind. Jeder für sich ist Experte auf seinem Gebiet. Denn es kann von keinem der Zwölf verlangt werden, ein Verständnis für die gesamten, rund eine Million Quellcodezeilen zu haben. Die Hauptaufgabe des Steering Committee ist es, die zentralen Entscheidungen im Sinne des Projektes zu fällen und deren Einhaltung zu gewährleisten. Zudem dürfen die Entscheidungen nicht den Grundideen des Projektes zuwiderlaufen [Com98].
Die Idee bei der Projektstruktur von GCC ist ein sogenannter free-speech-Ansatz. Deshalb sind alle Mailinglisten des Projektes öffentlich und jeder kann sich dort einbringen. Außerdem sind die Projektserver öffentlich lesbar, sodass sich jeder Interessierte einen Eindruck von dem aktuellen Stand der Entwicklung machen kann. Zwar gibt es Grundregeln, wie ein Patch auszusehen hat, doch werden alle Patches anhand von Prioritätsregeln behandelt. Das bedeutet insbesondere, dass es keine Privilegien für spezielle Entwickler gibt.

FreeBSD

Für das FreeBSD-Projekt gibt es i.a. keine Vollzeitbeschäftigten. Sämtliche Arbeit wird i.a. von Freiwilligen durchgeführt. Dennoch ist die Organisation des FreeBSD-Projektes sehr gut dokumentiert [Sae06] und besitzt strenge Rollenkonzepte. So gibt es ein Core Team von neun Personen, dass die Steuerung des Projektes übernimmt und als eine Art Vorstand und Verwaltungsrat agiert. Zudem haben die Core Team Mitglieder eine Schiedsfunktion in problematischen Situationen. Gewählt wird das Core Team alle zwei Jahre aus den Reihen der Committer. Letztere sind Entwickler mit besonderen Rechten im Projekt. Sie können unter anderem Veränderungen am CVS-Repository durchführen und haben Zugriff auf projektinterne Diskussionen und Resourcen.
Um einer von rund 300 Committern zu werden, sollte ein Contributer mehr als zwölf Monate ein aktiver Entwickler, Tester, Autor o.ä. ohne besondere Rechte im Projekt gewesen sein. Er kann dann vorgeschlagen werden und wird vom Core Team per Wahl ernannt (Stand 2004). Dann bekommt er einen Mentor, und bringt seine Beiträge in seinem Spezialgebiet ein. Es gibt jedoch keine technischen Hürden, die ihn hindern auch an anderer Stelle mitzuarbeiten. Eine Übersicht über die Hierachie, die Core Team, Committer und Contributoren bilden, gibt Abbildung 13.1.
Über die Hierachie hinaus gibt es sogenannte Hats. Das sind spezielle Rollen, die

Projekt	Gremiumname	Anzahl
Debian	Leader and Technical Committee	8
FreeBSD	Core	9
GCC	steering committee	13
gnome	sysadmin committee	11
KDE	Core Group	20
Mozilla	Project Managers („Drivers“)	13

Tabelle 13.1: Führungsgremien bei FOSS

Committer per Ernennung durch das Core Team einnehmen können. Diese Hats sind nicht immer speziell dokumentiert, doch gibt es einige wichtige Hats: die Maintainer. Dabei handelt es sich um Zuständige für einzelne Zweige. Ein Maintainer hat in seinem Zweig das Privileg des „letzten Wortes“ und er steht in bzw. legt die sogenannte Maintainer-Datei fest. Darin finden sich die Spezifikationen, die er für seinen Zweig festlegt. Will sich ein Committer außerhalb seines Spezialgebiets einbringen, wird ihm nahegelegt die Maintainer-Datei zu beachten.
Schon an diesem kleinen Einblick in das FreeBSD-Projekt wird klar, dass es eine klare Struktur gibt und die Rollen wohldefiniert sind. Entsprechend gibt es genaue Dokumentationen darüber, wie Privilegien verteilt werden und wer wofür zuständig ist [Sae06]. Ebenso ist für Ausfallzeiten von Maintainern ein Stellvertreter bestellt sowie klar definiert, was einen aktiven Committer auszeichnet.
Weiterhin sind auch die Entwicklungsprozesse klar definiert [Sae06] und erinnern an das Wasserfallmodell. Folglich sind je nach Projektzustand gewisse Änderungen verboten und werden bei Verstoß durch eine tageweise Aussetzung der Committer-Privilegien geahndet. Ebenso kann ein „unproduktives Verhalten“ zu einem Entzug der Privilegien zwischen zwei und fünf Tagen führen. Dabei wird je nach Ursache (commit wars, unhöfliches oder unangemessenes Verhalten, unerlaubtes commit während entsprechender Phasen) von einem Core Team Mitglied das Committ-Privileg ausgesetzt. Dazu ist gewöhnlich keine Diskussion innerhalb des Core Teams nötig, aber möglich. Notfalls wird per $2/3$ Mehrheit entschieden.

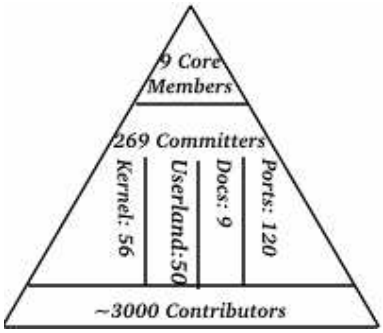


Abbildung 13.1: Hierarchie von FreeBSD gemäß [Sae06]

13.5.2 Konfigurationsmanagement

Die großen FOSS-Projekte setzen zusehends auf ähnliche, bewährte Werkzeugunterstützung. Zugrunde liegen beim Konfigurationsmanagement u.a. Programme wie CVS und verstärkt Subversion, BitKeeper, Bugzilla sowie die klassischen Werkzeuge patch und diff. Wichig ist, dass Programme für ähnliche Probleme oft auch ähnlichen Konzepten folgen. Nur deshalb kann es zu einer Angleichung des Konfigurationsmanagement zwischen den Projekten kommen – egal, wie diese Thematik innerhalb des Projektes bislang, historisch gewachsen, behandelt wurde. Interessant ist, dass trotz verschiedener Ansätze der Projeke ein ähnlicher Erfolg im Umgang mit dem Konfigurationsmanagement zu erkennen ist [NYNY02].
Die markanten Unterschiede liegen hauptsächlich in der Liberalität im Umgang mit den Quelltexten bezüglich der Veränderungen. Doch diese Eigenschaften unterliegen auch Diskussionen. Ursachen dafür sind z.B. die sogenannten commit wars [Leh02]. Dabei handelt es sich um konkurrierende Änderungen an gleichen Quelltextstellen, um eine Idee einzubringen bzw. eine andere zu überschreiben. Deshalb gibt es beim Linux Betriebsystemkern-Projekt inzwischen erhebliche Einwände gegen öffentliche CVS-Server, obwohl bei den „kritischen“ Stellen Linus Torvalds als eine Art Projektleiter das letzte Wort hat. Bei den BSD-Projekten existiert die Diskussion über öffentliche CVS-Server nicht.

13.5.3 Änderungsmanagement

Aufbauend auf die technische Struktur des Versionsmanagements, erfordern große Projekte ein Änderungsmanagement. Gemäß einer Untersuchung von van der Hoek [Hoe00], gehen FOSS-Projekte dabei ähnlich vor, wie es in kommerziellen Projekten bekannt ist. Einzig die Werkzeuge mußten sich erst entwickeln, wie es in Abschnitt 13.5.2 erläutert wurde. Doch gerade diese Werkzeuge ermöglichen heute komplexe Strukturen im Änderungsmanagement. Vorab sei schon darauf hingewiesen, dass die Facetten in den FOSS-Projekten ähnlich weit gefächert sind, wie in Abschnitt 13.2.2 bereits angedeutet. Das verwundet nicht, haben sich doch die Werkzeuge und die personelle Struktur anhand der Notwendigkeiten und deren Wechselwirkungen entwickelt.
Um jetzt einmal die Entwicklungsprozesse in FOSS-Projekte zu untersuchen, werden FreeBSD und GCC genauer betrachtet: Diese beiden Projekte sind aufgrund ihrer Unterschiedlichkeit gewählt und decken nur die Extrema des Spektrums ab, dass in der FOSS vorgefunden wird.
Entsprechend der wohldefinierten Rollen im FreeBSD-Projekt, folgt der Entwicklungsprozess streng definierten Vorgaben [Sae06]. Hier wird als Teil davon der Releaseprozess betrachtet: Zunächst wird zwischen einem Entwicklungszweig (CURRENT) und einem Anwendungszweig (STABLE) unterschieden. Zusätzlich werden von dem FOSS-Projekt noch Hinweise zum Einsatz von STABLE-Versionen im Produktivbetrieb gegeben.
Der wichtigste Unterschied zwischen STABLE und CURRENT ist, dass nur in dem CURRENT-Zweig die aktive Entwicklung stattfindet. Nur hier dürfen neue Funktionen

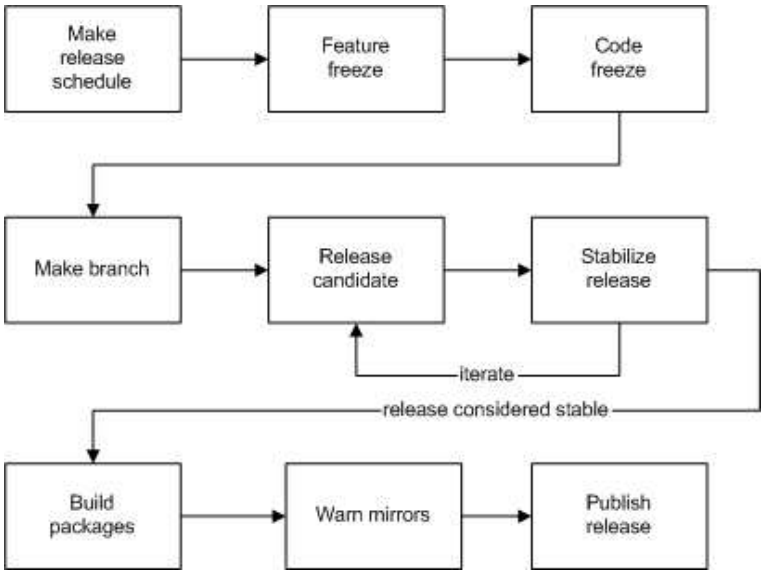


Abbildung 13.2: Entwicklungsprozess bei FreeBSD gemäß [Sae06]

eingebracht werden. Allerdings nur, wenn der Maintainer des Zweigs das befürwortet. Ist der Funktionsumfang definiert, können alle Committer ihre Beiträge liefern.
Anders verhält es sich im STABLE-Zweig. Dort dürfen i.a. keinerlei Funktionsänderungen vollzogen werden. Einzig Fehlerbehebung und Wartung finden in diesem Zweig statt. Änderungen am Funktionsumfang sind nur in einem sehr konservativen Umfang und in Ausnahmefällen möglich. Doch der Weg einer Version eines FreeBSD-Modules von CURRENT zu STABLE ist weit (Abbildung 13.2): Von einem feature freeze über einen code freeze bis zu einem release candidate und letztlich einem stable release gehen die Schritte, die teilweise iteriert werden [Sae06]. Sobald die Module des Zweiges gebaut sind und auf den Mirror-Servern verfügbar sind, ist der STABLE-Status erreicht. Bei jedem Schritt nimmt dabei die Anzahl möglicher Aktionen am Quelltext ab [DB04].
Wie bereits in Abschnitt 13.5.1 angedeutet, gibt es in der personellen Organisation von GCC keine Hierarchie. Deshalb werden alle Beiträge gleich behandelt und von dem Steering Committee ausgewertet. Da es sich dabei um Vollzeitbeschäftigte handelt, reicht offenbar eine Anzahl von zwölf Mitgliedern aus, um eine qualitativ hochwertige Entwicklung zu gewährleisten.
Somit stützen beide Fallbeispiele weitgehend die Hypothesen von Mockus aus Abschnitt 13.3.

13.6 Evolution der technischen Architektur

Tabelle 13.2 zeigt, dass erfolgreiche FOSS-Projekte durchaus älter als zwanzig Jahre sein können. Die populären Beispiele sind der Übersetzer GCC, die GNU-Tools und der Editor Emacs.

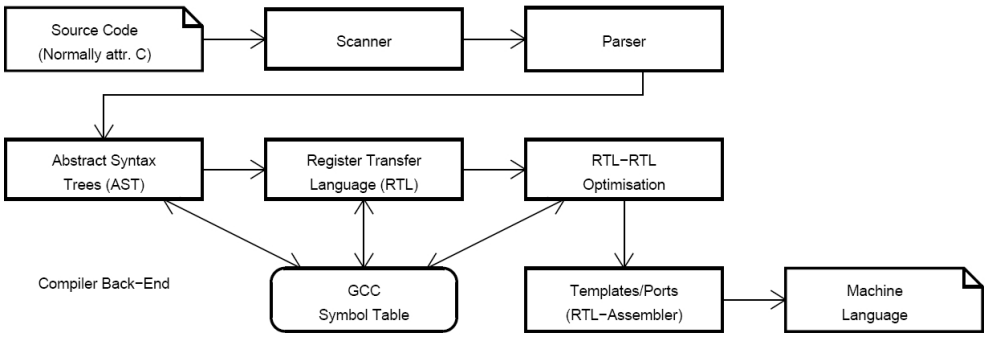


Abbildung 13.3: technische Architektur von GCC gemäß [Piz97]

Mit der Zeit haben sich die Anforderungen an die Programme geändert. Einerseits haben sich die Anwender und deren Erwartungen an „gute“ Programme (usability) geändert und andererseits haben sich die Anforderungen an den Funktionsumfang geändert. So ist bei dem Editor Emacs eine grafische Benutzeroberfläche hinzugekommen; der Übersetzer GCC hingegen hat sich der aktuellen Forschung angepasst, den Funktionsumfang und die unterstützten Plattformen erweitert. All diese Änderungen haben sich auch im Umfang des Quelltextes niedergeschlagen. Je nach FOSS-Projekt lässt sich zeigen, dass der Quelltext linear oder super-linear gewachsen ist [GT00].
Dieses Wachstum der Quelltexte birgt die Gefahr, dass das Projekt in einem unwartbaren Zustand endet. Diese Befürchtungen und Gefahren wurden von Cooke et al. und von Perkin formuliert [CUH99], [Per99].
Alter und Qualität der o.g. FOSS-Projekte zeigt allerdings, dass es über die Zeit hinweg in der technischen Architektur Veränderungen gegeben haben muss; was Pizka für den Übersetzer GCC gezeigt hat [Piz97]. Entsprechend finden über die Zeit in FOSS-Projekten Veränderungen in der technischen Architektur statt, die durch Notwendigkeiten getrieben wurden. Doch welche Methoden werden dabei eingesetzt?

13.6.1 Modularisierung und Ausgliederung

Als Intel seine 64-Bit-CPUs veröffentlichte, wurde beim Übersetzer GCC eine Veränderung des cross vendor application binary interface (register transfer language, kurz: RTL) notwendig [Piz97]. Dadurch sollte sichergestellt werden, dass die erstellten Bi-

Projekt	Veröffentlichungsjahr	Quelltextzeilen [ca.]
Emacs 20.7	1984	600.000
FreeBSDKernel	Mitte 80er	2 Mio
GCC 2.96	1984	1 Mio
Linux 2.4.2	1991	2.5 Mio
Mozilla	1998	2 Mio

Tabelle 13.2: Größe von FOSS-Projekten nach [Whe02]

närdateien weiterhin kompatibel zu denen anderer Übersetzer sind. Die Veränderungen an dem application vendor binary inteface sind allerdings alles andere als trivial, da diese Schicht völlig transparent in den Übersetzungsprozess eingebaut ist (Abbildung 13.3). Ohne einen entsprechend modularen Aufbau der technischen Architektur wäre diese Veränderung nicht möglich gewesen und der Übersetzer GCC langfristig nicht mehr brauchbar. In Abbildung 13.3 ist die logische und physische Gliederung des GCC-Kern dargestellt.
Bemerkenswert ist an Abbildung 13.3, dass der RTL-Assembler bei der Entstehung des Übersetzers GCC 1984 noch nicht existiert haben kann. Denn ursprünglich war der Übersetzer GCC von Richard Stallman nur für eine Plattform und als „schmales“ Werkzeug gedacht. Ebenso sind viele andere Module im Übersetzer GCC erst mit der Zeit entstanden; die technische Architektur hat sich weiterentwickelt. Ganz ähnliche Veränderungen in der technischen Architektur finden sich auch in anderen FOSS-Projekten.
Ein Weg, um Modularisierung zu erreichen, ist Quelltextaufteilung (code splitting). Dabei wird die Komplexität des Quelltextes duch geschickte Aufteilung von Programmteilen in Module reduziert. Als Maß für diesen Ansatz wird in [BP03] die Kostenschätzung nach COCOMO [Boe81] verwendet. Wachsen die Kosten eines Software-Projekts noch super-linear mit seiner Größe, so läßt sich für die Modularisierung folgendes zeigen [BP03]:
Zunächst werden die Kosten eines Projektes mithilfe der COCOMO-Methode zu $A \cdot n^B$ modelliert. Dabei ist n das Maß für die Größe des Systems in KLOC und es gilt $A, B > 1$. Sei nun ein Quelltext mit n Modulen gegeben, worin die Programmabhängigkeiten quadratisch in der Komplexität wachsen: $B = 2$. Dann ergibt sich für die Gesamtkomplexität des Projektes: $C_{gesamt} = O(n^2)$. Wird stattdessen der Quelltext auf l Module aufgeteilt, so reduziert sch die Gesamtkomplexität deutlich zu: $C_l = O(\frac{n^2}{l} + c \cdot l)$. Der Faktor c kennzeichnet die neuen Vermittlungskomponenten zwischen den neuen Modulen. Da c ein konstanter Faktor und l nicht quadratisch ist, kann der zweite Teil der Gleichung bei der Betrachtung vernachlässigt werden. Ebenso ist für die l neuen Komponenten nicht zu erwarten, dass sie quadratische Komplexität besitzen — interessanterweise folgen die Maintainer bei FOSS-Projekten intuitiv regelmäßig diesem Muster, um die Verständlichkeit des Quelltextes zu gewährleisten.
Modularisierung kommt heute in vielfältigen Formen vor: Zugunsten schlanker Kern-Programme wird das Plug-In Konzept genutzt. Somit kann der Basisfunktionsumfang eines Programms beliebig angepasst (reduziert) werden und definierte Schnittstellen sichern das reibungslose Zusammenwirken.
Teilweise kommt es vor, dass einzelne Teile aus FOSS-Projekten in Subprojekte ausgegliedert werden. Das wirkt sich auf die personelle Architektur aus. Wie die konkreten Ansätze des Aufteilens und Zusammenfügens in den einzelnen FOSS-Projekten definiert sind, ist in [Sca04] beschrieben.

13.6.2 Schichtenkonzept und Schnittstellen

Da FOSS-Projekte meist eine lange Entwicklungsgeschichte hinter sich haben, stehen häufig Abhängigkeiten innerhalb der FOSS-Projekte dem Modularisieren entgegen.

Einfache Quelltextaufteilung ist meist nicht möglich. Deshalb müssen die betreffenden Teile entzerrt, ggf. umgeschreiben und neu gegliedert werden. Dabei entstehen Module mit wohldefinierten Schnittstellen, die dann als Schichten miteinander interagieren; wie z.B. in Abbildung 13.3 dargestellt.
Durch die entstandenen Schichten kann sich eine Umorganisation im Projekt ergeben. Formuliert wurde diese Behauptung in [BP03], wonach „jede wichtige Änderung in der Organisation des Projektes zwangsläufig zu einer Veränderung der technischen Architektur führen [muss] und umgekehrt“. Als Beispiel wird der Übersetzer GCC angegeben. Dort sei, getrieben durch Notwendigkeit und nicht durch Autorität sondern durch agiles Vorgehen, eine Veränderung der Schichten vollzogen worden. Entsprechend sei die Evolution der Architektur in FOSS-Projekten ein natürlicher und gesunder sowie gesundender Vorgang.
Bei diesem Vorgang werde der Lösungsansatz gewählt, der der einfachste, anwendbarste und vollständigste sei. Wie es zu dieser Entscheidung komme, hänge von den internen Strukturen der Projekte ab (siehe Abschnitt 13.5.1). Ob und wie ausführlich die Diskussionen vor der endgültigen Übernahme der Änderungen gestaltet sind hängt auch von den eingesetzten Medien ab.
Ist dann eine „gute“ Struktur gefunden, können die einzelnen Schichten für sich betrachtet und bearbeitet oder ausgetauscht werden. So ist beispielsweise das Mozilla-Projekt genau nach dieser Beschreibung vorgegangen, als der Quelltext von der Netscape Communicator Suite übernommen war. Befand sich der Quelltext zu Beginn noch in einem unwartbaren Zustand [CB99], erreichte die Strukturierung nach dieser Art einen wartbaren Zustand, auf den dann eine weitere Technik angewandt werden konnte.
Denn heute besteht der Quelltext der Mozilla-Projekte nur noch zu geringen Teilen aus dem damals übernommenen Quelltext – von Grund auf neu geschrieben wurde allerdings nicht. Stattdessen wurden peu à peu Quelltextfragmente überarbeitet und ausgetauscht. Dieser Prozess wird als inkrementelles Neuschreiben (incremental rewriting) bezeichnet. In diversen Fallstudien [DB04], [CB99] usw. ist nachgewiesen worden, dass beim inkrementellen Neuschreiben eine vergleichsweise geringe Fehlerdichte entsteht. Deshalb ist auch während dieser Zeit gewährleistet, dass ein konkurrenzfähiges Programm existiert.
Deswegen wäre dieses Vorgehen bei vielen Closed Source Anwendungen ein Profit. Jedoch bestimmen dort viele andere, zusätzliche Faktoren die zugehörigen Entscheidungen und nicht das „Wohl“ des Projektes. In der FOSS bedeutet dieses Vorgehen, dass eine recht hohe Redundanz in der Entwicklung vorhanden ist und eher eine schleichende Verwandlung, eben Evolution, stattfindet. Ob das so in Closed Source Entwicklungen gewünscht ist, bleibt zu untersuchen.

13.6.3 Globale Softwareentwicklung und Qualitätssicherung

Wie in Abschnitt 13.5.1 angedeutet, spiegeln die Projektstrukturen bei Closed Source Programmen oft die geographische Gliederung von Firmen wieder. Insbesondere ist leicht vorstellbar, dass das nicht die Interessen oder Fähigkeiten der Mitarbeiter zusammenfasst. Deshalb wird diese Art der Projektorganisation meist als produktori-

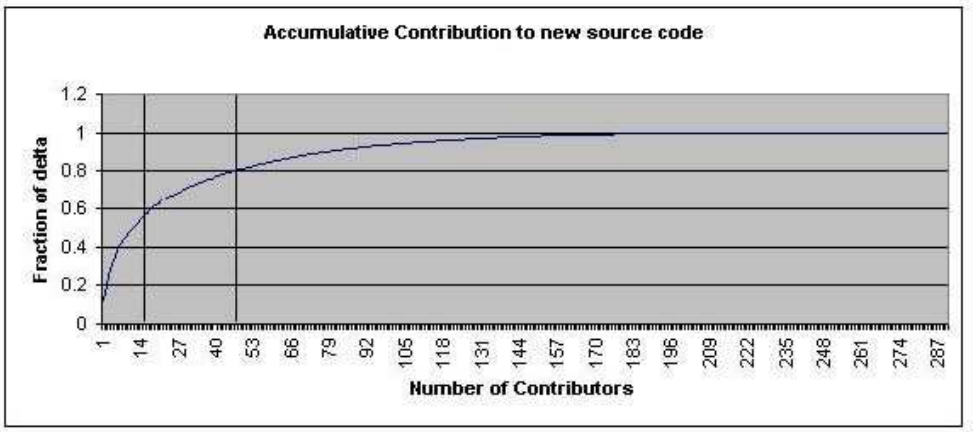


Abbildung 13.4: Beiträge zum Quelltext von FreeBSD aus [DB04]

entiert bezeichnet und ist weitgehend statisch, d.h. es findet kaum Evolution der Architektur in personeller oder technischer Hinsicht statt. Damit läuft dieses Verhalten oft dem Zweck der Modularisierung zuwider. Teilweise bestehen diese Zergliederungen aus gutem Grund. Denn oft stellen Probleme interkultureller Kommunikation eine Barriere dar. Bei FOSS-Projekten reicht das Projekt als „höheres“ Ziel oft, um über diese Differenzen hinwegzusehen.
In der Fallstudie von Cubranic [CB99] über Mozilla wurde gezeigt, dass die Änderung der Architektur die globale Softwareentwicklung möglich gemacht haben. Doch sind solche Veränderungen in Closed Source schwer. So hat Netscape diesen Versuch durchgeführt und ist damit gescheitert. Nachteilig an der Situation von Netscape war allerdings, dass die Communicator Suite unter Zeitdruck komplett neu geschrieben werden sollte. Der Umfang dieses Vorhabens und der Zeitaufwand wurden von Netscape falsch eingeschätzt — In der FOSS Community ist dieses Vorhaben dann geglückt. Nachdem die technische Architektur überarbeitet war, wurde per inkrementellem Neuschreiben die Evolution vorangetrieben.
Folglich ist es im kommerziellen Bereich, bedingt durch die Marktsituation, nicht immer möglich ein Programm komplett neuzuschreiben. Beim inkrementellen Neuschreiben gibt es allerdings nur einen langfristigen Return-of-Invest, der keine kurzfristigen Erfolge und Kunden erzielt. Die sind im kommerziellen Umfeld zwingend erforderlich [LR01]. Der langfristige Nutzen steht außer Frage.
Insgesamt ist es keine triviale Aufgabe eine „alte“ / „umfangreiche“ Architektur zu überarbeiten oder auszutauschen. Auf die Fallstudie bezogen kommen Bauer und Pizka [BP03] zu dem Schluss, dass „in freier Umgebung das Vorgehen wegen der flexiblen, selbstgesteuerten Evolution sehr erfolgreich [war]. In der kommerziellen Umgebung scheiterte das Vorhaben aufgrund fehlgeleiteter und unrealistischer Planung.“
Denn, bei der Entwickung von FOSS kann zeitweilig auf neuen Funktionsumfang verzichtet werden. Denn hier sind die Anwender (Kunden) gleichzeitig Entwickler [MFH02] (sechste These) und somit ist die Nutzung und damit der Produktivtest des Programms sichergestellt. Entsprechend können die Konzepte, der vorangegangenen Abschnitten „in Ruhe“ eingesetzt werden (Quelltextaufteilung, Plug-In Konzept, in-

krementelles Neuschreiben).
Außerdem gibt es bei FOSS eine andere „Test-Kultur“. Schon aus Abbildung 13.1 wird deutlich, dass es bei FreeBSD außer den Entwicklern (Committern) noch eine deutlich größere Anzahl an Menschen gibt, die ihre Beiträge in Form von Fehlerbeschreibungen o.ä. einbringen. So kommen bei FreeBSD auf etwa 250 Committer etwa 3000 Contributoren. Zeigt Abbildung 13.4 noch, dass rund vierzehn Entwickler gut sechzig Prozent der neuen Ideen realisieren (Mockus erste These - abgeschwächt), schaffen vierzehn Entwickler bei der Fehlerbehebung deutlich weniger als vierzeig Prozent der Arbeit (Abbildung 13.5). Das stützt Mockus dritte These, wonach die Gruppe der Fehlerbeheber in FOSS größer ist als die der Entwickler [MFH02]. Nochmals größer ist die Gruppe der Tester, die bei FreeBSD die Contributoren sind. Eine schnelle Reaktionszeit auf Fehler konnte von Dinh-Trong aus Mangel an Daten nicht gezeigt werden, trifft allerdings in der Fallstudie von Mockus über Apache und Mozilla zu [MFH02].
Um auf Lehmans Gesetze der Softwareevolution zurückzukommen: FOSS stützt die Gesetze eins, zwei und sechs trivialerweise (Abschnitt 13.3). Zudem ist das dritte Gesetz (Self Regulation) eine der obigen Kernaussagen. Das vierte Gesetz (Conservation of Organisational Stability) ist anhand der vorliegenden Informationen schwer zu untersuchen. Beim fünften Gesetz (Conservation of Familiarity) widersprechen das inkrementelle Neuschreiben und die flexible, modulare Architektur. Gesetz sieben (Declining Quality) widerspricht der Qualitätsaussage über FOSS und trifft deshalb für die erfolgreichen Projekte nicht zu. Beim achten Gesetz (Feedback System) ist die Gültigkeit wiederum offensichtlich, da die Rückmeldungen der Anwender und iterative Zyklen direkten Einfluss auf die Arbeit der Entwickler haben (Abbildung 13.2).

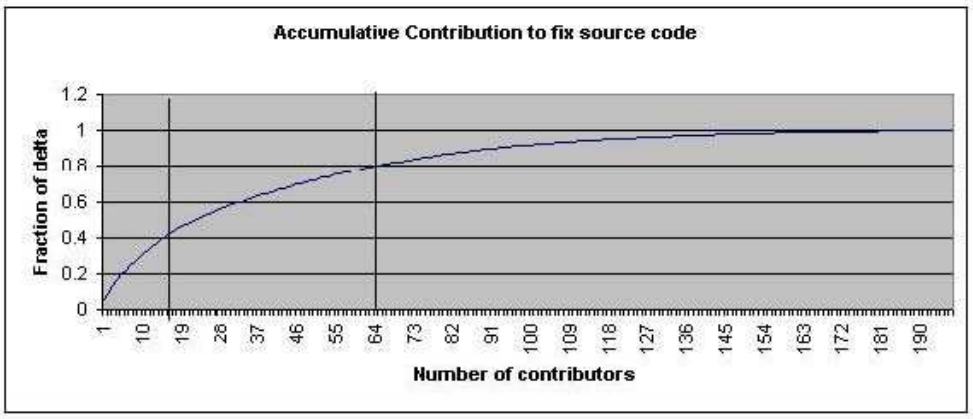


Abbildung 13.5: Beiträge zur Fehlerbehebung von FreeBSD aus [DB04]

13.7 Fazit

Wie schon in der Motivation beschrieben, ist es ein Fakt, das einige FOSS-Projekte erfolgreich sind. Doch stellt sich die Frage, woher der Erfolg kommt.

Einen Teil der Antwort liefert der vorliegende Text. Denn die Entwicklungsprozesse und deren dynamische Anpassung an die Anforderungen sowie das Skalieren der organisatorischen Struktur sind ein essenzieller Bestandteil der FOSS-Projekte. Beginnt ein FOSS-Projekt ohne organisatorischen Aufwand, wächst dieser stetig mit der Projektgröße. Deshalb ergeben sich durch Notwendigkeiten getriebene Veränderungen in der technischen Architektur, die auch Veränderungen in den Prozessen nach sich ziehen. Ebenso verändern sich die Prozesse aufgrund der „neuen“ Werkzeuge, sodass insgesamt hochwertige Programme entstehen.
Können die Ideen der FOSS-Projekte auf den kommerziellen Bereich angewandt werden? Die Antwort bedarf sicherlich noch ausgiebiger Forschung, doch erscheint folgende Argumentation sinnvoll: Die Ideen sind nur eingeschränkt übertragbar, da redundante Entwicklungsprozesse, wie in der FOSS, kaum ein Arbeitgeber/Auftraggeber bezahlen wird. Zudem entsteht die Gefahr des „suspenders and belts“-Ansatz. Einer verlässt sich auf die Arbeit des anderen.
Eine These, warum die Prozesse in der FOSS-Entwicklung funktionieren, könnte lauten: „Die Motivation eines FOSS-Entwicklers ist von ideeller Natur, da es sich um keine Pflichtaufgabe handelt.“ Doch sind diese Aspekte zahlreich untersucht und unter dem Stichwort „intrinsische Motivation“ bekannt.
Dass Programme in Dialog und Diskussion entstehen sollten, wie es in der FOSS-Entwicklung üblich ist, erschließt sich von selbst. Doch auch hier wird die Übertragung auf den kommerziellen Bereich schwierig. Denn dort würden durch dieses Vorgehen schnell vorhandene Autoritäten untergraben [Hof96]. Dass diese Diskussionen in FOSS-Projekten funktionieren, mag daran liegen, dass dort keine solch starken Hierarchien existieren, wie sie über Jahre in Unternehmen wachsen. Denn schlimmstenfalls sucht sich ein „verprellter“ Entwickler eines FOSS-Projektes ein anderes FOSS-Projekt. Das bestätigt auch die Studie von [HO02], wonach 60% der FOSS-Entwickler in mehr als einem FOSS-Projekt [2-10] aktiv sind. Entsprechend hoch könnte die Dynamik der Entwicklerfluktuation erwartet werden. Dem ist nach [Sca04] allerdings nicht so, da Rücksicht hier mehr zählt. Es wird sogar noch mehr gezeigt: Es existiert sogar ein sehr verwobenes Netzwerk unter den FOSS-Entwicklern.
Doch zurück zu den dynamischen Veränderung von Prozessen in FOSS-Projekten. Diese könnten und sollten auch im kommerziellen Umfeld Anwendung finden. Ob in großen Unternehmen oft ganz andere Gründe bei Projektorganisation usw. eine Rolle spielen, als das „Wohl des Projektes“ bleibe hier vernachlässigt.
Des Weiteren erscheint die dynamische Evolution der technischen Architektur im kommerziellen Bereich möglich. Wünschenswert ist sie bestimmt, doch ist hier noch viel Diskussion nötig. Dass sich dadurch die Qualität der Produkte steigern lässt, ist klar.
Insgesamt lässt sich feststellen, dass die Bedingungen in FOSS-Projekten und im kommerziellen Bereich sehr unterschiedlich sind und deshalb erfolgreiche Praktiken schwer übertragbar sind oder eine enorme Anpassung erfordern.

Danksagung

Ein besonderer Dank geht an Max Möllers und Holger Schackmann für die vielen Hinweise und Korrekturvorschläge. Für die Anmerkungen zu Sprache, Ausdruck, FreeBSD und darüber hinaus geht ein ganz besonderer Dank an Christian Brüffer. Jeder Tippfehler, der jetzt im Text zu finden ist, kann erst nach seiner Korrektur entstanden sein.

Literaturverzeichnis

[Boe81] B. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[BP03] Bauer, Andreas and Pizka, Markus. The Contribution of Free Software to Software Evolution. *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of Publication*, pages 170 – 179, 2003.

[CB99] Cubranic, D and Booth, K.S. Coordinating Open-Source Development. pages 61–65, 1999.

[Com98] GCC Steering Committee. Announcement from November 10, 1998, 1998.

[CUH99] Cooke, D, Urban, J., and Hammilton, S. Unix and beyond: An interview with Ken Thompson. pages 32(5):58–62, 1999.

[DB04] Dinh-Trong, Trung and Bieman, James M. Open Source Software Development: A Case Study of FreeBSD. *International Symposium on Software Metrics (METRICS04)*, 2004.

[Fre91] Free Software Foundation Inc. Cambridge, Massachusetts. GNU General Public License, 1991.

[GT00] Godfrey, M.W. and Tu, Q. Evolution in Open Source Software: A Case Study. *International Conference Software Maintenance (ICSM-00)*, pages 131–142, 2000.

[He02] Harris, N and et al. Linux Handbook / A guide to IBM Linux solutions and resources. *IBM International Technical Support Organization*, 2002.

[HO02] Hars, A. and Ou, S. Working for Free? Motivations for Participating in Open Source Software Projects. pages 6(3), 25–39, 2002.

[Hoe00] Hoeck, A. van der. Configuration Management and Open Source Projects. *Proceedings of the 3rd Intenational Workshop on Software Engineering over the Internet, Limerick, Ireland*, 2000.

[Hof96] Hofstede, Geert H. *Cultures and Organizations - Software of the Mind Intercultural Cooperation and Its Importance for Survival*. McGRAW-HILL, 1996.

[Int95] Standish Group International. CHAOS, 1995.

[Int99] Standish Group International. CHAOS: A Recipe for Success, 1999.

[Leh02] Lehey, G. Evolution of a Free Software Project. *Proceedings of the Australian Unix User's Group Annual Conference, Melbourne, Australia*, pages 11–21, 2002.

[LR01] Lehman, M.M. and Ramil, J.F. Rules and Tools for Software Evolution Planing and Management. *Anuals of Software Engineering*, 2001.

[MFH02] Mockus, Audris, Fielding, Roy T., and Herbbsleb James D. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Trans. Software Engineering and Methodology*, pages 309–346, 2002.

[Net06] Netcraft. http://www.netcraft.com/survey/, 2006.

[NYNY02] Nakakoji, K, Yamamoto, Y, Nishinaka, Y., and Ye, Y. Evolution Patterns of Open-Source Software Systems and Communities. *Proceedings of the international workshop on Principles of software evolution*, pages 76–85, 2002.

[Per99] Perkins, J. Cultural Clash and the Road to World Domination. pages 16(1):23–25, 1999.

[Piz97] Pizka, Markus. Design and Implementation of the GNU INSEL Compiler gic. *Technical Report TUM I-9713*, 1997.

[Ray98] Eric Steven Raymond. *Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, 1998.

[Sae06] Saers, Niklas. A project model for the FreeBSD Project, 2006.

[Sca04] Walt Scacchi. Understanding Open Source Software Evolution. *Institute for Software Research*, 2004.

[Sou] SourceForge. http://sourceforge.net/.

[Uni98] University of California, Berkeley. The BSD License, 1998.

[Whe02] Wheeler, D.A. More Than a Gigabuck: Estimating GNU/Linux's Size. 2002.