Wertgetriebene Softwarewartung

Harry M. Sneed

Testabteilung
ANECON GmbH, Wien
harry.sneed@t-online.de

Abstract: Der folgende Beitrag überträgt das Konzept von "value-based software engineering" wie vom Boehm ursprünglich vorgeschlagen auf die Softwarewartung. Zunächst werden verschiedene Ansätze zur Bewertung eines Softwaresystems und zur Berechnung eines "return of investment (ROI)" vorgestellt. Anschließend schlägt der Autor eine Bewertungsmethodik vor, die auf Hayek's Theorie der Werterhaltung von Kapitalgüter in einer schnell wandelnden Wirtschaft fasst. Einerseits werden die Kosten von Wartungsarbeiten auf der Basis einer Impactanalyse kalkuliert. Anderseits wird der Nutzen, der durch die Wartungsarbeiten entsteht, mit Hilfe der Hayekischen Wertsteigerungstheorie berechnet. Kosten und Nutzen fließen in die Kalkulation der Wartungsamortisationsberechnung ein. Eine Fallstudie aus der industriellen Praxis illustriert die Anwendung des Ansatzes.

1 Value driven Software Engineering

Der "value-based" Software-Engineering Ansatz kombiniert eine Reihe altbewährter Software-Engineering- Prinzipien darunter "participatory design", "user engineering", "cost estimation", "software economics", "software investment analysis" und "software engineering ethics". Wertgetriebene Software-Engineering umfaßt u.a. folgende wertgetriebene Aktivitäten:

- Wertgetriebene Requirements-Engineering mit der Gewichtung und der Prioritätssetzung der Anforderungen nach Wirtschaftlichkeitskriterien
- Wertgetriebene Softwarearchitektur mit der Versöhnung der Systemziele mit erreichbaren architektonischen Lösungen.
- Wertgetriebene Software-Design, mit Schwerpunkt auf die Auslegung des Designs um den Projektzielen und Wirtschaftlichkeitsüberlegungen gerecht zu werden.
- Wertgetriebener Test nach dem die Funktionen mit dem höchsten am intensivsten getestet werden.
- Wertgetriebene Projektplanung- und -steuerung mit dem Vorziehen jener Aufgaben die den höchsten Kundenwert versprechen.
- Wertgetriebene Risikoanalyse bei der jene Risiken mit dem höchstmöglichsten betriebswirtschaftlichen Verlustpotential identifiziert werden.

- Wertgetriebenes Qualitätsmanagement durch die Bestimmung der Qualitätsziele nach ihrem Nutzen für den Kunden.
- Wertgetriebene Entwicklungsprozesse in denen der Kundenwert im Mittelpunkt der Arbeitsaufteilung steht.

Diese Forschungsagenda wird von der "Economics-driven Software Engineering research community" mit der Website (www.edser.org) vorangetrieben [Blom08].

Der Kerngedanke der wertgetriebenen Softwareentwicklung ist, dass jeder einzelne Arbeitsschritt ein Ergebnis hervorbringen sollte, der an und für sich einen Wertgewinn für den Auftraggeber darstellt, d.h. jedes Dokument, jedes Werkzeug, jedes Teilmodell, jede Codekomponente und jeder Testfall soll sich wirtschaftlich rentieren. Es soll ein positives „Return on Investment" haben nach der Gleichung:

$$ROI = (benefit - cost) / costs$$

Jedes Arbeitsergebnis, egal wie klein, stellt einen Wert dar. Es verursacht auch Kosten. Seine Kosten dürfen seinen Wert niemals überschreiten. Ist dies der Fall, darf das Ergebnis nicht erst produziert werden. Es rentiert sich nicht. Ergebnisse mit dem höchsten ROI sind vorzuziehen. Jene mit einem niedrigen ROI sind zurückzustellen. Gerade im Zeitalter der agilen Softwareentwicklung sollen Projektverantwortlichen eine Leitlinie haben nach der sie entscheiden können was als nächstes zu machen ist. Diese Leitlinie ist der Wertbeitrag der jeweiligen Aufgaben [BoTu05].

In einem iterativen Entwicklungsprojekt gehören jene Bausteine mit dem höchsten Wertbeitrag in der ersten Iteration. Nach jeder Iteration werden die anstehenden Aufgaben neu bewertet um den Inhalt der nächsten Iteration zu bestimmen. Der Kundenwertbeitrag ist das Kriterium nach dem die Ziele einer jeden Iteration, bzw. eines jeden Sprints, gesetzt werden. Dies schließt technische Überlegungen nicht aus, z.B. wenn eine Komponente die technische Voraussetzung für die Nächste ist, aber die wirtschaftliche Überlegung soll immer im Vordergrund stehen. Sie bestimmt die Reihenfolge der Implementierung und die Intensivität des Tests, die sich in der Testüberdeckung ausdruckt [Mell05].

In seinem Buch "Value-based Software Engineering" fasst Biffl die Hauptfaktoren bei der Bewertung einer Software-Engineering Aktivität zusammen. Diese sind:

- der Nutzen eines Produktes oder einer Dienstleistung minus die Kosten zur Bereitstellung derselben.
- der prozentuale Beitrag der Aktivität zum Nutzen des Gesamtproduktes bzw. zur gesamten Dienstleistung
- die Kosten der Aktivität relativ zu den Gesamtkosten des Produktes, bzw. der Dienstleistung.

Da Nutzwert relativ ist, muss der Nutzwert einer jeden Aktivität im Bezug zum Nutzwert des Ganzen betrachtet werden. Aktivitäten, die mehr Nutzen bei weniger Kosten liefern, sind vorziehen. Dies ist auch ein Grundsatz der agilen Softwareentwicklung [Biff06].

Die Voraussetzung für wertgetriebene Software-Engineering ist die Fähigkeit Nutzen, Kosten und Risiken zu quantifizieren. Wenn Nutzen nicht quantifizierbar ist, ist sie auch nicht mit den Kosten vergleichbar. Demzufolge, ist es kaum möglich wertgetriebene Software-Engineering ohne Vermessungssystem einzuführen. Es muss möglich sein, die Funktionalität und Qualität der Software die man bereits hergestellt hat zu messen, um die Funktionalität und Qualität der geplanten Software zu projektieren. Erst wenn diese Voraussetzung erfüllt ist, kann man dazu übergehen Kosten, Nutzen und Risiken miteinander zu vergleichen. Für den wertgetriebenen Ansatz muss sich die Software in Zahlen auslegen lassen [Soli04].

2 Softwarewartung als Werterhaltung von Kapitalgütern

Die Theorie der relativen Werterhaltung geht zurück auf die Forschung des deutschen Volkswirt Franz Schmidt im 19. Jahrhundert. Schmidt vertrat die These, dass der wahre Wert eines Kapitalguts nur relativ zu den anderen Konkurrenzgütern auf dem Markt zu ermitteln wäre. Das hängt wiederum von anderen Faktoren ab wie die Inflationsrate, die Wirtschaftslage und die Marktentwicklung. Nach Schmidt hängt der Wert eines Wirtschaftsgutes von der Verfügbarkeit vergleichbarer Güter ab, vorausgesetzt es hat überhaupt einen Wert. Wo keine Nachfrage ist, ist auch keinen Wert. Ein Produkt hat nur so viel Wert, wie der Benutzer des Produktes ihm beimisst [Schm22].

Im 20. Jahrhundert griff der österreichische Volkswirt Friedrich Hayek Schmidt's These auf und erweiterte sie auf die Erhaltung von Gütern. Hayek interessierte sich für den Wert der Kapitalerhaltung. Ihn ging es im Gegensatz zu den anderen Volkwirten seiner Zeit weniger um die Schaffung neuer Kapitalgüter als viel mehr um die Erhaltung bestehender Kapitalgüter. Er wollte wissen, was es kostet die Güter zu erhalten relativ zu den Kosten einer Neubeschaffung. Hayek schrieb "The question is, how much of the gross receipts from his capital income does the entrepreneur have to reinvest in order to ensure a constant flow of income? Either he may reduce his investment to a minimum until the value of his product has been depreciated by change, or he may reinvest amortization quotas of the same on an increasing magnitude as before." [Haye39]

Auf die Erhaltung von Softwaresystemen übertragen, heißt das, die Besitzer eines Softwaresystems können die Wartungskosten auf ein Minimum reduzieren und zuschauen wie das System im Sinne der Gesetze der Softwareevolution von Belady und Lehman immer weniger Wert wird, oder sie können Kapital verhältnismäßig zu dem was sie in die Entwicklung gesteckt haben in die Verbesserung des Systems investieren. Durch regelmäßige Sanierungen und funktionale Erweiterungen können sie den Wert ihrer Software nicht nur erhalten sondern auch über die Zeit steigern. Ansonsten nimmt der Wert ihres Systems ständig ab. Zum Einen, fehlt die Funktionalität immer weiter hinter den Anforderungen der Benutzer zurück. Um die dringlichsten Anforderungen gerecht zu werden, wird die Software notdürftig geändert und erweitert. Es werden aber auch immer neue Mängel entdeckt. Diese Mängel müssen beseitigt werden.

Die Behebung der Mängel und die Änderung bzw. Erweiterung der Funktionalität werden als Softwarewartung bezeichnet. Mit jedem Eingriff in die Software, ob zur Korrektur eines Fehlers oder zur Änderung einer Funktion, steigert die Komplexität jener Software. Es entstehen neue Beziehungen zwischen Codebausteinen und dadurch mehr Abhängigkeiten relativ zur Codegröße. Gleichzeitig sinkt die Qualität in dem Maße wie die Software sich immer weiter von ihrer ursprünglich beabsichtigten Struktur entfernt. Steigende Komplexität und sinkende Qualität bedeuten Wertverlust. Die Software wird immer weniger Wert [BeLe85].

Hayek erkannte die Verantwortung der Güterbesitzer für die Werterhaltung ihrer Güter. In diesem Zusammenhang betonte er die Notwendigkeit voraus zu planen, wenn es um die Erhaltung der Güter geht. Er fasst die Pflicht zur Werterhaltung wie folgt zusammen: "All this means that the mobility of capital, i.e. the degree to which it can be maintained in a changing world depends on the foresight of the entrepreneurs…It also means that the amount of capital available at any moment in a dynamic society depends much more on the foresight of the entrepreneurs in constructing maintainable goods than on current cost savings or on time to market" [Haye41].

Der Softwaremarkt ist zweifelsohne ein dynamischer Markt und Software als Gut nimmt schnell an Wert ab, wenn sie nicht ständig erneuert wird. Nach der Lehre von Hayek ist der Wert der Softwarewartung gleich der Höhe der Produktabwertung wenn es nicht gewartet wird. Ein Softwaresystem mit einem momentanen Nutzwert von € 10 Million und einer jährlichen Abwertung von 10% wird schon nach 5 Jahren die Hälfte seines Wertes verloren haben und nur noch € 5 Million wert sein wenn es nicht erneuert wird. Um diesen Wertverfall zu verhindern musste in der gleichen Zeit € 5 Million in die Softwarewartung reinvestiert werden. Die gängigen Wartungsgebühren in der Höhe von 15-25% des Kaufwertes bestätigen die Notwendigkeit dieser Investition. Wenn schon die Hälfte dieser Gebühren gegen die ursprünglichen Entwicklungskosten gebucht wird, bleiben noch circa 10% für die Investition in die Werterhaltung, bzw. in die Evolution, übrig. Ein Anwender selbstentwickelter Softwaresysteme müsste demzufolge bereit sein mindestens 10% vom dem was er in die Entwicklung jener Systeme investiert hat, jährlich für die Erhaltung jener Systeme bereitzustellen. Diese Erhaltungskosten müssen auch in die Berechnung der Amortisation der Produkte einfließen.

Der deutsche Volkswirt Lachmann hat die Lehre von Hayek durch den Begriff der Komplementarität ergänzt. Güter sind komplementär wenn der Wert des einen Gutes vom Wert der anderen Güter abhängt. Die Abwertung von Software ist mit dem Begriff komplementärer Systeme zu erklären, d.h. Systeme die gegenseitig abhängig sind. In der Betriebswirtschaft sind die meisten Produkte komplementär. Sie stützen sich aufeinander. Wenn ein Produkt an Wert abnimmt, zieht er den Wert der anderen Produkte mit herunter. Es geht also nicht nur um den Wert eines einzelnen Systems. Sofern die Systeme mit einander integriert sind, geht es um den Wert aller Systeme. Es muss nur ein System an Wert verlieren um den Wert des ganzen Betriebes herunterzuziehen. Dies ist der Preis eines hohen Integrationsgrades.

Software existiert und funktioniert in einer komplementären Umgebung in der fast alles von allem abhängig ist. Diese Umgebung, bzw. die Systemlandschaft eines Unternehmens ist eine Kapitalstruktur die sich mit der Zeit wandelt. Neue Systeme werden eingeführt, andere Systeme werden ersetzt oder erneuert. Wenn ein System stehen bleibt, hat dies eine negative Auswirkung auf die Andere. Ihr Wert wird dadurch vermindert. Da die Systeme komplementär sind müssen, um den Wert des Ganzen zu erhalten, alle Teilsysteme in etwa den gleichen Stand behalten. Zur Werterhaltung von Software gehört die gleichmäßige Evolution aller komplementären Systeme. Lachmann schreibt „for any particular type of capital good, maintenance is a matter of maintaining its complementarity to the rest of the changing capital structure. Hence, maintenance may mean not only preventing any change through deterioration, but actually changing a good directly, in a manner that adapts it to the changing capital structure around it, thereby delaying obsolescence and increasing usefulness." [Lach75].

Weil Wandel in einem dynamischen Markt unausweichlich ist, müssen die Kapitalgüter sich auch wandeln. Hayek weist darauf hin, dass die Erhaltung von Kapitalgütern mehr darauf ausgerichtet ist den Wert jener Güter zu erhalten als den Verfall vorzubeugen. Sowohl Hayek als auch Lachmann sind der Meinung dass komplementärer Produkte mit der Zeit zunehmend komplexer werden. Sie stellen fest: „that over time there develops "an increasing degree of complexity of the pattern of complementary displayed by the capital structures..". Diese Beschreibung passt genau zu Softwareprodukten. Software altert wenn sie ihre Umgebung nicht weiter komplementiert. Der Wert eines Softwaresystems ist direkt proportional zu dem Grad in dem andere Systeme, einschließlich die manuellen Systeme, vom ihm abhängig sind. Wert hängt von der Bedeutung eines Produktes ab und Bedeutung hängt von der Abhängigkeit anderer Akteure im Umfeld des Produktes von diesem Produkt ab. Die Erhaltung eines Kapitalgutes in einer wandelenden Kapitalstruktur, verlangt dass das Gut nicht nur in seinem bisherigen Zustand bleibt, sondern weiterentwickelt wird um mit seiner veränderten Umgebung Schritt zu halten [Baet98].

Nach Hayek lässt sich Kapital als verpacktes Wissen definieren. Softwareprodukte sind demnach Kapitalgüter bei denen das Wissen im Code verpackt ist. Das Wissen das im Code steckt ist das Ergebnis eines organisatorischen Lernprozesses, der sich über Jahre hinstrecken kann. Durch gemeinsames Probieren kommt eine Arbeitsgruppe endlich zu einem annehmbaren Ergebnis. Deren Zwischenergebnisse verkörpern das Wissen, das bis zu diesem Zeitpunkt gesammelt wurde [Tock05]. Deshalb haben sie einen Nutzwert auch wenn sie nicht voll einsatzfähig sind. Der lange Prozess bis hin zu einem brauchbaren Softwareprodukt ist sehr kapitalintensiv. Er bindet Geld und Ressourcen. Drum müsste es ein Hauptanliegen des Kapitalbesitzers sein, dieses Wissen zu erhalten und zu erweitern. Wer dies nicht tut verschwindet Kapital. Der Zweck der Softwarewartung ist die Erhaltung und Vermehrung der Funktionalität eines Softwareproduktes. Der Zweck von Software-Reengineering ist die Erhaltung und Vermehrung der Produktqualität [Snee91].

3 Anwendung wertgetriebener Software Engineering für Softwarewartung

Softwarewartung so wie sie von der IEEE Standard-1219 formuliert ist "includes modification of a software product after delivery or to adapt the product to a modified environment" [IEEE93]. Die Literatur zum Thema Softwarewartung identifiziert vier Klassen von Wartungsaufgaben:

- korrektive Wartung
- adaptive Wartung
- perfektive Wartung
- enhancive Wartung.

Korrektive Wartung befasst sich mit der Behebung von Fehler und sonstiger Mängel. Adaptive Wartung umfasst die Änderungen bestehender Funktionen und Daten, also die Anpassung der Software an geänderte Anforderungen, bzw. die Bearbeitung von Change Requests. Perfektive Wartung zielt auf eine Steigerung der Softwarequalität bei gleichbleibender Qualität. Restrukturierung und Refaktorierung gehören zur perfektiven Wartung. Enhansive-Wartung bedeutet Erweiterung der Software. Zusätzliche Funktionen und/oder Daten werden hinzugefügt, die Größe der Software wächst [Chap01].

Wartungsaufgaben werden durch Fehlermeldungen, Änderungsanträge oder durch neue Anforderungen ausgelöst. Korrektive Wartung wird durch Fehlermeldungen bzw. Error-Reports, adaptive sowie perfektive Wartung durch Änderungsanträge, bzw. Change-Requests und erweiternde Wartung durch neue Anforderungen ausgelöst. Die drei Arten von Wartungsaufträgen sind demnach wie folgt:

Auftragsart		Wartungsart
error report	=	Korrektive
change request	=	Adaptive oder Perfektive
new requirement	=	Erweiterung

Wartungsaufgaben werden in Releases zusammengefasst. Jedes Release, bzw. jede Freigabe einer neuen Version, ist ein Teilprojekt im Lebenszyklusprojekt des Produktes. Zu entscheiden welche Aufgaben einem Release zugewiesen werden, müssen die Aufgaben einzeln bewertet werden, denn da das Produkt bereits im Einsatz ist, hat man immer die Wahl welche Änderungen man vornimmt und welche nicht. Hierbei kann wertgetriebene Software-Engineering eine Rolle spielen. Jede einzelne Aufgabe, sprich Fehlerkorrektur, Änderung, Nachbesserung und Erweiterung wird einer Nutzwertanalyse unterzogen um sie relativ zu den anderen anstehenden Aufgaben zu priorisieren. Dazu wird ihre ROI berechnet [Snee04].

Zur Kalkulation der RoI einer Wartungsaufgabe gehören drei Voraussetzungen:

a) die Aufwandsschätzung jener Aufgabe
b) die Risikoanalyse jener Aufgabe
c) der Nutzwertanalyse jener Aufgabe

4 Aufwandsschätzung von Wartungsaufgaben

4.1 Expertenschätzung

Les Hatton hat einen Artikel in der IEEE Software mit dem Titel “How accurately do Engineers predict Software Maintenance Tasks” veröffentlicht. Der Artikel basiert auf einer Studie von 957 Wartungsaufgaben bezogen auf 13 verschiedene Softwareprodukte in der Zeit vom März 2001 bis November 2005. 30% der untersuchten Wartungsaufgaben waren Korrekturen, 20% waren Änderungen und 50% waren Nachbesserungen oder Erweiterungen. Zwischen den letzten zwei Aufgabenarten hat Hatton nicht unterschieden. Die große Mehrzahl der Aufgaben, etwa 77%, nahm weniger als einen Personentag in Anspruch. Am anderen Ende der Aufwandsskala brauchten weniger als 5% der Wartungsaufträge mehr als vier Personentage. Der höchste Aufwand für eine Wartungsaufgabe betrug 44 Stunden.

Der Zweck der Studie war heraus zu bekommen wie gut Wartungsaufgaben von dem Wartungspersonal geschätzt werden. Hatton stellte fest, dass 75% der Aufgaben korrekt geschätzt und nur 16% grob unterschätzt worden. Das verbleibende 9% wurde leicht unterschätzt. Daraus folgt, dass nur eine von vier Wartungsaufgaben falsch eingeschätzt wurde, aber dass bei diesen die Mehrzahl der Schätzungen völlig daneben lag. Diese Fehlschätzungen waren fast alle von kleinen Aufgaben mit einem Aufwand von wenigen Stunden. Ergo, kann man daraus schließen, dass Wartungsingenieure ihre Aufgaben selbst ausreichend gut schätzen können [Hatt07].

4.2 Werkzeugschätzung

Ein automatisierter Vorgang zur Schätzung einzelner Wartungsaufgaben wurde vom Sneed im Rahmen des GEOS Projektes in Wien implementiert [Snee01]. Dieser begann mit der Textanalyse eines Wartungsauftrages – Fehlermeldung oder Change Requests. Aufgrund der erkannten Objekte im Text wurde der Wartungsauftrag einem bestehenden Anwendungsfall in der Software-Repository zugeordnet. Falls er sich nicht automatisch zuordnen lies, musste der zuständiger Analytiker ihn manuell zuordnen. Nachdem der betroffene Anwendungsfall einmal identifiziert wurde, konnte dieser über eine automatisierte Impaktanalyse zu den dazu gehörigen Modulen und Klassen zurückverfolgt werden. Die Beziehungen der Anwendungsfälle und Datenobjekten zu den Klassen und Datenbanktabellen waren in der Repository gespeichert.

Waren die betroffenen Komponenten einmal erkannt, konnte der zuständige Analytiker schätzen um wie viel Prozent sie geändert werden müssten. Der Änderungsanteil reichte in der Regel von 2 bis 40%. Mit einem Änderungsanteil von über 40% muss man sowieso davon ausgehen, dass der Modul oder Klasse ganz zu ersetzen ist. Der prozentuale Änderungsanteil wurde anschließend mit der Anzahl Anweisungen, bzw. Datenelemente, in diesem Baustein multipliziert um die Größe der Änderung zu projektieren.

Da die betroffenen Klassen auch von anderen Klassen abhängig sein können, entweder durch Vererbung oder durch Assoziation, werden die abhängigen Klassen auch in den Wirkungsbereich (Impact Domain) des Wartungseingriffes einbezogen. Bei ihnen wird allerdings der prozentuale Änderungsanteil geringer angesetzt als den der vollbetroffener Klassen. Dieser ist eine Funktion der Entfernung von den vollbetroffenen Klassen. Die Formel für die Ermittlung des gesamten Wirkungsbereiches ist:

$$\begin{aligned} \textit{Raw change size} = {} & \textit{(directly impacted class size * change rate)} \\ & + \textit{(}\Sigma\textit{ indirectly impacted class sizes * (change rate/4))} \end{aligned}$$

Die rohe Änderungsgröße wird sowohl durch die Komplexität als auch durch die Qualität der direkt und indirekt betroffenen Klassen justiert. Die Komplexität ist der gewichtete Mittelwert aller Komplexitätsmaße der betroffenen Klassen. Dazu gehören solche einzelne Komplexitätsmaße wie Ablaufkomplexität, Schnittstellenkomplexität, Zugriffskomplexität, Sprachkomplexität und Verknüpfungskomplexität, die auf einer rationalen Skala von 0 bis 1 eingestuft werden. Hohe Komplexität ergibt einen Wert über 0,5 während niedrige Komplexität einen Wert unter 0,5 ergibt. Zum Zweck der Vereinigung werden sämtliche Komplexitätsmaße wie Kopplungsgrad, Vererbungstiefe, zyklomatische Komplexität und Sprachvolumen in ein rationales Maß überführt. Die Komplexitätsjustierung ergibt sich aus der Teilung der gemessenen Komplexität durch den Mittelwert 0,5.

$$\begin{aligned} \textit{ComplexityAdjustment} = {} & \textit{measured Complexity/} \\ & \textit{medianComplexity} \end{aligned}$$

Danach folgt aus einem Komplexitätskoeffizient von 0,7 einen Justierungsfaktor von 1,4.

Das gleiche Verfahren wird auch für die Qualität des Wirkungsbereiches angewandt. Hier ist die Qualität des betroffenen Codes der gewichtete Mittelwert aller Einzelqualitäten – Modularität, Wiederverwendbarkeit, Flexibilität, Testbarkeit, Konformität, usw. Zum Zweck der Vergleichbarkeit werden auch sie auf ein rationales Skala konvertiert. Eine überdurchschnittliche Qualität ergibt einen Wert größer als 0,5, eine unterdurchschnittliche Qualität einen Wert darunter. Die Qualitätsjustierung folgt aus der Teilung der mittleren Qualitätsnote 0,5 durch die gemessene Qualität.

$$\begin{aligned} \textit{QualityAdjustment} = {} & \textit{MedianQuality/} \\ & \textit{measuredQuality} \end{aligned}$$

Danach ergibt einer Qualitätskoeffizient von 0,6 einen Justierungsfaktor von 0,83.

Die endgültige justierte Größe des Wirkungsbereiches der geplanten Änderung ist die rohe Anzahl betroffener Anweisungen justiert durch die Komplexität und Qualität des Codes.

$$\textit{adjusted impact size} = \textit{raw–size * (complexity / 0.5) * (0.5 /quality)}$$

Angenommen, die direkt betroffenen Klassen haben 800 Anweisungen und die indirekt betroffenen Klassen 4000 Anweisungen und die Änderungsrate wird auf 10% geschätzt. In dem Fall umfasst der Wirkungsbereich 180 Anweisungen.

*(800 + (4000/4) * 0.1 = 180 statements.*

Mit einem Komplexitätsjustierungsfaktor von 0,7 und einem Qualitätsjustierungsfaktor von 0,6 ist die justierte Größe der betroffenen Codemenge 209 Anweisungen.

*180 * [0.7/0.5] * [0.5/0.6] = 209 statements.*

Die justierte Größe des Wirkungsbereiches wird anschließend durch die bisherige Produktivität in Anweisungen pro Personentag dividiert um den Aufwand für die Wartungsaufgabe in Personentage zu ermitteln. Bei einer Produktivität von 20 Anweisungen pro Personentag wären das hier 209/20 = 10,5 Personentage für die Durchführung der Änderung.

Die Kosten eines neuen Releases ist die Summe der Kosten aller einzelnen Wartungsaufträge – Fehlerkorrekturen und Änderungen. Falls sich die Wirkungsbereiche zwei oder mehr Wartungsaufträge überschneiden werden die gemeinsam betroffenen Anweisungen nur einmal gezählt. Dadurch fallen die Gesamtkosten um etwas weniger als die Summe aller Einzelkosten aus, denn eine Modul oder Klasse wird nur einmal gezählt unabhängig davon wie viele Wartungseingriffe sie betreffen. Der Wirkungsbereich eines Releases ist die vereinigte Menge der Wirkungsbereiche aller einzelner Wartungsaufträge. Dieser wird durch die mittlere Produktivität dividiert um die Bruttokosten des Release zu ermitteln. Die endgültigen Nettokosten sind die Bruttokosten justiert durch die projektspezifischen Bedingungen aus dem COCOMO-Modell wie Teamzusammengehörigkeit, Prozessreife, Werkzeugunterstützung und Erfahrung mit dem Produkt. Hinzu kommt ein Overhead-Faktor für das Produktmanagement um noch 20 bis 30%. Zusammenfassend sind die Kosten eines neuen Releases:

*{Betroffene Anweisungen / Produktivitätsrate} * Einflussfaktoren * Overheadfaktor*

Zu empfehlen ist es, zweigleisig vorzugehen und die Schätzung aus der automatisierten Impakt-Analyse mit der Expertenmeinung der Wartungsingenieure zu vergleichen. Man benutzt die eine Methode um die Andere zu bestätigen. Durch diesen dualen Ansatz kommt man zu einer plausiblen Vorhersage der Wartungskosten.

5 Nutzwertanalyse der Softwarewartung

Die Kosten und Risiken der Wartungsaufgaben sind die eine Seite der RoI Kalkulation. Der Nutzen der Wartung steht auf der anderen Seite. Zu unterscheiden ist zwischen den Nutzen der vier Wartungsarten – der Nutzen der Fehlerbeseitigung, der Nutzung der Änderungen, und der Nutzen der Nachbesserungen. Jede Wartungsart hat seinen eigenen Nutzen. Es ist deshalb erforderlich sie differenziert zu betrachten.

5.1 Nutzen der korrektiven Wartung

Der Nutzwert einer Fehlerkorrektur ist das Inverse von dem was der Fehler später kostet. Die Kosten eines Fehlers sind zweierlei. Zum Einen gibt es die Kosten der Fehlerbeseitigung in der Produktion, die nach Boehm und Anderen das Vierfache mehr ist als die Kosten der Fehlerbeseitigung in der Wartung. Der Nutzen der früheren Korrektur ist also die Differenz zwischen den Kosten der jetzigen Korrektur in der Wartung und der späteren Korrektur in der Produktion. Zum Zweiten gibt es die Kosten der Produktionsausfälle. Jede Arbeitsstunde, die aufgrund eines Fehlers zusätzlich entsteht, sei es ein durch einen Systemausfall oder durch die Umgehung eines falschen Ergebnis verursacht einen Geldverlust. Dieser Geldverlust entspricht den Kosten der verlorenen Arbeitsstunden.

Die verlorene Arbeitszeit ist nur die eine Seite des Schadens der durch Softwarefehler entsteht. Hinzu kommt der Verlust an Vertrauen sowie an verlorenen Geschäftsgelegenheiten. Dieser Verlust kann in die Millionen gehen, je nachdem wie kritisch das System ist. Ein ernsthafter Fehler kann zur Produktionsunterbrechung, zum Kundenverlust oder gar zu einem Gerichtsprozess führen. Es ist schwer solche Kosten im Voraus zu entziffern. Boehm und Huang schlagen eine negative Pareto Verteilung für die Schätzung derartiger Verluste vor [BoHu06].

Die hohen potentiellen Kosten von Fehlern in der Produktion erklären warum korrektive Wartung immer den Vorrang vor den anderen Wartungsarten hat. Zunächst muss die Qualität des Produktes gesichert werden, dann folgt die Funktionalität.

5.2 Nutzen der adaptiven Wartung

Der Nutzen einer Änderung ist die Differenz in dem Wert des Produktes ohne die geänderte Eigenschaft und dem Wert mit der geänderten Eigenschaft. Jede Änderung müsste zu einer Wertsteigerung führen. Die Höhe der Wertsteigerung ist gleich der Nutzwert. Manche Softwaresysteme verlieren ihren ganzen Wert wenn sie nicht geändert werden, z.B. im Falle einer Gesetzesänderung oder eines Währungswechsels. In solchen Fällen ist der Wert der Änderung gleich dem Wert des Gesamtproduktes.

Andere Änderungen wie die Verschönerung der Benutzeroberfläche bringen nur marginale Wertsteigerungen. Es muss nachgewiesen werden, dass solche Änderungen zu einer Produktivitätssteigerung führen. Wenn vorher eine Kassiererin 20 Kunden pro Stunde abfertigt, muss sie nachher mindestens 21 Kunden pro Stunde abfertigen können. Demnach ist der Wert der Änderung 5% des Wertes der Kundenabfertigungstransaktion. Um ihre Kosten zu rechtfertigen musste jede Änderung entweder zu einer Produktivitätssteigerung oder zu einer Kostenersparnis führen. Ausnahmen sind Änderungen, die nicht quantifizierbaren Nutzen wie die Steigerung der Benutzerzufriedenheit bringen, aber auch solche Änderungen sollten mit einem fiktiven Nutzwert versehen werden. Die Höhe der Wertsteigerung ist der entscheidende Faktor bei der Rechtfertigung von Änderungsanträgen [Mock00].

Wertsteigerung	=	*Wert des geänderten Systems*
	-	*Wert des ursprünglichen Systems*

5.3 Nutzen der perfektiven Wartung

Kent Beck schreibt "the cost of a piece of code over its many-year life is dominated by how well it communicates to others… Every method name and every class name is an opportunity to communicate what is going on…" [Beck95]. Effektive Softwarewartung setzt voraus dass die Software verständlich und handhabbar ist. Perfektive Wartung zielt darauf, das Produkt in diesem Sinne nachzubessern. Die Namen sollten sprechender sein, die Kommentare ausführlicher, die Codestruktur transparenter, die Codebausteine mehr von einander isoliert sein. Kurzum, es sollte einfacher und weniger gefährlich sein den Code zu ändern, bzw. zu erweitern. Auch perfektive Wartungsmaßnahmen müssen einen Nutzwert haben. Messbar ist dieser Wert nur anhand der reduzierten Wartungskosten. Man musste vergleichen, was die Durchführung einer Änderung vor der Restrukturierung/Refaktorierung kostet und was sie nachher kostet. Der Wert der Perfektionsaktion ist die Differenz zwischen den Kosten vorher und den Kosten nachher. Man wird aber selten die Gelegenheit haben, einen derartigen empirischen Vergleich durchzuführen. Ergo ist man auf die Messung von Systemeigenschaften die den Wartungsaufwand treiben angewiesen. Diese Eigenschaften sind jene Komplexitäten und Qualitäten die mehr oder weniger Arbeitsaufwand verursachen.

Leider ist noch nie bewiesen, welche Eigenschaften das sind. Viele Forscher haben sich mit dieser Frage beschäftigt aber noch keiner konnte bisher eine eindeutige Korrelation zwischen Codeeigenschaft und Wartungsaufwand nachweisen. Das Ersparnis an Wartungsaufwand ist sogar mit relativ banalen Eigenschaften wie Goto freier Code nicht eindeutig demonstrierbar. Dass die Erhaltung objektorientierter Systeme weniger kostet als die von prozeduralen Systemen ist auch noch nie bewiesen worden [Eier07]. Ergo können wir nur vermuten welche die wahren Wartungskostentreiber sind.

Dieser Autor hat vorgeschlagen, die Größe, Komplexität und Qualität des Produktes vor und nach der perfektiven Wartung zu messen. Das Ziel einer Sanierung müsste sein, die Größe und die Komplexität der Software zu reduzieren und die Qualität zu steigern.

Die Wertsteigerung ergibt sich aus der Summe dieser Faktoren. Sollte aber die Größe und Komplexität steigen und die Qualität sinken folgt eine Wertminderung [Snee05]. Die Differenz wird wie folgt errechnet:

Added Value = *(1 - NewSize/OldSize))*
+ *(OldComplexity - NewComplexity)*
+ *(NewQuality - OldQuality)*

Im Falle eines Systems mit 100.000 Anweisungen, eine Komplexität von 0,70 und eine Qualität von 0,45 vor der Sanierung und 90.000 Anweisungen, eine Komplexität von 0,60 und eine Qualität von 0,55 nach der Sanierung wäre die Wertsteigerung:

(1 – (90.000 / 100.000)) +
(0,70 – 0,60) +
(0,55 – 0,45) = 0,30 = 30% Wertsteigung

Eine negative Wertsteigung würde folgen, wenn ein System bei gleichbleibender Funktionalität an Codemenge und Komplexität zunimmt. Steigt die Größe durch ein Refactoring von 100 auf 110 Anweisungen und die Komplexität von 0,70 auf 0,80, käme auch bei einer Steigerung der Qualität um 0,10 einen Wertverlust zustande.

(1 – (110.000 / 100.000)) +
(0,70 – 0,80) +
(0,55 – 0,45) = -0.10 = 10% Wertverlust

Dies ist das Gefahr bei Reengineering Projekten bei denen die Codemenge aufgebläht wird, um die Codequalität zu erhöhen. Der Gewinn an Qualität wird durch die vermehrte Quantität negiert, es sei denn man behauptet die Größe des Codes spiele keine Rolle. Wir wissen jedoch, je mehr Code zu warten ist, desto höher die Wartungskosten. Es muss ein Anliegen sein, bei gleichbleibender Funktionalität die Codemenge zu reduzieren, z.B. Durch die Entfernung von Klonen.

Ein ähnlicher Ansatz zur Rechtfertigung der Migration konventionell verteilter Komponente in Web Services wurde von Tilley, Huang und andere vorgeschlagen [TGH04].

6 Projektierte Kosten und Nutzen der Wartung

6.1 Feststellung des Nutzens

Der Nutzwert eines Wartungsprojekts entspricht dem Nutzwert des neuen Releases, das aus dem Projekt hervorgeht. Der Wert des Release ist der Wert des Produktanteils, welches von dem Projekt betroffen ist. Wenn das Gesamtprodukt einen Wert von € 1 Million hat und 5% der Anweisungen geändert oder hinzugefügt werden, hat dieses Release den Wert von € 50,000. Das entspricht dem proportionalen Anteil am Gesamtwert.

Wenn der Aufwand für das Release drei Personenmonate beträgt und der Mannmonat € 10.000 kostet, sind die Kosten des Releases 3 x 10,000 = € 30.000. Die RoI für dieses Wartungsprojekt ist dann:

$$(50.000 – 30.000) /30.000 = 0.66$$

Eine Möglichkeit den Wert eines IT-Systems zu fixieren ist der Kostenersparnisansatz. Demzufolge ist der Wert des Systems die Differenz zwischen den Kosten des betroffenen Geschäftsprozesses ohne das IT-System und den Kosten desselben Prozesses mit dem System [Levy87].

$$Value = (Operation\ without\ IT) – (Operation\ with\ IT)$$

Mit dieser Gleichung kommt man auch zu einem negativen Nutzen, nämlich dann wenn der Prozess mit IT mehr kostet als der Prozess ohne IT. Dies war z.B. der Fall beim Arbeitslosengeld-II System. Dies ist aber eine grobe Vereinfachung weil ein IT-System in der Regel Nutzen hat, die nicht in gesparten Kosten auszudrucken sind, z:B. bequemer arbeiten oder schneller abfertigen. Solche qualitative Nutzen sind jedoch schwer zu erfassen. Deshalb bleiben wir lieber beim Kostenersparnis.

Die Kosten sollen außerdem noch durch die Risiken ergänzt werden. Jeder Eingriff in ein bestehendes System birgt Risiken in sich. Die Zuverlässigkeit der Software könnte durch die Änderungen beeinträchtigt werden oder es kommt zum Verlust an Performanz. Es gilt diese Risiken zu identifizieren und zu gewichten – nach Wahrscheinlichkeit und nach Schadensausmaß. Es gilt ferner die potentieller Kosten dieser Risiken zu schätzen. Danach wird die Summe der potentiellen Risikokosten zu den Kosten des Projektes selbst hinzugefügt.

$$Value = Benefit – (Costs + Risks)$$

Diese Gleichung wurde angepasst und angewandt in einer Doktorarbeit von dem Betriebswirt Eckhart von Hahn mit dem Titel “Preservation of Software Value” [Hahn05]. Von Hahn befasst sich darin mit der Werterhaltung von Softwareprodukten in einem dynamischen Markt. Der Wert der Produkte sollte nicht nur erhalten sondern auch noch gesteigert werden. Reengineering ist ein Mittel dieses Ziel zu erreichen. Durch ein Reengineering Projekt sollte die Qualität eines Produktes steigen und die Komplexität sinken. Dadurch sollten die Wartungskosten zurückgehen. Ein Produkt mit reduzierten Wartungskosten steigt im Wert. Von Hahn weist darauf hin, dass der Wert eines Produktes nie gleich bleiben kann. Entweder er sinkt weil die Funktionalität hinter den Erwartungen hinterher hängt oder er sinkt weil die steigende Komplexität oder Größe zu höheren Wartungsaufwänden führt. Mit jedem neuen Release muss also die Funktionalität erweitert und die Komplexität relativ zur Größe reduziert werden. Wenn nicht immer Mehr in ein Produkt investiert wird, verliert es an Wert. Dies führte SAP dazu ihre Wartungsgebühren von 17 auf 22% vom Kaufpreis zu erhöhen, eine Erhöhung die von den Anwendern abgelehnt wurde. Diese Erhöhung war aber notwendig um die Kosten der Produktweiterentwicklung zu finanzieren. Leider ist es der SAP nicht gelungen dies den Kunden klar zu machen.

6.2 Schätzung der Kosten

Die Schätzung von Wartungskosten kann sowohl auf der Mikroebene als auch auf der Makroebene stattfinden. Auf der Mikroebene werden die Kosten der einzelnen Wartungsaufträge geschätzt. Wie dies erreicht wird, wurde im 3. Abschnitt geschildert. Auf der Makroebene werden die jährlichen Wartungskosten zum Beginn eines jeden Jahes projektiert. Hierfür hat Boehm in dem originalen COCOMO Model folgende Formel vorgeschlagen:

*AnnualMaintEffort = Systemtype * [(AnnualChangeRate * DevelopmentEffort) * QualityAdjustment]*

Der Systemtyp könnte Standalone, Integriert, Verteilt oder Embedded sein. Jeder Typ hat einen anderen Multiplikationsfaktor. Die jährliche Änderungsrate ist der Anteil geänderter und hinzugefügter Anweisungen relativ zu allen Anweisungen. Der Entwicklungsaufwand ist die Anzahl Personenmonate, die gebraucht worden um das System bis zur ersten produktiven Freigabe zu bringen. Der Qualitätsjustierungsfaktor ist ein Multiplikator der den Qualitätsstand der Software wiederspiegelt. Eine überdurchschnittliche Qualität mindert den Wartungsaufwand und ist deshalb < 1. Eine unterdurchschnittliche Qualität steigert den Wartungsaufwand und ist deshalb > 1. Der Qualitätsfaktor ist also genau inverse zur gemessenen Qualitätsnote. Boehm hat die Komplexität außeracht gelassen. Vielleicht meinte er sie wäre in dem Entwicklungsaufwand eingeschlossen. Dies ist aber wie wir inzwischen wissen eine Fehlannahme, denn Komplexität bleibt nicht konstant, sie steigt in dem Maße wie die Software sich von seiner ursprünglichen Form entfernt [SnHT04].

Deshalb hat dieser Autor die Boehm Formel um einen Komplexitätsjustierungsfaktor ergänzt. Die erweiterte Formel sieht so aus.

*AnnualMaintEffort = Systemtype * [(AnnualChangeRate * DevelopmentEffort) * QualityAdjustment * ComplexityAdjustment]*

Wie die Komplexitäts- und Qualitätsjustierungsfaktoren zustande kommen wurde schon beschrieben. Der Entwicklungsaufwand dürfte aus der Zeiterfassung zu holen sein. Die jährliche Änderungsrate wird einfach auf der Basis der bisherigen Änderungsraten hochgerechnet [Snee05].

7 Berechnung der RoI eines Wartungsprojektes

Als Beispiel für die Berechnung einer RoI für die Softwarewartung wird ein Testwerkzeug herangezogen. In einem Testprojekt für ein Webportal hatte der Autor die Aufgabe gehabt, ein Anforderungsdokument zu analysieren um daraus anforderungsbasierte Testfälle abzuleiten. Es ist in Testprojekten üblich dies manuell zu tun. Dafür waren 18 Personentage erforderlich. Bei einem Tagessatz von € 800,- kamen die Kosten für die manuelle Testfallermittlung auf € 14,400. Später erstelle der Autor einen Textanalysator, um diese Arbeit zu automatisieren. Mit dem Textanalysator konnte die gleiche Anzahl an Testfällen aus demselben Dokument innerhalb 3 Personentage gewonnen werden. Der Werkzeugeinsatz brachte damit ein Ersparnis von 15 Personentage = € 12.000 gegenüber dem manuellen Verfahren. Dies stellt den Nutzwert dieser Software für ein Testprojekt dar. Seitdem werden mit dem Werkzeug im Durchschnitt zwei Testprojekte pro Jahr unterstützt. Das macht einen jährlichen Nutzwert von € 24.000 aus.

Die Kosten zur Entwicklung des Werkzeuges betrugen 40 Personentage, bzw. € 32.000. Es dauerte also 1,5 Jahre bis die Entwicklungskosten sich amortisiert haben. Danach fielen rund € 8.000 pro Jahr – ein Viertel der Entwicklungskosten - für die Wartung und Weiterentwicklung an. Dies scheint etwas hoch zu sein, ist aber eine Folge der ständigen Anpassungen an neuen Anforderungstypen. Der Risikofaktor bei der Wartung dieses Produktes ist vernachlässigbar weil das Produkt keine kritische Anwendung ist, die dauernd im Betrieb sein muss. In einem Zeitabschnitt von einem Jahr rechnet sich die RoI der Erhaltung wie folgt:

$$RoI = [24.000 – 8000] / 8.000 = 2$$

Demnach erwirtschaftet der Textanalysator einen Gewinn von € 16.000 pro Jahr oder das Doppelte der Investitionshöhe. Würde die Nutzung auf vier Testprojekte pro Jahr bei gleichbleibender Wartungskosten, steigern wäre die RoI = 5 oder das Fünffache der Investition in der Wartung. In diesem Falle lohnt es sich sehr wohl das Produkt weiter zu warten. Würde aber die Nutzung auf ein Testprojekt pro Jahr zurückfallen und die Wartungskosten auf € 12.000 steigen, wäre der Gewinn = Null. Hier musste man sich die Frage stellen ob es lohne dieses Produkt weiter zu erhalten.

8 Zusammenfassung

Softwarewartung darf nicht allein als Kostenfaktor angesehen werden. Er stellt einen Wert dar, nämlich was es kosten würde die gleiche Aufgabe ohne die Software zu bearbeiten oder was es kosten würde eine neue Lösung zu schaffen. Der Nutzwert einer Neuentwicklung muss im Bezug zum Nutzwert der bestehenden Lösung angesehen werden. Wenn er den Nutzen der vorhandenen Lösung bei nicht mehr als 50% übertrifft, empfiehlt es sich bei der alten Lösung zu bleiben. Das liegt daran, dass die Risiken einer Neuentwicklung sich zwischen 10 und 50% der Entwicklungskosten bewegen. Man braucht einen Mehrwert von mindestens 50% um diese Risiken abzufangen. Dies ist der Hauptgrund warum die Entwicklung neuer Ablösesysteme immer wieder verschoben wird. Die Erhaltung der bestehenden Systeme hat also sehr wohl einen quantifizierbaren Nutzwert, nämlich die ersparten Kosten einer Neuentwicklung. Es sei erforderlich diesen zu ermitteln um die Kosten der Wartung zu rechtfertigen.

Viele IT-Manager meinen die Wartung der bestehenden Systeme kostet zu viel. Sie fragen, warum ist Softwarewartung so teuer? Die Antwort von Tom DeMarco ist „relative to what?“ [DeMa97] Wenn man die Wartungskosten mit dem Wert des gewarteten Gutes vergleicht, denn sind sie eher zu niedrig.

Literaturverzeichnis

[Baet98] Baetjer, H.: Software as Capital – An Economic Perspective on Software Engineering, IEEE Computer Society Press, Los Alamitos, 1998, p. 87

[Beck95] Beck, K.: Clean Code – Pipe Dream or State of Mind, Smalltalk Report Nr. 4, June, 1995, p. 20

[BeLe85] Belady, L./Lehman, M.: Laws of Software Evolution, in Software Evolution, Academic Press, London, 1985.

[Biff06] Biffl et a.: Value-based Software Engineering, Springer Pub., Berlin, 2006, p. 21

[Blom08] Blom, S. /Gruhn, V./Koehler, A./Schaefer, C.: Methoden und Grundlagen der wertebasierten Softwareentwicklung, Objektspektrum, Nr. 1, Feb 2008, p. 12

[BoHu06] Boehm, B./Huang, L.: How much Software Quality investment is enough – A value-based Approach, IEEE Software, Sept. 2006, p. 88

[BoTu05] Boehm, B./Turner, R.: Management Challenges to implementing Agile Processes, IEEE Software, Sept. 2005, p. 30

[Chap01] Chapin, N./Hale, J./Kahn, K./Ramil, J./Tan,W.: Types of Software Evolution and Maintenance, Journal of Software Maintenance and Evolution, Vol. 13, Nr. 1, Jan. 2001, p.3

[DeMa97] DeMarco, T.: Warum ist Software so teuer, Hanser Verlag, München/Wien, 1997

[Eier07] Eierman, M./Dishaw, M.: Comparison of object-oriented and third generation development languages, Journal of Software Maintenance and Evolution, Vol. 19, No. 1, Jan. 2007, p. 33

[Hahn05] von Hahn, E.: Werterhaltung von Software, German University Press, Wiesbaden, 2005, p. 19

[Hatt07] Hatton, L.: How accurately do Engineers predict Software Maintenance Tasks, IEEE Computer, Feb. 2007, p. 64

[Haye39] Hayek, F.: The Maintenance of Capital, in Profits, Interest and Investment, Routledge & Sons, London, 1939.

[Haye41] Hayek, F.: The Theory of Capital, University of Chicago Press, Chicago, 1941, p. 93
[IEEE93] IEEE: ANSI/IEEE Standard 1219-1993, Standard for Software Maintenance, IEEE Press, New York, 1993, p. 15
[Lach75] Lachmann, L.: Reflections on the Hayekian Capital Theory, in Proc. of Allied Social Science Association, Dallas, 1975, p. 132
[Levy87] Levy, L.: Taming the Tiger – Software Engineering and Software Economics, Springer Verlag, London, 1987, p. 111
[Mell05] Mellor, S.: Adapting Agile Approaches to your Project Needs, IEEE Software, May, 2005, p. 17
[Mock00] Mockus, A./Votta, L.: Identifying reasons for Software Change using Historic Databases, in Proc. of 16th ICSM, IEEE Press, San Jose, 2000, p. 120
[Schm22] Schmidt, F.: Organische Tageswertbilanz, Herder Verlag, Leipzig, 1922, s. 133
[SnHT04] Sneed,H./Hasitschka,M./Teichmann,M.T.: Software-Produktmanagement, dpunkt Verlag, Heidelberg, 2004, p. 18
[Snee91] Sneed, H.: Economics of Software Reengineering, Journal of Software Maintenance, Vol. 3, Nr. 1, Sept. 1991, p. 163
[Snee01] Sneed, H.: Impact Analysis of Maintenance Tasks, in Proc. of 17th ICSM, IEEE Press, Florence, 2001, p. 180
[Snee04] Sneed, H.: A Cost Model for Software Maintenance and Evolution, in Proc. of 20th ICSM, IEEE Press, Chicago, 2004, p. 264
[Snee05] Sneed, H.: Estimating the Costs of Reengineering Projects, in Proc. of 12th WCRE, IEEE Press, Pittsburgh, 2005, p. 111
[Snee05] Sneed, H.: Software-Projektkalkulation, Hanser Verlag, München, 2005, p. 117
[Soli04] Solingen, R.: "Measuring the ROI of Software Process Improvement", IEEE Software, May 2004, p. 32
[TGH04] Tilley,S./Gerdes,J./Hamilton,T./Huang,S./Mueller,H./Smith,D./Wong,K.: On the Business Value and technical Challenge of adopting web services, Journal of Software Maintenance and Evolution, Vol. 16, Nr. 1-2, Jan. 2004, p. 31
[Tock05] Tockey, S.: Return on Software – maximizing the Return on your Software Investment, Addison-Wesley, Boston, 2005, p 211