Aufwandsschätzung der Softwarewartung und –evolution

Harry M. Sneed

Technische Universität Dresden
harry.sneed@t-online.de

Abstract: In diesem Beitrag wird eine Methode für die Schätzung der Aufwände zur Erhaltung und Evolution bestehender Software-Systeme dargelegt. Zunächst wird kurz erläutert, was Softwareevolution beinhaltet und wie sie sich von Softwareentwicklung unterscheidet. Es wird betont, dass Software Evolution anders geplant und geschätzt werden muss. Dadurch wird ein werkzeuggestütztes Verfahren für die Messung der bestehenden Systeme vorgestellt. Im Anschluss daran wird eine Schätzmethode für den Entwicklungs- und Evolutionsaufwand vorgestellt. Die Methode ist eine Erweiterung der COCOMO-II Schätzmethode von Boehm. Zum Schluss werden zwei Schätzprojekte geschildert, in dem die Methode angewandt wurde – eine Schätzung für die Erhaltung und Evolution eins Legacy Mainframe Systems und eine für die Erhaltung und Evolution eines modernen DotNet Systems. In beiden Beispielen werden die gleichen Schätz-Algorithmen angewandt. Der Beitrag soll dazu dienen, Software Produktmanagement besser planbar zu machen.

1 Abgrenzung der Begriffe – Entwicklung, Wartung und Evolution

Die Wartung bestehender Softwaresysteme beinhaltet sowohl die Fortschreibung der Anforderungen als auch die Fortschreibung des Codes. Die Anforderungen sind deshalb fortzuschreiben, damit sie als Testbasis für den Code dienen können. Bei jeder Änderung sind die Testfälle für den Test der geänderten Codeabschnitte aus den geänderten Anforderungen abzuleiten. Wartung heißt also Anforderungen, Code und Test miteinander im Gleichklang zu halten. Dieses stellt eine besondere Herausforderung an das Produktmanagement dar und an diejenigen, die Wartungs- und Evolutionsprojekte schätzen müssen. Diese Aktivitäten müssen sorgfältig auseinander gehalten werden. Neben Entwicklungsaktivitäten gibt es auch Integrations- Wartungs- Evolutions-, Migrations- und Testaktivitäten. Schätzer müssen wissen, welche Art von Aktivität, bzw. welche Projektart, sie zu schätzen haben, denn jede hat andere Maße, Einflüsse, Qualitäten und Produktivitätsdaten. Es kann nicht nur eine Schätzmethode geben. Es kommt auf die Projektart an.

1.1 Zur Entstehung des Begriffs „Maintenance“

Am Anfang gab es den Begriff *„Software Maintenance“* für alle Arbeiten an einem Softwareprodukt nach dessen erstem produktiven Einsatz [MaOs83]. Die Unterscheidung von Wartung und Entwicklung war aus Budgetgründen notwendig geworden, denn allzu oft wurden Entwicklungsprojekte einfach nach Ablauf der vereinbarten Zeit

und/oder des vereinbarten Budgets als beendet erklärt. Aus den verbleibenden Entwicklungsaufgaben sind Wartungsaufträge geworden. Die US-Bundesverwaltung wollte vermeiden, dass einmalige Projektkosten als laufende Operationskosten getarnt werden. Daher die scharfe Trennung zwischen Entstehung- und Erhaltungskosten, gebunden an einem Stichtag. Es gab auch bis in die 90er Jahren hinein nur der Begriff „Maintenance" für all das, was geleistet wird, nachdem ein Softwaresystem produktiv eingesetzt ist. Dazu gehört u.a. die Fehlerkorrektur, die Anpassung, die Optimierung, die laufende Sanierung und die begrenzte Erweiterungen. Große Sanierungsvorhaben sowie größere Erweiterungen und Migrationen zählten nicht dazu, weil diese Projekte mit eigenem Budget sein sollten. Diese Einteilung von Softwarewartungsarbeiten ist in dem klassischen Buch über Wartungsmanagement von Lienz und Swanson aus dem Jahr 1980 beschrieben [LiSw80].

1.2 Evolution kommt hinzu

In den 90er Jahren kam der Begriff *„Evolution"* auf. Die Fachwelt musste zur Kenntnis nehmen, dass komplexe Softwareprodukte nie vollendet sind. So lange sie benutzt werden, werden sie weiterentwickelt. Insofern passt der Begriff „Projekt" nicht so recht zu der Entwicklung eines Softwareproduktes in seiner Gesamtheit, denn ein Projekt ist zeitlich und kostenmäßig begrenzt. Er passt besser zu der Entwicklung eines Teilproduktes, bzw. zu einer Iteration im Software-Lebenszyklus. Der Tag des ersten produktiven Einsatzes verlor seine Bedeutung, denn es folgen noch viele solche Tage, wenn das Produkt immer weiter ergänzt wird. *Software Evolution* bezeichnet die ständige Weiterentwicklung eines bestehenden Softwareproduktes. Der Begriff „Software Maintenance", zu Deutsch „Softwareerhaltung" wurde auf die Aktivitäten *Fehlerkorrektur* und unumgängliche technische *Anpassungen* beschränkt. Die Zeitschrift „Journal of Software Maintenance" heißt seit dem Jahr 2000 „Journal of Software Maintenance and Evolution" und die internationale IEEE Konferenz „ICSM" heißt inzwischen „ICSME". Damit wird die Unterscheidung der beiden Begriffe betont [BeRa00].

Dennoch ist auch diese Begriffsunterscheidung fraglich. Sie setzt nämlich voraus, dass es einen Stichtag gibt, an dem die Entwicklung zu Ende ist und die Erhaltung und Evolution beginnt. Infolge neuer agiler Entwicklungsansätze wie SCRUM und KANBAN ist es nicht mehr eindeutig, wann das letzte Release eines Produktes erfolgt [Cohn05]. In agilen Projekten werden neue Definitionen von Wartung und Evolution benötigt, denn beides findet während der Entwicklung statt. Es muss aber einen Zeitpunkt geben, an dem das Produkt nicht mehr unter Entwicklung ist. Dieser Zeitpunkt könnte sein, wenn das Produkt – sprich der Source Code – sich in einem Release nicht mehr als *n* % ändert. Wenn sich in zwei aufeinander folgenden Releases der Source Code weniger als 5% ändert, dann hat sich das Produkt wohl stabilisiert. Wenn es auch noch produktiv eingesetzt wird, kann das Produkt als „done" bezeichnet werden. Ab diesem Punkt gelten die Gesetze der Software-Evolution. Die Software wird zwar noch fortgeschrieben und verbessert, aber nur noch in beschränktem Maße. Ab diesem Zeitpunkt gelten andere Regeln für die Budgetierung der Kosten [NiVl00].

1.3 Wartung = Korrektur und Anpassung

Softwarewartungsarbeiten werden durch Fehlermeldungen und Änderungsanträge ausgelöst. Bei einer Fehlermeldung soll die Software in einen Zustand versetzt werden, die sie vom Anfang an hätte sein sollen. Sie wurde an irgendeiner Stelle falsch implementiert und diese Stelle muss ausgebessert werden. Eine solche Fehlerkorrektur dürfte in der Regel nicht mehr als zwei Tage dauern – einen Tag, um das Problem zu lokalisieren und auszubessern und einen Tag um die ausgebesserte Version zu bestätigen. Bei einem Änderungsauftrag soll das Verhalten der Software geringfügig geändert werden, z. B. ein Text auf der Benutzeroberfläche wird ausgetauscht oder eine lokale Berechnungsformel wird umgestellt. Derartige Änderungen dürfen nicht länger als eine Fehlerkorrektur dauern. Deshalb werden sie in die gleiche Kategorie geworfen. Wir können daraus den Schluss ziehen, dass Wartungsarbeiten zeitlich und aufwandsmäßig begrenzt sind. Sie dürfen nicht länger als n Tage dauern und nicht mehr als m Personentage in Anspruch nehmen. Was die Größen n und m wirklich sind, hängt von der Größe des jeweiligen Produktes ab [Sned04].

1.4 Evolution = Erweiterung und Verbesserung

Software-Evolutionsarbeiten werden durch Weiterentwicklungsanträge oder Verbesserungsvorschläge ausgelöst. Es handelt sich hier um eine Wertsteigerung. Das Softwareprodukt soll entweder etwas leisten, was es bisher noch nicht geleistet hat oder es soll es besser leisten als bisher. Im ersten Fall wird die Funktionalität erweitert, im zweiten Fall die Qualität verbessert. In beiden Fällen steigt der Wert des Produktes [BGKS08]. Es kann mehr leisten oder es wird schneller, benutzerfreundlicher oder leichter zu pflegen. Eine derartige funktionale Erweiterung, bzw. qualitative Verbesserung kann mehrere Tage oder gar Wochen dauern und mehrere Personen beanspruchen. Es darf aber nicht mehr als ein Drittel des Codes betroffen sein, sonst handelt es sich um eine Neuentwicklung. Typische Evolutionsarbeiten sind die Sanierung bestimmter Codekomponenten, die Optimierung bestimmter Abläufe und der Einbau zusätzlicher Funktionen.

Wartung und Evolution sollten von einander getrennt abgerechnet werden, auch wenn sie gleichzeitig von den selben Personen durchgeführt werden. Sie dienen zweierlei Zwecken – der eine Zweck ist die Erhaltung des Status Quo, der andere ist die Erweiterung bzw. Verbesserung des gegenwärtigen Zustandes [CHKR01].

2 Softwaremessung als Voraussetzung der Evolutionsplanung

Aufwandsschätzung ist eine wichtige Funktion im Rahmen der Produktplanung. Ohne sie gibt es keine verbindliche Leitlinie. Das Produktmanagement soll wissen, was ein neues Release kostet und wann es fertig wird, um zu entscheiden ob es überhaupt wünschenswert ist. Außerdem braucht das Produktmanagement Soll-Vorgaben für Zeit und Geld, damit es das Ist mit dem Soll vergleichen kann. Damit sind wir bei den Zahlen gelandet. Ohne Zahlen ist keine vernünftige Planung und Steuerung der Wartungs- und Evolutionsarbeiten möglich.

Um Zahlen über die Zeit und Kosten der Softwarewartung und -evolution zu gewinnen, braucht das Produktmanagement zunächst Zahlen über das Produkt selbst. Die Messung des bestehenden Produktes ist daher eine Vorbedingung für die Produktplanung [Sn-HT04]. Der Umfang der anstehenden Wartungs- und Evolutionsaufgaben kann nur ermittelt werden, wenn die Größe, Komplexität und Qualität des Produktes bekannt sind. Dies ist der große Unterschied zur Schätzung von Entwicklungskosten, bei denen es noch kein real existierendes Produkt gibt. Die Schätzer müssen sich dort entweder auf abstrakte Modelle oder auf ungenaue Vergleiche stützen. Als Modell für die Schätzung gilt das sog. "Teufelsquadrat" mit den Ecken Größe, Qualität, Zeit und Kosten (Abb. 1).

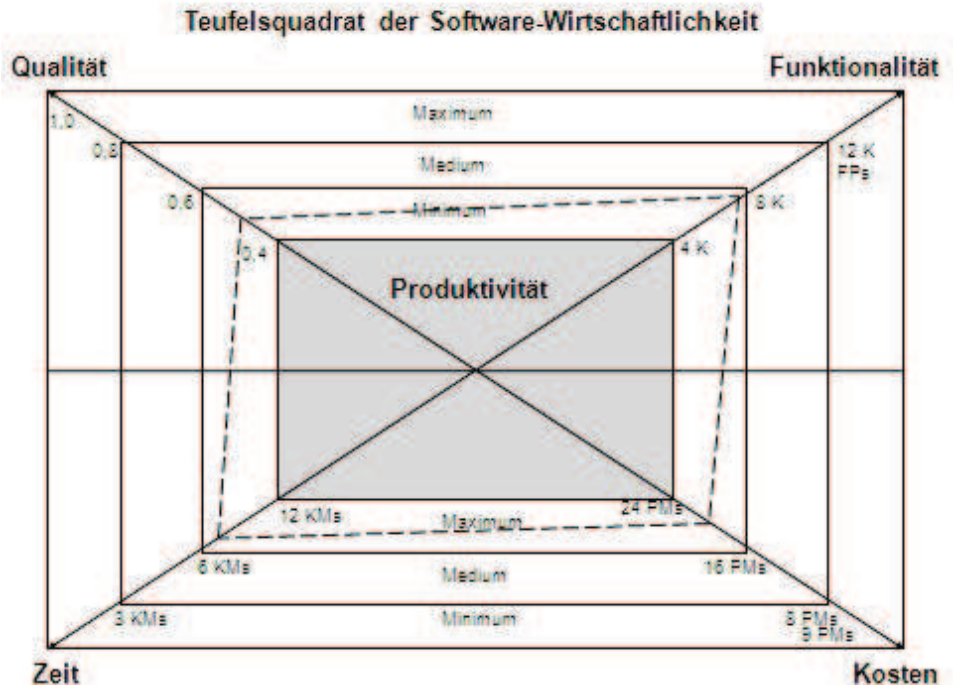

Abbildung 1: Zusammenhang von Qualität, Funktionalität, Zeit und Kosten bei Software

2.1 Messung der Produktgröße

Die Größe bestehender Softwaresysteme wird am leichtesten in Anweisungen, Data-Points oder Object-Points gemessen. Nur Lines of Code zu messen, ist unzulänglich, weil Codezeilen leicht manipulierbar sind. Sie sind auch abhängig von der Programmiersprache und dem Editor, der benutzt wird. Function-Points sind nicht ohne weiteres brauchbar, weil sie keine unmittelbare Beziehung zum Code haben. Je nachdem, wie gut der Code implementiert ist, wird es mehr oder weniger Codeanweisungen pro Function-

Point geben. Dies hat die Erfahrung mit bisherigen Schätzungen belegt [Sned05]. Was aber gewartet werden muss, ist der Code, unabhängig davon, wie viele Function-Points damit implementiert sind. Von Capers Jones wird die „Backfiring Technik“ als Methode zur Ermittlung der Function-Points anhand der Codezeilen vorgeschlagen, aber diese Technik basiert auf statistischen Durchschnittswerten mit einer sehr breiten Streuung [Jone98]. Folglich können sie für einzelne Systeme weit daneben liegen. Die Zählung von Function-Points anhand des Codes setzt die Erkennung der Ein- und Ausgabeoperationen voraus und dies kann sehr schwierig sein.

Use-Case-Points scheiden aus, weil es im Code – vor allem im objektorientierten Code – nicht möglich ist, die Anwendungsfälle zu erkennen. Der Code ist nach Objekten strukturiert und nicht nach Funktionen. Die Ausführungsfolge der Objekte wäre nur durch den Test des Codes herauszufinden, ansonsten befinden sie sich nur in der Produktdokumentation [KiFK04]. Diese ist aber in der betrieblichen Praxis nur selten vorhanden und wenn ja, dann ist sie meistens veraltet. Außerdem gilt für Use-Case-Points das Gleiche wie für Function-Points: Es besteht kein festes Verhältnis zur Codegröße. Wenn Produktgröße gemessen wird, sollen deshalb Anweisungen und Data-Points oder Object-Points gezählt werden.

2.2 Messung der Produktkomplexität

Der Code-Komplexitätsmaß ist der gewichtete Mittelwert mehrerer Einzelkomplexitäten – darunter Datenkomplexität, Datenflusskomplexität, Zugriffskomplexität, Interaktionskomplexität, Steuerflusskomplexität, Vererbungskomplexität und Sprachkomplexität. Die Formel für die Berechnung dieser Komplexitäten ist in früheren Papers sowie in dem Buch „Software in Zahlen“ erläutert [Sned11]. Wichtig hier ist, dass aus den vielen Komplexitätsmessungen am Ende ein einziges rationales Maß auf der Skala von 0 bis 1 herauskommt, wobei 0 die absolute Abwesenheit von Komplexität bedeutet. Je niedriger der Komplexitätsmaß ist, desto besser ist es für die Systemerhaltung. .

2.3 Messung der Produktqualität

Ein Code-Qualitätsmaß kann der gewichtete Mittelwert einzelner Qualitätsmesswerte sein. Dazu gehören *Modularität* = Modulgrößen * Kopplung * Kohäsion; *Flexibilität* = Abwesenheit eingebauter Konstanten und Texte; *Wiederverwendbarkeit* = geringe Abhängigkeit von anderem Code; *Portabilität* = geringe Abhängigkeit von der Umgebung; *Testbarkeit* = schmale Schnittstellen und möglichst wenig Pfade zu testen sowie Hooks für Testhilfen, flache Codestrukturen; *Sicherheit* = eingebaute Kontrollen der Vor- und Nachzustände, *Konformität* mit den Codierregeln und adäquate Kommentierung. Auch diese Qualitäts-Koeffizienten sind in früheren Veröffentlichungen sowie in dem Buch „Software in Zahlen“ zu finden [SnMe85]. Wie bei der Komplexität soll hier ein einziges rationales Maß auf der Skala von 0 bis 1 herauskommen. Je höher die Qualität, desto besser für die Softwareerhaltung und -evolution. Der Code soll nachhaltig sein.

2.4 Durchführung der Messung

Das Objekt der Messung ist der Source Code. Ein Source Analyse Tool parst den Source Text und prüft die einzelnen Anweisungen. Es wird festgestellt, um welchen Typ Anweisung es sich handelt und welche Variablen wie behandelt werden. Die wichtigsten Anweisungstypen sind:

- Strukturanweisungen
- Bedingungsanweisungen
- Zuweisungsanweisungen
- Aufrufanweisungen und
- Daten-Ein-Ausgabeanweisungen.

Die Anzahl der Anweisungen ergibt die Codegröße. Die Anzahl der Anweisungstypen fließt in die Komplexitätsmessung hinein. Neben den Anweisungen werden auch die Variablen, Konstanten, Datentypen, Klassen, Prozedurblöcke und Methoden gezählt. Daraus ergibt sich das Codeprofil.

Data-Points werden anhand der Datendeklarationen, Datengruppen und Datenstrukturstufen gezählt. Object-Points werden anhand der Attribute bzw. Datendeklarationen, der Datengruppen, der Klassen, der Parameter und der Methoden gezählt. Function-Points sind zwar kein zuverlässiges Maß für die Wartung, werden aber trotzdem anhand der Datenbankzugriffe und der IO-Operationen sowie durch die Datei-, Datenbank- und Maskendefinitionen gezählt. Möglicherweise will das Produktmanagement die aus dem Code abgeleitete Function-Point Zahl mit der ursprünglich geschätzten Zahl vergleichen. Als Ergebnis der Codemessung stehen fünf Größenmaße zur Verfügung:

1. Codezeilen,
2. Anweisungen,
3. Data-Points,
4. Object-Points und
5. Function-Points [Sned95].

Bei größeren Software Systemen wird diese Messung auf verschiedenen Hierarchiestufen durchgeführt. Auf der untersten Ebene wird jedes Source-Member gemessen Hier findet auch die Prüfung der Codierregeln statt, die für die Konformitätsmessung erforderlich ist. Auf der obersten Ebene wird das Produkt in seiner Gesamtheit gemessen. Dazwischen werden die Komponenten, die Teilsysteme und die Systeme gemessen. Ein Produkt kann aus mehreren Systemen bestehen. Auf jeder Ebene werden die Grundzahlen aggregiert und die Komplexitäts- und Qualitätsmaße neu berechnet. Dadurch ist es möglich, sowohl den Aufwand für das ganze Produkt als auch der Aufwand für jede einzelne Komponente und jedes System unabhängig von den anderen zu kalkulieren.

3 Schätzung der Erhaltungskosten

Da Softwareerhaltung nicht ohne weiteres planbar ist, muss sie anders als die Softwareevolution geschätzt werden. Erhaltungsmaßnahmen wie Fehlerkorrektur und geringfügige Änderungen erfolgen je nach Bedarf unregelmäßig. So wie Unfälle und Verbrechen nur nachträglich statistisch erfasst werden können, können Softwarefehler und Notänderungen nur empirisch bzw. aufgrund bisheriger Erfahrungen hochgerechnet werden.

3.1 Projektion der Wartungsaufträge

Um zu erläutern, wie die Anzahl Wartungsaufträge hochgerechnet wird, wird hier eine konstruierte Fallstudie benutzt. In der letzten Planungsperiode sind 12 Fehler und 2 geringfügige Änderungen pro "Kilo" Anweisungen vorgekommen. Es ist zu erwarten, dass diese Zahlen sich mit abnehmender Tendenz in der nächsten Planungsperiode wiederholen werden. Der Abnahmegrad hängt von der steigenden Reife der Software ab. Bei gleichbleibender Größe, Komplexität und Qualität sinkt die Zahl der erforderlichen Wartungseingriffe erfahrungsgemäß um circa 25% pro Release [LaBr06]. Die Zahl der zu erwartenden Eingriffe wäre demnach

$$n^{0,8} \text{ wobei } n = \text{Anzahl Eingriffe in der letzten Release-Periode.}$$

Natürlich wird ein Softwareprodukt nie ganz fehlerfrei. Aber nach fünf Releases, wenn alles sonst unverändert bleibt, wird die Zahl der zu erwartenden Eingriffe von 4 auf 1,5 pro Kilo Anweisungen zurückgehen.

3.2 Schätzung des Aufwandes pro Wartungsauftrag

Dass alles unverändert bleibt, ist jedoch eher unwahrscheinlich. Durch die Eingriffe in den Code wächst die Komplexität und sinkt die Qualität. Dies geht aus den Gesetzen der Software Evolution von Lehman und Belady hervor [BeLe85]. Ergo muss die letzte Komplexität und Qualität in die Rechnung einbezogen werden. Nehmen wir ein System mit 200 Kilo Anweisungen, eine Komplexität von 0,6 und eine Qualität von 0,55 an. In der letzten Planungsperiode hat es 2 Fehlerkorrekturen und 1 geringfügige Änderung pro Kilo Anweisungen gegeben. Das ergibt für die 200 Kilo Anweisungen insgesamt 600 Eingriffe. Gegen die 600 Eingriffe wurden 600 Personentage gebucht bzw. im Durchschnitt 1,0 PT pro Wartungseingriff.

Beim neuen Release ist das System um 5% von 200 auf 210 Anweisungen gewachsen. Gleichzeitig ist die Komplexität von 0,6 auf 0,62 gestiegen und die Qualität von 0,55 auf 0,52 gesunken. Die neue justierte Größe ist demnach:

Größe * (Komplexität / Qualität) = 210 * (0,62 / 0,52) = **250 Kilo Anweisungen**.

Die alte justierte Größe war

200 * (0,6 / 0,55) = **218 Kilo Anweisungen**.

Der **unjustierte Größenunterschied** zwischen den beiden Releases beträgt:

210 / 200 = **5%**.

Der **justierte Größenunterschied** beträgt aber aufgrund steigender Komplexität und sinkender Qualität:

250 / 218 = **14,6 %**.

Ausgehend von einer zunehmenden Reife müsste die Zahl der Eingriffe von 3 auf 2,4 pro Kilo Anweisungen zurückgehen, oder von 600 auf 504 (210 * 2,4).

$3^{08} = 2{,}4$ Eingriffe.

Demnach hätten wir einen Aufwand von 504 * 1 = 504 Personentagen für die Erhaltung der nächsten Release-Periode. Aber das System ist nicht nur größer sondern auch komplexer geworden und gleichzeitig ist die Qualität gesunken. Der justierte Größenunterschied beträgt 14,6%. Also ist der justierte Aufwand 504 * 1,146 = 578 Personentage für die Erhaltung des neuen Releases. Am Ende sind die Erhaltungskosten trotz weniger Eingriffe nur 4% weniger als im letzten Release. Dass die absinkende Anzahl Wartungseingriffe durch den zunehmenden Aufwand pro Wartungseingriff ausgeglichen wird, ist eine Tendenz, die in der Wartungspraxis beobachtet wird [K-MA00].

Diese Zahlen rechtfertigen den Aufwand für die Software-Sanierung [Sned91]. Wäre die Komplexität der Software von 0,6 auf 0,5 gesunken und die Qualität von 0,55 auf 0,6 gestiegen, hätten wir einen justierten Größenunterschied von

{ 210 * (0,5 / 0,6) } / 218 = 0,8.

In diesem Fall wäre der justierte Aufwand 504 * 0,8 = 403 Personentage, d. h. die Sanierung würde eine Ersparnis von 175 Personentagen in nur einer Release-Periode bringen.

4 Schätzung der Evolutionskosten

Im Gegensatz zur Erhaltungsarbeit ist die Evolutionsarbeit sehr wohl planbar. Das Produktmanagement hat auch die Wahl, sie durchzuführen oder nicht. Falls sie zu teuer erscheint, kann sie verschoben oder ganz abgelehnt werden. Umso wichtiger ist es, sie richtig zu schätzen.

4.1 Ermittlung des Impakt-Bereiches

Voraussetzung für die Evolutionsschätzung ist eine Impakt-Analyse [Sned01]. Der Schätzer muss wissen, um welches Codegrößenausmaß es sich handelt, bzw. welcher Anteil der Software dadurch betroffen ist. Der Ausgangspunkt für die Impaktanalyse ist das „Change Request“, das eigentlich ein „Enhancement Request“ ist. Darin wird beschrieben, welche zusätzliche Funktionalität das Produkt haben sollte. Die Herausforde-

rung besteht darin, das „Change Request“ der richtigen Stelle in der Systemarchitektur zuzuweisen. Manchmal wird es eindeutig sein, z. B. wenn die Erweiterung sich auf eine neue Benutzerschnittstelle oder einen neuen Bericht bezieht. Ein anderes Mal ist es nicht so leicht erkennbar, wo die zusätzliche Funktionalität am besten einzubauen ist. Sofern der Änderungstext sich auf bestehende Datenobjekte in der Anforderungsdokumentation oder in dem Entwurfsmodell bezieht, können diese Objekte lokalisiert und die neuen Funktionen an die bestehenden Objekten angehängt werden. Verweist jedoch der Änderungstext auf neue Objekte, so muss der Systemarchitekt entscheiden, wo die neuen Codebausteine am besten zu platzieren sind. In beiden Fällen wird der neue Code eine Auswirkung auf den bestehenden Code haben. Durch die Impakt-Analyse wird ersichtlich, welche der bestehenden Codekomponenten durch die Erweiterung betroffen sind. Der Schätzer hat also nicht nur zu schätzen, wie viel neuer Code hinzukommt, sondern auch wie viel alter Code eventuell modifiziert werden muss.

4.2 Messung des Impakt-Bereiches

Die Anzahl der Anweisungen in den betroffenen Codebausteinen lässt sich aus der letzten Codemessung entnehmen. Sie werden anhand des Betroffenheitsgrades anteilmäßig gezählt. Sie könnte zwischen 5 und 50 % liegen [Sned95]. Die Zahl der Anweisungen in den neu zu erstellenden Modulen müssen auf herkömmliche Weise geschätzt werden. Dies geschieht durch einen Vergleich mit bestehenden Moduln, die in etwa den gleichen Funktionsumfang haben. Der Impact-Bereich erfasst die neuen + die betroffenen alten Bausteine. Die Größe des Impact-Bereiches ist die Summe der geschätzten neuen Anweisungen und des Anteils der betroffenen alten Anweisungen.

Die resultierende Anweisungszahl wird durch die Komplexität und die Qualität des Systems in seiner Gesamtheit justiert. Wenn z. B. zu dem System mit 200 Kilo Anweisungen 2000 neue Anweisungen hinzukommen und 4.000 alte Anweisungen zu 25% betroffen sind, beträgt der Impact-Bereich 3.000 Anweisungen. Der letzte gemessene Komplexitätsgrad ist 0,62 und der letzte gemessene Qualitätsgrad ist 0,52. Daraus folgt eine justierte Größe von

3.000 * (0,62 / 0,52) = 3.577 Anweisungen.

Dies wird durch die bisherige Produktivität in Anweisungen pro Tag dividiert, um die Personentage zu errechnen. Eine mittlere Produktivität ist z. B. circa 40 getestete Anweisungen pro Personentag. Diese Produktivitätsziffer geht aus einer empirischen Untersuchung der Softwarewartung bei englischen Anwendern hervor [Hatt07]. Demnach werden für 3.577 justierte Anweisungen 89 Personentage benötigt. Natürlich muss die Produktivität stets aufgrund der eigenen Erfahrungen kalibriert werden, aber im Laufe des Lebenszyklus' eines Software Produktes dürfte sie relativ stabil bleiben. Wenn drei solche Erweiterungen durchgeführt werden, kommt der Evolutionsaufwand auf 3 * 89 = 267 Personentage.

4.3 Schätzung der Gesamtkosten für eine Release-Periode

Die Kosten für einen neuen Lebenszyklusabschnitt setzen sich zusammen aus den Erhaltungskosten für das letzte Release + die Evolutionskosten für das neue Release + die Managementkosten. Die Managementkosten sind bei der Softwareerhaltung und –evolution besonders hoch. Sie beinhalten nicht nur das Produktmanagement, sondern auch das Konfigurationsmanagement, das Qualitätsmanagement und das Releasemanagement. Demzufolge ist mit einer Overheadrate von mindestens 33 % zu rechnen, d. h. die Managementskosten betragen ein Drittel der Gesamtkosten [Rama00]. Wenn für die Erhaltung des letzten Releases 578 Personentage und für die Entwicklung des neuen Releases 267 Personentage geschätzt sind, werden die Managementkosten

(578 + 267) / 2 = 422 Tage betragen.

Das ist ein Drittel der Gesamtkosten von 1.267 Personentagen. Wesentlich hier ist, dass alle Kostenarten berücksichtigt werden und die Erhaltung des letzten Releases nicht vernachlässigt wird.

5 Zwei Fallstudien aus der betrieblichen Softwarewartungspraxis

Um zu zeigen, wie die Schätzmethode angewandt wird, werden hier zwei Schätzprojekte aus der Wartungspraxis zitiert. Das eine Projekt betrifft die Erhaltung eines DotNet Systems für die Steuerung einer industriellen Fertigung von Automobilteilen in verschiedene Werken, das andere Projekt zielt auf die Erhaltung eines Personalabrechnungssystems auf dem IBM Mainframe in einer Landesverwaltung.

5.1 Die Erhaltung eines DotNet Systems für die Produktionssteuerung

Das DotNet System dient zur Steuerung der Automobilteile Produktion im diversen Werke verteilt in Europa. Das System hat eine Master-Version, die an einer zentralen Stelle gewartet wird. Diese Version ist aber nirgendwo im Einsatz, sie dient lediglich als Muster für die lokalen Versionen. Jedes Werk hat eine eigene lokale Version der Software, die mehr oder weniger von der Grundversion abweicht. Bei manchen Werken ist sie fast identisch zur Grundversion, in anderen hat sie fast das Zweifache an Größe und Funktionalität. Trotz der Unterschiede in die Funktionalität wurde angestrebt, die Software aus Kostengründen an einer zentralen Stelle für alle Werke zu pflegen. Das Ziel des Schätzprojektes war es, diese Entscheidung zu rechtfertigen [SnEr12].

Wie bei allen Schätzprojekten dieser Art war auch dieses zeitlich und aufwandsmäßig begrenzt. Die Schätzung durfte nicht länger als 10 Tage dauern und nicht mehr als 15 Personentage kosten. Sie begann mit der Sammlung der Source-Code Komponente aus vier stellvertretenden Werken plus die Master-Version. Die Source-Bibliotheken bestanden aus C# Klassen, XSAML Oberflächendefinitionen, XRES resource files und XSD bzw. WSDL Schnittstellendefinitionen. Es gab insgesamt

20.388 Source-Dateien mit 2,8 Millionen Codezeilen.

Allerdings gab es die meisten Source-Dateien mehrfach, da sie bei jedem Werk die gleichen waren. Es gab aber auch unterschiedliche Varianten der Sourcen für jedes Werk und auch ganz andere Sourcen. Das größte Werk hatte 5.418 Sourcen, das kleinste nur 3.405.

Es war deshalb erforderlich, die Software-Bibliotheken getrennt zu halten und jedes Werk für sich zu messen. Ziel war, die Größe, Komplexität und Qualität der Software für jedes Werk zu messen und daraus den jährlichen Erhaltungsaufwand abzuleiten. Die Summe aller Werksaufwände ergibt dann der Gesamtaufwand für die dezentrale Lösung. Der Gesamtaufwand für die zentrale Lösung wurde errechnet als die Summe der Aufwände für jedes Werk minus dem Aufwand für die Erhaltung der Master-Version, die dann nur einmal dazu addiert wurde. Die Werksysteme werden hier als System A, B, C und D bezeichnet, die Master-Version als System M.

	Stmnts	FP	Komplexität	Qualität	Justierte Größe (Stmts)
System-A	227.903	9690	0,412	0,645	145.575
System-B	243.205	10289	0,411	0,648	154.255
System-C	329.715	12795	0,423	0,654	213.250
System-D	382.554	14059	0,425	0,661	245.969
System-M	215.327	9086	0,406	0,648	134.912

Ausgegangen wird von drei Wartungseingriffen (Fehlerkorrekturen und Änderungen) pro Kilo Anweisungen pro Jahr. Für den Wartungseingriff wird durchschnittlich 1,5 PT berechnet, da auch der Test dazukommt. Dies basiert auf der Erfahrung in den ersten zwei Jahren. Daraus ergeben sich folgende Aufwände:

	Justierte Größe (Stmts)	Eingriffe	Aufwand (PT)
System-A	145.575	436	654
System-B	154.255	462	693
System-C	213.250	639	958
System-D	245.969	738	1107
System-M	134.912	270	405

Wenn jedes der vier Werke allein für sich gepflegt wird, werden 3.412 Personentage pro Jahr benötigt, bzw. 17 Personen in den 4 Werken.

Wenn die Software zentral gepflegt wird, werden nur 2197 Personentage pro Jahr benötigt:

$$\{\Sigma\,(\text{Werksaufwände}) - \text{Masteraufwand}\} + \text{Masteraufwand}.$$

Das ergibt 11 Wartungsingenieure für die vier Werke, ein Ersparnis von 6 Personenjahren. Je mehr Werke dazukommen und je größer der Anteil der Master-Version am gesamten Source-Code wird, desto höher die Ersparnis. Dies spricht für einen möglichst großen, wiederverwendbaren Codeanteil.

Der Unterschied wird noch krasser, wenn wir die Managementkosten berücksichtigen. Falls jedes Werk die Wartung seines Systems selbst betreibt, muss es auch die Overhead Kosten tragen. Diese wären für

System-A	654 / 2 = 327 PT
System-B	693 / 2 = 346 PT
System-C	958 / 2 = 479 PT
System-D	1.107 / 2 = 554 PT
Insgesamt	3.412 / 2 = 1.706 Personentage für Management-Overhead.

Zusammen mit den Overhead-Kosten kommt ein Aufwand von 4.512 Personentagen, bzw. 226 Personenmonaten zu Stande. Daraus leitet sich eine Kopfzahl von 25 Personen ab.

3.412 PTs + 1.706 PTs = 5.118 PTs pro Jahr bzw. 25 Personen.

Die zentrale Lösung würde nur 1.300 Personentage Overheadkosten verursachen.

System-A	(654 – 405) / 2 = 125 PT
System-B	(693 – 405) / 2 = 144 PT
System-C	(958 – 405) / 2 = 277 PT
System-D	(1107 – 405) / 2 = 551 PT
System-M	405 / 2 = 203 PT,
Insgesamt: 300 Personentage für Management-Overhead.	

Dazu kämen 2.197 PT für die Wartungseingriffe. Gesamtkosten für die zentrale Lösung wären also 2.197 + 1.300 = 3.497 Personentage. Das sind nur 18 Personen, bzw. 2/3 der Kosten der verteilten Erhaltung. So gesehen ist die zentrale Wartung der verteilten Werkssysteme eindeutig die wirtschaftlichere Lösung.

5.2 Die Erhaltung eines Legacy Mainframe Systems für die Personalabrechnung

Das Legacy Mainframe System, das hier als Fallstudie dient, ist ein Personalabrechnungssystem für ein österreichisches Bundesland. Das System wurde bereits in den 80er Jahren entwickelt und zwar mit der damals führenden Software-Technologie der Fa. IBM, PL/I, für die Batchprozesse im Backend und Visual Age für die Dialogprozesse im Frontend. Frontend und Backend sind über eine gemeinsame relationale Datenbank – DB2 verbunden. Die Architektur ist typisch für die damalige Zeit.

Der Anlass für das Schätzprojekt war – wie so oft, eine bevorstehende Migration. Die letzten PL/I und VisualAge Entwickler sind, wie das System selbst, "in die Jahre gekommen". Nachwuchs ist nicht mehr in Sicht, da die Sprachen von damals nicht mehr gelehrt werden. Demzufolge wird es zunehmend schwer, Personal zu finden, das bereit ist, sich mit den alten Technologien zu befassen. IBM hat dem Anwender in diesem Fall nahegelegt, nach Java auf einer IBM AIX Maschine zu migrieren.

Ein Versuch ist auch gemacht worden, dieser wurde jedoch bald wieder aufgegeben. Es war geplant, die VisualAge COBOL Programme in EGL – Enterprise Generation Language – zu versetzen und daraus Java Code zu generieren. Wegen technischer Probleme mit der Größe der Programme wurde dieser Versuch bald aufgegeben. Mit den PL/I Programmen ist es nicht besser gelaufen. Der Versuch, sie in Java umzusetzen, erwies sich als viel zu aufwändig. Nach dem gescheiterten Migrationsversuch stand der Anwender da und wusste nicht weiter. Daher der Auftrag, den Migrationsaufwand nochmals systematisch zu schätzen. Der Anwender brauchte Information für seine Entscheidungsfindung, u. a. darüber, welche Kosten anfallen würden, das System neu zu entwickeln, erneut zu konvertieren, oder es so zu erhalten wie es ist.

Dieses Messprojekt war wegen der Dringlichkeit der Situation auf fünf Tage beschränkt. Es begann mit dem Abzug der Source-Bibliotheken vom Mainframe-Rechner und der Übertragung auf einen PC-Arbeitsplatz. Das Messobjekt bestand aus

- 387 VA-COBOL Sourcen mit 6.822.599 Codezeilen
- 2.397 PL/I Sourcen mit 1.811.271 Codezeilen
- 324 BMS Masken mit 16.945 Codezeilen
- 387 DB-2 SQL Schemen mit 21.903 Codezeilen
- 303 MVS-JCL Prozeduren mit 540.502 Codezeilen.

Insgesamt gab es 3.798 Source-Dateien mit 9,38 Million Codezeilen. Die automatisierte Analyse dieser Source dauerte nicht länger als einen Tag.

Als Voraussetzung für die Schätzung der Erhaltungs- und Evolutionskosten war es notwendig, die Größe, Komplexität und Qualität des Codes zu messen. In diesem Fall handelte es sich um prozeduralen Code Die Größe wurde demzufolge in

- Anweisungen,
- Data-Points und
- Function-Points

gemessen. Da es hier um die Erhaltung des Systems ging, wurde die Größe in Anweisungen benutzt.

Die Code-Komplexität wurde als Mittelwert der einzelnen Komplexitätsmaße:

- Datenkomplexität
- Datenflusskomplexität
- Schnittstellenkomplexität
- Zugriffskomplexität
- Steuerflusskomplexität
- Bedingungskomplexität
- Verzweigungskomplexität
- Sprachkomplexität

errechnet.

Die Codequalität wurde als gewichteter Mittelwert der einzelnen Qualitätsmaße:

- Modularität
- Portabilität
- Wiederverwendbarkeit
- Testbarkeit
- Flexibilität
- Testbarkeit
- Lesbarkeit und
- Konformität

ermittelt.

Die Regeln für die Berechnung der einzelnen Komplexitäten und Qualitäten wurden der Literatur zu Softwaremetriken entnommen [EbDu07]. Diese Regeln mit geringfügigen Modifikationen werden seit mehr als 20 Jahren verwendet. Mit ihnen sind über 40 Messprojekte in fünf verschiedenen Ländern ausgeführt worden [Sned08].

Mit den aggregierten Komplexität- und Qualitätsmaßen wird die Größe der Software justiert. Im Falle des PL/I Codes gab es insgesamt 916 Kilo deklarative und prozedurale Anweisungen. Die aggregierte Komplexität betrug 0,531 und die aggregierte Qualität 0,647.

Nach der Justierung

*916 * (0,531 / 0,647)*

gab es 752 Kilo Anweisungen als justierte Größe.

Für die Kalkulation der Erhaltungskosten sind zwei weitere Größen erforderlich:

- die Änderungsrate und
- die Wartungsproduktivität.

Was die Änderungsrate anbetrifft, müsste gemessen werden, um wieviel sich der Code jährlich ändert. Dies geschieht durch einen Abgleich der Anzahl Anweisungen am Ende des Jahre mit der Anzahl Anweisungen zu Beginn des Jahres. Von den 916 PL/I Anweisungen sind nur 17.488 im letzten Betriebsjahr dazugekommen. Das ergab eine Änderungsrate von 2%, weit unter dem Durchschnitt von 5%. Allerdings wird davon ausgegangen, dass aufgrund zurückgestellter Anforderungen und etlicher neuer Gesetze bezüglich der Beamtenbezahlung sich diese Rate in der nächsten Zeit verdoppelt wird. Gerechnet wird mit einer Änderungsrate von 4%. Das ergibt dann 36.651 Anweisungen. Die Justierung dieser Größe durch die Komplexität und Qualität ergibt folgende Größe:

*36.651 * (0,531 / 0,647) = 30.054.*

Die relativ niedrige Komplexität – knapp über den Durchschnitt – verbunden mit der hohen Qualität – trägt dazu bei, die zu bearbeitende Codemenge um 18% zu verringern. So

schlägt also Komplexität und Qualität bei der Softwareerhaltung zu Buche, in diesem Falle positiv.

Die andere Größe – die Wartungsproduktivität – ist zu ermitteln, indem man die Anzahl hinzugekommener und geänderter Anweisungen durch die Anzahl Personentage dividiert, die im letzten Jahr gegen die Wartung des Systems gebucht wurden. Im letzten Betriebsjahr dieses Systems wurden 4.416 Stunden bzw. 552 Personentage gegen die Erhaltung des Systems gebucht. Daraus erfolgt eine Produktivität von

17.488 / 552 = 32 Anweisungen pro Personentag.

Der zu erwartenden Aufwand für die Erhaltung des Personalabrechnungssystems wurde als die justierte Größe in Anweisungen:

30.054 Stmts / 32 Stmts pro Tag = 939 PTs, bzw. 47 Personenmonate

ermittelt.

Zu diesen reinen Operationskosten müssen 50% Managementkosten hinzugerechnet werden. Es kommen also weitere 24 Personenmonate dazu:

47 PMs + (47 PMs / 2) = 71 Personenmonate insgesamt.

Dies spricht für eine Wartungsmannschaft von circa 7 Personen und ein jährliches Erhaltungsbudget von rund € 350.000.

Falls größere Erweiterungen vorkommen, bei denen mehr als 5% des Codeumfanges betroffen ist, oder falls Teile des Codes konvertiert werden, müssen diese Vorhaben als Evolutionskosten extra geschätzt und berechnet werden.

6 Folgerungen aus den beiden Fallstudien

Aus den hier präsentierten Studien werden folgende Schlüsse bezüglich der Erhaltung bestehender Softwaresysteme gezogen.

Erstens ist die Anzahl der zu pflegenden Anweisungen ausschlaggebend für die Größenmessung. Es sind die *Anweisungen*, die zu bearbeiten sind und nicht die Funktionen.

Zweitens ist die *Größe* eines Systems nicht alleine ausschlaggebend. Die Komplexität und Qualität des Systems kommen hinzu. Je niedriger die Komplexität und je höher die Qualität, desto geringer der Erhaltungsaufwand. Je höher die Komplexität und je niedriger die Qualität, desto größer der Erhaltungsaufwand. Deshalb muss die Größe durch Komplexität und Qualität justiert werden.

Drittens sind *Erhaltungskosten* nur voraussagbar, wenn man weiß, um wieviel sich die Software verändern wird. Statische Systeme, die sich kaum verändern, sind

leichter zu planen als dynamische Systeme, die sich ständig verändern. Für dynamische Systeme gelten andere Regeln.

Viertens variiert die *Wartungsproduktivität* von Betrieb zu Betrieb erheblich. Daher sollte jeder Wartungsbetrieb seine Wartungsaufwände pedantisch buchen und mit der Zahl der betroffenen Anweisungen vergleichen. Nur auf diese Weise werden sie zu zuverlässigen Schätzaussagen kommen. Leider wird dies in den wenigsten Betrieben getan. Produktivitätsmessung gilt als politisch inkorrekt. Solange diese Einstellung herrscht, wird es kaum möglich sein, vernünftige Schätzungen durchzuführen.

Fünftens sollten Software-Produktmanager den Zustand ihrer Produkte ständig verfolgen, um Änderungen der Größe, Komplexität und Qualität der Software zu erkennen und darauf zu reagieren. Dazu brauchen sie ein "Armaturenbrett", das von Analysewerkzeugen automatisch gefüttert wird. Es kommt darauf an, die Schätzungen fortwährend zu kalibrieren (siehe Abbildung 1).

Als Folge der hier beschriebenen Messpraxis sollten andere Anwendungsbetriebe ermuntert werden, ihre vorhandenen Softwaresysteme zu messen und deren Erhaltung und Evolution systematisch zu planen, auch wenn sie noch keine eigenen Wartungsproduktivitätsdaten haben. Diese können sie im Laufe der Zeit allmählich gewinnen. Die Kosten der Systemerhaltung und –evolution sind viel zu hoch, als dass sie einfach pauschal berechnet werden.

Literaturverzeichnis

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

[BeRa00] Bennet, K.; Rajlich, V.: Software Maintenance and Evolution – A staged Model. Proc. On the Future of Software Eng., ICSE-2000, IEEE Press, Limmerick 2000, S.73.

[BGKS08] Blom, S.; Gruhn, V.; Koehler, A.; Schaefer, C.: Methoden und Grundlagen der wertebasierten Softwareentwicklung, Object Spektrum, Nr. 1, Feb 2008, S.12.

[CHKR01] Chapin, N.; Hale, J.; Kahn, K.; Ramil, J.; Tan, W.: Types of Software Evolution and Maintenance, Journ. of Software Maint. & Evolution, 13(2001) 1, S.3.

[Cohn05] Cohn, M.: Agile Estimating and Planning, Prentice-Hall, Englewood Cliffs, 2005.

[EbDu07] Ebert, C.; Dumke, R.: Software Measurement, Springer, Berlin et al. 2007.

[Hatt07] Hatton, L.: How Accuratly do Engineers Predict Software Maintenance Tasks? IEEE Computer, Feb. 2007, S.64.

[Jone98] Jones, C.: Estimating Software Costs, McGraw-Hill, New York, 1998, S.101.

[K-MA00] Kajko-Mattson, M.; Forsannander, S.; Andersson, G.: Software Problem Reporting and Resolution Process at ABB, Journ. of Software Maint., 12(2000) 5, S.255-286.

[KiFK04] Kusumoto, S.; Fumikazu, M.; Katsuro, I.: Estimating Effort by Use Case Points, Proc. of IEEE Symposium on Software Metrics, Computer Society Press, Chicago, Sept. 2004, S.292.

[LaBr06] Laird, B.; Brennan,C.: Software Measurement and Estimation – A Practical Approach, Wiley & Sons, New York 2006, S.133.

[LiSw80] Lientz, B., Swanson, E.B.: Software Maintenance Management, Addison-Wesley, Rreading, 1980, S.105.

[MaOs83] Martin, R.J., Osborne, W.: Guidance of Software Maintenance, U.S. National Bureau of Standards, NBS Pub. 500-129, Dec. 1983.

[NiVl00] Niessink, F.; van Vliet, H.: Software Maintenance from a Service Perspective, Journ. of Software Maint., 12(2000), 2, S.103-120.
[Rama00] Ramaswamy, R.: How to staff business critical Maintenance Projects. IEEE Software, June 2000, S.90.
[SnMe85] Sneed, H.; Merey, A.: Automated Software Quality Assurance, IEEE Trans. on SE, 11(1985) 9, S.896.
[Sned91] Sneed, H.: Economics of Software Reengineering. Journ. of Software Maint., 3(1991) 1, S.163.
[Sned95] Sneed, H.: Understanding Software through Numbers, Journ. of Software Maintenance, 7(1985) 6, S.405.
[Sned95] Sneed, H.: Estimating the Costs of Software Maintenance Tasks, IEEE Proc. of 11th ICSM, Opio France, Oct. 1995, S.168.
[Sned01] Sneed, H.: Impact Analysis of Maintenance Tasks for a Distributed Object-oriented System. IEEE Proc. of the 17th ICSM, Florence, Nov. 2001, S.180.
[Sned04] Sneed, H.: A Cost Model for Software Maintenance and Evolution, in Proc. of 20th ICSM, IEEE Press, Chicago 2004, S.264.
[SnHT04] Sneed, H.; Haschitschka, M.; Teichmann, M.T.: Software-Produktmanagement. dpunkt, Heidelberg 2004, S.18.
[Sned05] Sneed, H.: Software-Projektkalkulation, Hanser, München 2005, S.117.
[Sned08] Sneed, H.: Measuring 70 Million Lines of Code, Proc. of Int. Workshop on Software Measurement, Springer, München 2008, S.271.
[Sned11] Sneed, H.; Baumgartner, M.; Seidl, R.: Software in Zahlen. Hanser, München – Wien, 2011.
[SnEr12] Sneed, H., Erdös, K.: Evaluating a DotNet Application System. Proc. of Int. Workshop on Software Measurement (IWSM-2012), Assisi, Italy, 2012, S.91.