Evolutionsschätzung

Harry M. Sneed

ANECON Gmbh, Wien

Abstrakt: Der folgende Betrag ist ein Ausschnitt aus dem neuen Buch von Harry M. Sneed zum Thema Software-Evolution, das demnächst im dpunkt Verlag erscheint. Er schreibt ein Verfahren vor um die Kosten der nächsten Release-Periode zu ermitteln. Das Verfahren nutzt neue Ansätze zur Voraussage der Fehleranzahl und zur Projektierung der Fehlerbehebungskosten sowie zur Kalkulation des Deltas zwischen dem alten und dem neuen Release aufgrund einer Impaktanalyse und zur Schätzung des Aufwandes um dieses Delta bereitzustellen.
Software-Evolution beinhaltet die parallelen Aktivitäten Fehlerbehebung, Änderung, Sanierung und Weiterentwicklung eines bestehenden Softwareproduktes. In regermäßigen Intervallen von drei bis sechs Monaten werden neue Releases des Produktes herausgegeben, in denen neue Funktionen und Daten erscheinen. Gleichseitig wird im letzten Release Fehler beseitigt und Notänderungen durchgeführt. Der Produktleiter muss deshalb bei der Planung der nächsten Release-Periode beides berücksichtigen – die Pflege des zuletzt ausgelieferten Release und die Bereitstellung des nächsten Release, denn beide Aktivitäten laufen neben einander her.

1 Die Notwendigkeit einer Evolutionsstatistik

Die Voraussetzung für eine Evolutionsplanung ist eine Erfahrungsdatenbank mit statistischen Daten über bisherigen Releases. Die Daten stammen zum Teil aus der Produktanalyse und zum anderen Teil aus der Prozessanalyse. Sie werden von der Datenbankverwaltungssoftware aggregiert, ausgewertet und in Form von Graphiken und Berichten ausgegeben. Am Ende werden sie benutzt um die Kosten der nächsten Release-Periode zu kalkulieren. (siehe Abbildung 1)

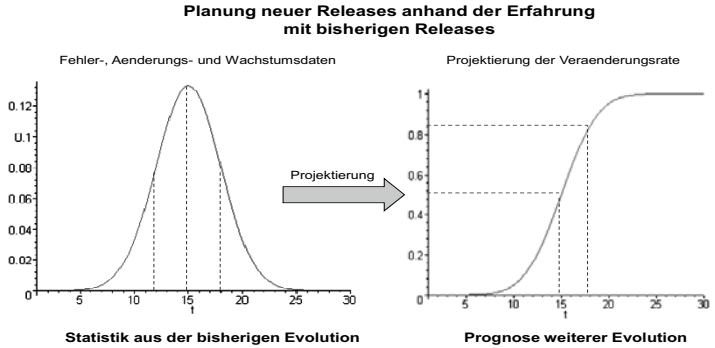

Abbildung 1: Release-Planung anhand von Erfahrungen

1.1 Statistik aus der Produktanalyse

Die Analyse der Software wirft viele Zahlen ab. Neben den Zahlen über den Source Code werden Zahlen über die Anforderungen, den Systementwurf und die Testware produziert.

Vieles wird gezählt, wie z.B. die Anzahl Dokumente, die Anzahl Anforderungen, die Anzahl Entwurfsdiagramme, die Anzahl Datenentitäten, die Anzahl Anwendungsfälle und die Anzahl Source Code Module. Aus diesen elementaren Zahlen werden aggregierte Zahlen wie die Anzahl Function-Points, die Anzahl Use-Case-Points, die Anzahl Object-Points und die Anzahl Test-Points abgeleitet. Außerdem werden Komplexitäts- und Qualitätsmetriken errechnet. Es gibt viele Komplexitäten und ebenso viele Qualitäten. In der Metrikdatenbank muss eine Auswahl getroffen werden. Es darf nur eine Untermenge der möglichen Metriken herangezogen werden. Die Metriken sollen aber der Gesamtkomplexität und –Qualität widerspiegeln. Für die Komplexität sind es Metriken wie

- die Schnittstellenkomplexität
- die Zugriffskomplexität
- die Entscheidungskomplexität und
- die Ablaufkomplexität.

Für die Qualität sind es Metriken wie

- die Modularität
- die Flexibilität
- die Wiederverwendbarkeit und
- die Testbarkeit.

Hinter jeder diese Metriken steckt eine Formel wie sie aus den Grundzahlen errechnet wird. Immer wenn die Grundzahlen sich ändern, werden diese abstrakte Metriken neu berechnet. (siehe Abbildung 2).

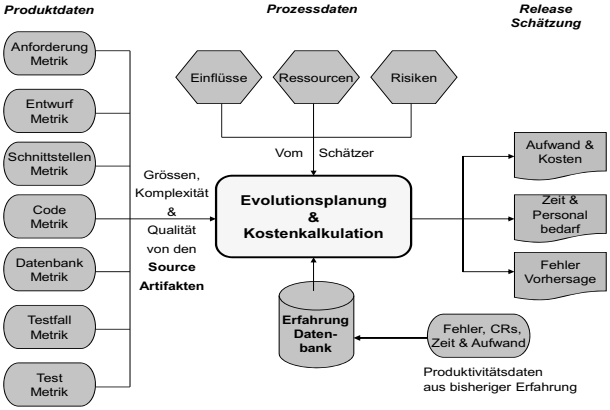

Abbildung 2: Evolutionsplanung & Kostenkalkulation

1.2 Statistik aus der Prozessanalyse

In dem Evolutionsprozess fallen Fehlermeldungen, Änderungsaufträge und neue Anforderungen an. Sie werden hier in der Summe als Evolutionsanträge verzeichnet. Es kommt darauf an, diese Aufträge zu klassifizieren und zu zählen.

Fehlermeldungen werden nach der Schwere des Fehlers klassifiziert in

- katastrophale Fehler
- kritische Fehler
- schwere Fehler und
- leichte Fehler.

Änderungsanträge werden nach der Art der Änderung in

- technische Änderungen bzw. Sanierungen
- fachliche Änderungen und
- Datenänderungen

unterteilt.

Neue Anforderungen unterscheiden sich nach

- funktionalen Anforderungen und
- nicht-funktionalen Anforderungen.

Es ist für die Metrikdatenbank wichtig, sämtliche Evolutionsanträge nach Auftragsart und Release zu zählen, d.h. jeder Auftrag muss einem bestimmten Release zugeordnet werden.

Diese Zählung der Attribute geschieht über eine Analyse der Antragsdokumente. Sie werden gescanned um die wichtigsten Attribute herauszuholen und an die Metrikdatenbank zu übergeben. Hier werden sie aggregiert und benutzt, um das nächste Release zu planen. Letztendlich werden auch die Aufwandsdaten benötigt. Die Mitarbeiter der Evolutionsmannschaft haben ihre gearbeitete Stunden gegen die Evolutionsaufträge zu buchen. Für jeden Auftrag werden die gebuchten Stunden summiert. Aus der Summe der gegen ihn gebuchten Stunden folgen die Kosten eines Auftrags. Die Summe der Kosten für alle Aufträge eines Releases ergibt die Kosten des jeweiligen Releases. Hinzu kommen die Materialkosten für Hardware-, Software und Raumnutzung in der besagten Zeit. Es ist deshalb so wichtig eine genaue Kostenrechnung über die Kosten der letzten Releases zu führen damit die Produktleitung die Kosten der nächsten Releases daraus ableiten kann. (siehe Abbildung 3)

Das gleiche gilt für den Nutzen der Releases. Der Nutzen eines Releases ergibt sich aus den korrigierten Fehler, den durchgeführten Änderungen und den funktionalen Erweiterungen. So, wie bei der Polizei, wo die Leistung Anhand der aufgeklärten Fälle gemessen wird, wird hier die Leistung Anhand der erledigten Aufträge gemessen, d.h. soviel behobene Fehler, erfolgte Änderungen und neue Anforderungen erfüllt.

Wenn es gelingt, diese Leistungen in Geld umzusetzen, z.B. die Behebung eines Fehlers spart 50.000 Euro oder die Durchführung einer Änderung ist dem Anwender 100.000 Euro Wert, dann umso besser, aber dies sei nicht absolut notwendig. Die Leistung der Ordnungshüter können auch nicht genau in Mark und Pfennig ausgedrückt werden. Was ist denn die Aufklärung eines Verbrechens Wert? Die gleiche Frage stellt sich bei der Schließung einer Sicherheitslücke in der Software oder bei der Anpassung an einer Gesetzesänderung. Es muss sein, um die Ordnung zu erhalten bzw. den Betrieb zu erhalten, koste was es wolle.

2 Die Schätzung eines neuen Release

Die Schätzung eines neuen Software Releases besteht aus zwei verschiedenen Schätzungen:

- die Schätzung der Fehlerbeseitigung im letzten Release und
- die Schätzung der Erstellung des nächsten Release.

Beide Schätzungen haben zwar die gleiche Ausgangsbasis, nämlich die gegenwärtige Software, aber unterschiedliche Vergleichswerte. Die Schätzung der Fehlerbeseitigung geht von der Anzahl der bisherigen Fehler relative zur Softwaremenge aus. Die Schätzung der Weiterentwicklung geht hingegen von dem Umfang der betroffenen Softwaremenge – den Auswirkungsbereich - relative zur Gesamtmenge aus. (siehe Abbildung 4)

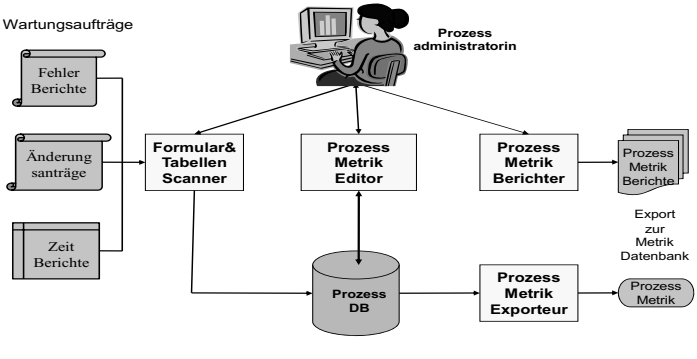

Abbildung 3: Steuerung des Evolutionsprozesses

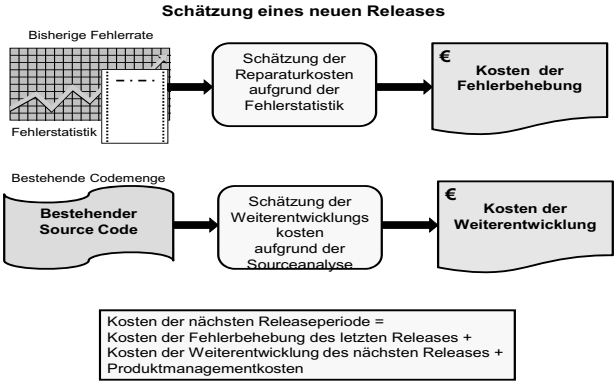

Abbildung 4: Schätzung eines neuen Release

2.1 Messung der gegenwärtigen Systemgröße

Als Erstes ist zu klären, was die Softwaremenge ausmacht. Das einfachste Mengenmaß ist die Anzahl der Source Code Anweisungen. Wohl bemerkt, hier werden Anweisungen und nicht Codezeilen gezählt. Codezeilen könnten auch benutzt werden, aber sie sind weniger aussagekräftig als Anweisungen bzw. Statements. Man könnte auch eine abstraktes Maß wie Anzahl Function-Points, Data-Points oder Use-Case-Points benutzen, aber diese Masse sind für die Messung bestehender Software weniger geeignet. Wie schon besprochen, wurden sie für die Entwicklung von Software konzipiert und beziehen sich mehr auf das Soll als auf das Ist. Anweisungen lassen sich mit einem Source Analyse Werkzeug schnell und zuverlässig zählen. Sie sind auch der eigentliche Gegenstand der Evolution.

Auch noch zu berücksichtigen sind die Testfälle. Testfälle sind ebenso Bestandteil eines Softwareproduktes wie der Code selbst. Sie sollten im Gleichschritt mit dem Code

fortgeschrieben werden, da sie ein Spiegelbild dessen sind, was der Code sein sollte. Daher zählt man nicht nur die Anzahl Anweisungen sondern auch die Anzahl Testfälle.

Als dritte Größe käme die Anzahl Entwurfsdokumente bzw. die Anzahl Modellelemente in Frage. Das Problem hier ist, dass Entwurfsdokumente selten fortgeschrieben werden. Das Systemmodell wird, wenn überhaupt, einmal zu Beginn einer Entwicklung aufgebaut. Es dient sozusagen als Starthilfe, Danach bleibt es stehen, während der Code und der Test weiterentwickelt werden. Es hat nur Sinn das Modell als Größenmaß zu verwenden, wenn es tatsächlich im Einklang mit dem Code ist und bleibt. In dem Falle könnte man auch die Anzahl der zu pflegenden Modellelemente zählen.

Schließlich sind noch die Daten zu berücksichtigen, Neben dem Code und dem Test und eventuell noch der Entwurfsdokumentation werden die Daten gewartet und weiterentwickelt. Die Datenbanken werden mittels Schemen beschrieben. Das Schema bestimmt die Topologie und die Typologie einer Datenbank. Daraus ist zu entnehmen, wie viele Datenentitäten mit wie vielen Attributen und wie viele Beziehungen in einer Datenbank sind. Hinzu kommen die Regel und Zugriffsprozeduren, die mit den Daten zusammen gespeichert werden. Da auch die Datenbanken korrigiert, geändert und erweitert werden, müssen sie in die Evolutionsschätzung einbezogen werden. Der Grundmaß für die Menge der Daten ist die Anzahl der Datenattribute bzw. Felder, unabhängig davon, wie oft sie vorkommen.

Zusammenfassend besteht ein Software Release aus

- Codeanweisungen
- Testfälle
- Datenbankattribute und
- eventuell Entwurfsdiagramme.

Um den Umfang eines Release zu bestimmen sollten alle vier Größen zu einer Composite Größe zusammengefasst werden, denn alle vier Mengen sind Teil des nächsten Releases. Es geht auch wenn nur zwei Größen – Anweisungen und Testfälle – gezählt werden. Wenn nur der Code fortgeschrieben wird, reicht es die Anzahl Anweisungen als Größenmaß zu benutzen. Wenn jedoch der Test und die Daten fortgeschrieben werden, müssen deren Größen mitgezählt werden, und wenn auch noch das Entwurfsmodell aktualisiert wird, muss seine Größe mit in die Rechnung einbezogen werden.

Unabhängig davon, welche Größen man benutzt, sind sie alle durch die Komplexität und die Qualität der jeweiligen Menge zu justieren. So wird die Codegröße durch die Komplexität und Qualität des Codes justiert, die Datengröße durch die Komplexität und Qualität der Daten, die Testgröße durch die Komplexität und Qualität der Testfälle und die Entwurfsgröße durch die Komplexität und Qualität des Entwurfsmodells justiert. In allen Fällen ist die Justierungsformel die Gleiche:

*Justierte Größe = rohe Größe * Komplexitätsfaktor * Qualitätsfaktor*
Komplexitätsfaktor = gemessene Komplexität / mittlere Komplexität
Qualitätsfaktor = mittlere Qualität / gemessene Qualität

Erläutert wird dies am folgenden Beispiel eines DotNet Anwendungssystems. Die Vermessung des Systems hat ergeben, dass das System 1804 C# Source Dateien mit 496.124 Codezeilen und 309.823 Anweisungen hat. Die gemessene Komplexität des Codes ist 0,604. Die gemessene Qualität des Codes ist 0,506. Demnach ist die justierte Codegröße

*309.823 * (0,604 / 0,506) = 368.689 justierte Anweisungen*

Neben dem Code hat das System 170 Datenbanktabellen mit 2069 Datenattribute und 2053 Stored-Prozeduren mit 28.800 Anweisungen. Die gemessene Komplexität der Daten ist 0,402. Die gemessene Qualität der Daten ist 0,564. Demnach ist die justierte Datenbankgröße

*2069 * (0,402 / 0,564) = 1.720 justierte Datenelemente*
*28.800 * (0,402 / 0,564) = 20.528 justierte Anweisungen*

Schließlich gibt es für den Test des Systems 8.568 Testfälle mit einer Testfallkomplexität von 0,518 und einer Testfallqualität von 0,623. Danach ist es die justierte Testgröße

*8.568 * (0,518 / 0,623) = 7.120 justierte Testfälle*

Eine Entwurfsdokumentation wird wie üblich nicht gepflegt. Es werden nur die Codekomponente, die Datenbanken und die Testfälle fortgeschrieben. Also ist die Gesamtgröße der zu pflegenden Software

368.689 justierte C# Anweisungen
\+ 20.528 justierte SQL Anweisungen
\+ 1.720 justierte Datenbankattribute
\+ 7.120 justierte Testfälle
= 398.057 justierte Softwareelemente

2.2 Hochrechnung der Systempflegekosten

Die Pflege bezieht sich auf das Release mit sämtlichen Varianten, die zurzeit im Einsatz ist. Behandelt werden alle gemeldeten Probleme, die sich nicht auf das nächste Release verschieben lassen – Fehler, die den Betrieb der Software beeinträchtigen bzw. die Nutzung erschweren, aber auch Anpassungen, die nicht auf sich warten lassen. In der ITIL Terminologie spricht man von Vorfällen. Ein Vorfall kann die Folge einer Fehlbedienung oder einer Unstimmigkeit in der Umgebung sein, aber meistens ist er die Folge eines Fehlers in der Software. In der ITIL Konvention ist vorgeschrieben, sämtliche Problemmeldungen zu registrieren und darüber Buch zu halten. Es ist jedoch auch ohne ITIL Konvention sinnvoll alle Probleme zu verfolgen. Dies ist die Grundlage für einen ordentlichen Wartungsbetrieb.

Die aufbewahrten Problemberichte lassen sich auf verschiedene Art und Weise auswerten. Hier geht es um die Führung einer Fehlerstatistik. Wir wollen wissen, wie oft welche Fehler vorkommen und was es gekostet hat sie zu beheben. Zu diesem Zweck sind die Fehler nach Schwere zu klassifizieren. Die IEEE Standard 1044 schreibt vier Fehlerklassen vor:

- kritische Fehler
- schwere Fehler
- mittelschwere Fehler und
- leichte Fehler.

Hinzu kommen die unaufschiebbaren Anpassungen.

Für jede dieser Klassen werden die Aufwände zu deren Korrektur zusammengefasst. Hier ist der Aufwand die Summe der Arbeitsstunden, die zur Beseitigung des Problems erforderlich sind, z.B. wurde bisher für die Beseitigung von

- 8 kritischen Fehler 160 Stunden,
- 16 schweren Fehler 256 Stunden
- 24 mittelschweren Fehler 192 Stunden
- 20 leichten Fehler 80 Stunden
- 10 Anpassungen 120 Stunden

gebraucht.

Demzufolge ist

- ein kritischer Fehler = 20 Stunden
- ein schwerer Fehler = 16 Stunden
- ein mittelschwerer Fehler = 8 Stunden
- ein leichter Fehler = 4 Stunden und
- eine notwendige Anpassung = 12 Stunden.

Diese Arbeitsstunden beinhalten den gesamten operativen Aufwand von der Fehleranalyse bis hin zum Regressionstest und die Wiedereinspielung der korrigierten Komponente in die Produktion. Damit haben wir den Aufwand für die bisherige Pflege dieser Software. Wir wissen jetzt, was die Behebung jeder Problemklasse in etwa gekostet hat.

Man könnte diesen Aufwand, z.B. 16 Arbeitsstunden für jeden schweren Fehler, einfach auf die gegenwärtige Version übertragen. Das wäre jedoch eine Vereinfachung, denn das System wird immer größer und komplexer und somit immer schwieriger zu handhaben. Wir müssen daher dem Unterschied zwischen den justierten Release-Größen Rechnung tragen.

Das jetzige Release, das im Einsatz ist, hat eine justierte Größe von 398.057 Elementen. Um die Aufwandssteigerung zu ermitteln müssen wir den Unterschied in justierter Größe zwischen dem jetzigen Release und dem Letzten berechnen, d.h. wir müssen die Größe des letzten Release mit der Größe des jetzigen Release vergleichen. Nehmen wir an, das letzte Release hatte

350.000 justierte C# Anweisungen
20.000 justierte SQL Anweisungen
1.500 justierte Datenbankattribute und
6.400 justierte Testfälle
= 377.900 justierte Softwareelemente.

Der Unterschied zwischen dem letzten und dem jetzigen Release beträgt

$$\frac{\textit{die jetzige justierte Größe}}{\textit{die letzte justierte Größe}} = \frac{398.057}{377.900} = 1.05$$

Demnach ist das jetzige Release um 5 % größer und komplexer, bzw. schlechter als das letzte Release. Ergo muss der Aufwand zur Pflege der jetzigen Releases um 5 % höher sein.

Die Anzahl Stunden zur Bearbeitung jeder Problemart wird um 1,05 erhöht. Daraus ergeben sich folgende Aufwände für die Behebung der verschiedenen Fehlerarten:

- kritische Fehler = 20 * 1,05 = 21 Stunden
- schwere Fehler = 16 * 1,05 = 17 Stunden
- mittelschwere Fehler = 8 * 1,05 = 8,5 Stunden
- leichte Fehler = 4 * 1,05 = 4 Stunden
- Anpassungen = 12 x 1,05 = 12,6 Stunden

Diese Steigerung der Pflegeaufwände widerspiegelt der steigenden Größe und Komplexität und der sinkenden Qualität des Zielsystems.

Als nächstes müssten wir wissen, wie viele Probleme von jeder Problemart es in dem gegenwärtigen Release geben wird. Natürlich könnten wir einfach behaupten, es werden genauso viele Probleme in diesem Release geben, wie es im letzten gegeben hat. Hatten wir im letzten Release 16 schwere Fehler, so wird es in diesem auch 16 geben. Aber das wäre zu

einfach. Wir sollten dem Trend zu weniger Fehlern pro Release Rechenschaft leisten. Wenn es im ersten Release 25 schwere Fehler gegeben hat, im zweiten 20 und im letzten 16, so haben wir eine absteigende Tendenz. Mit jedem Release sinkt die Anzahl schwerer Fehler um 25 %. Demnach hätten wir mit 12 schweren Fehlern im jetzigen Release zu rechnen.

Die Fehlerhäufigkeit sinkt jedoch nicht linear. Wir wissen von den Gesetzen der Evolution, dass mit steigender Größe und Komplexität die Qualität sinkt. Demnach müssen wir die veränderte, justierte Größe des Codes einbeziehen und aufgrund der Fehlerdichte hochrechnen. Das letzte Release hatte bei einer nicht justierten Größe von 305.000 C# und SQL Anweisungen 16 schwere Fehler gehabt. Das ergibt eine schwere Fehlerdichte von 0.000052. Das jetzige Release hat eine nicht justierte Anweisungszahl von 338.623 Anweisungen. Mit der gleichen Fehlerdichte kämen wir auf 17,6 schwere Fehler. Das würde bedeuten, dass die Fehlerrate relativ zur Systemgröße leicht gestiegen ist. In Wirklichkeit hängt die Fehlerrate auch von der Testüberdeckung ab und diese steigt vom Release zu Release. Beim letzten Release lag die gemessene Testüberdeckung bei 75 %. In diesem Release ist sie auf 80 % gestiegen. Man muss deshalb die Systemgröße durch den Prozentsatz ungedeckter Funktionalität justieren. Demnach hatte das letzte Release

305.000 * (1 – 0,75) = 76.250 nicht getestete Anweisungen

und das jetzige Release

338.623 * (1 - 0,80) = 67.725 nicht getestete Anweisungen.

Demzufolge hatte es beim letzten Release (76.250 / 8) 9.531 nicht getestete Anweisungen pro schweren Fehler gegeben. Wenn wir von dem gleichen Verhältnis Fehler zu ungetesteten Anweisungen ausgehen, so haben wir

67.725 / 9.531 = 7 schwere Fehler in diesem Release zu erwarten.

Für die anderen Problemarten wird der gleiche Algorithmus angewandt:

Anzahl Fehler = ungetesteter Anzahl Anweisungen / (letzte ungetestete Anzahl Anweisungen / Anzahl Fehler)

Demnach wird die Anzahl Fehler aller Klassen aufgrund der gestiegenen Testüberdeckung um 12,5 % sinken. Es wird geschätzt

7	kritische Fehler
14	schwere Fehler
21	mittelschwere Fehler
17	leichte Fehler und
9	Anpassungen
68	Problemfälle insgesamt

Um den zu erwartenden Pflegeaufwand zu kalkulieren brauchen wir jetzt nur noch die einzelnen Fehlerarten mit dem geschätzten Aufwand pro Fehlerart zu multiplizieren:

7	kritische Fehler	x 21	= 147	Stunden
14	schwere Fehler	x 17	= 238	Stunden
21	mittelschwere Fehler	x 8,5	= 178	Stunden
17	leichte Fehler	x 4	= 68	Stunden
9	Anpassungen	x 12,5	= 112	Stunden
63	Problemfälle		= 743	Stunden

Der Aufwand für die Pflege des aktuellen Releases, das jetzt ausgeliefert wird, wird auf 743 Stunden bzw. 93 Personentage geschätzt. Falls der Release-Intervall 3 Monate bzw. 60 Arbeitstage beträgt, brauchen wir 1,5 Mitarbeiter für den Pflegedienst.

2.3 Schätzung der Weiterentwicklungskosten

Der Aufwand für die Entwicklung des nächsten Release hängt von der Anzahl und dem Ausmaß der anstehenden Änderungsanträge und neuer Anforderungen ab. Bis zum Beginn der Arbeit am nächsten Release ist die Anzahl der Änderungen und Ergänzungen bekannt. Weitere ausschlaggebende Größen für die Schätzung des Weiterentwicklungsaufwandes sind die bisherige Produktivität und die Auswirkung der geplanten Änderungen und Anpassungen auf die aktuelle Version.

Die Produktivität in der Evolution ist am besten in Anweisungen oder Object-Points zu berechnen. Function-Points und Use-Case-Points sind, wie wir bereits festgestellt haben, für die Schätzung der Evolution zu grobkernig und zu ungleichmäßig verteilt über das System. Function-Points können nur für eine Anwendung in seiner Gesamtheit verwendet werden und nicht für Teile davon weil Function-Points auf der Interaktion zwischen dem System und seiner Umgebung basieren und diese Interaktion findet nur in bestimmten Systemteilen statt. Use-Case-Points gehen von einer 1:1 Umsetzung der Anwendungsfälle in Code aus und dies ist nur selten der Fall. Der Code wird optimiert um mit den gleichen Code-Bausteinen möglichst viele Anwendungsfällen zu bedienen. Je nachdem wie gut dies den Entwicklern gelingt, wird die Codemenge pro Anwendungsfall anders ausfallen. Deshalb muss der Aufwand für bisherige Releases in Relation zu der Anzahl Anweisungen oder Object-Points gesetzt werden. Man nimmt einfach die justierte Größe des betroffenen Codebereiches im letzten Release und dividiert sie durch die Anzahl der gegen das Release gebuchten Stunden.

Das jetzige Release hat eine Gesamtgröße von 349.260 Anweisungen einschließlich Datenbankattribute und Testfälle. Nehmen wir an 10% davon, bzw. 34.926 Elemente, ist beim letzten Release dazu gekommen. Dagegen wurden 10.000 Stunden gebucht. Das ergibt eine Produktivität von 3,5 Anweisungen pro Arbeitsstunde, bzw. 28 Anweisungen pro Personentag.

Produktivität = 34.926 betroffene Anweisungen
/ 10.000 Arbeitsstunden
= 3,5 Anweisungen pro Stunde.

Der Auswirkungsbereich des neuen Release geht aus der automatisierten Impaktanalyse hervor. Hier wird für jeden Änderungsantrag der Code, die Daten und die Testfälle daraufhin untersucht, ob sie von der Änderung direkt oder indirekt betroffen sind. Bei direkt betroffenem Artefakte werden alle Anweisungen mitgezählt. Bei indirekt betroffenen Artefakten werden nur ein viertel oder 25 % der Anweisungen gezählt. Es stellt sich heraus, dass 200 Artefakte mit insgesamt 4000 Anweisungen direkt und 400 Artefakte mit insgesamt 8000 Anweisungen indirekt betroffen sind. Da die indirekt betroffene Anweisungen nur zum viertel gezählt werden, ergibt dies

*4000 direkte + (8000 * 0,25) = 6.000 Anweisungen,*

die von den geplanten Änderungen betroffen sind.

Es ist außerdem geplant 2 neue Komponente hinzufügen. Diese beiden Komponenten werden jeweils circa 2000 Anweisungen, 20 neue Datenelemente und 200 Testfälle haben. Das macht 2220 Anweisungen pro Komponente oder *4440 Anweisungen* insgesamt. Zusammen mit den 6.000 Anweisungen, die durch die Änderungen betroffen sind, kommen wir auf eine Gesamtzahl von *10.440 Anweisungen*, die entweder geändert oder hinzugefügt werden.

Wenn wir diese Größe des Impact-Domains durch die letzte Produktivität von 3,5 Anweisungen pro Stunde dividieren, so kommen wir auf einen Aufwand für die Weiterentwicklung von *2.983 Stunden* oder *373 Personentage* für die Entwicklung und Test des nächsten Releases. Das sind 19 Personenmonate. Beim Release-Intervall von drei Monaten brauchten wir 6 Mitarbeiter um das nächste Release zu bauen.

2.4 Kalkulation der Gesamtkosten des nächsten Release

Die Kosten eines Release-Zyklus setzen sich zusammen aus dem

- Kosten für die Pflege des jetzigen Releases
- Kosten für die Entwicklung des nächsten Releases und
- Kosten für das Management Overhead.

Die Kosten für das Management Overhead belaufen sich aufgrund der Erfahrungswerte auf mindestens 25 % der Gesamtkosten. Dazu gehören nicht nur die Kosten des Produktmanagements sondern auch die Kosten des Konfigurations- und des Qualitätsmanagements.

Die Gesamtkosten sind demnach:

93 Personentage für die Pflege des gegenwärtigen Release
+ 373 Personentage für die Entwicklung des neuen Release
= 466 Personentage / 3 = 155 Personentage für das Management-Overhead

Daraus folgt, dass wir mit Kosten im Höhe von 621 Personentage für die nächste Release-Periode zu rechnen haben, wobei 25% = 155 Personentage für Produktmanagement und Support-Aktivitäten erforderlich sind. Sollte der Release-Intervall bei nur 3 Monate bleiben, brauchten wir 10 Mitarbeiter für die Pflege und Weiterentwicklung des Produktes einschließlich Management und Support-Personal. Wenn wir die 10 Mitarbeiter nicht haben, müssten wir der Release-Intervall auf 4 oder gar 6 Monate ausdehnen. Mit einem Release-Intervall von 6 Monaten brauchten wir nur 5 Mitarbeiter – ein Produktmanager, drei Entwickler und ein Tester.

Da die Schätzung der Evolutionsaufwände derart kompliziert ist, ist es am besten wenn der Schätzer mit einem Schätzwerkzeug durch das komplexe Schätzverfahren geführt wird. Auch dann bleibt das Verfahren komplex, aber viele Routineberechnungen werden vom Tool abgenommen und, was wichtiger ist, das Tool erinnert den Schätzer daran welche der vielen Faktoren er zu berücksichtigen hat. Neben den Größen, Komplexitäten und Qualitäten, kommen die Produktivität und Einflussfaktoren hinzu. Es ist nicht einfach solche Werkzeuge zu bedienen. Deshalb muss der Schätzer ein ausgebildeter Experte sein.