Nachhaltigkeit durch gesteuerte Software-Evolution

Harry M. Sneed

ANECON GmbH, Wien,
Universität Regensburg, Bayern
Harry.Sneed@T-Online.de

Abstract: In diesem Beitrag zur Förderung der Softwarenachhaltigkeit wird der herkömmliche Begriff des Projektes in Frage gestellt. Es komme weniger darauf an, Softwareprojekte auszuführen als vielmehr, Softwareprodukte zu bauen und über die Zeit ständig auszubauen und nachzubessern. Ein Softwareprodukt ist nie fertig bzw. „Done", es wird so lange weiterentwickelt, bis keiner mehr daran Interesse hat. Ergo kann ein Projekt niemals abgeschlossen sein. Der Beitrag beschreibt, wie Softwareprodukte entstehen und über viele aufeinander folgende Releases immer reifer und nützlicher werden. Der Anwender arbeitet stets mit einem vorübergehenden Zustand, nie mit einem endgültigen. Da die IT-Welt immer im Wandel begriffen ist, kann es keinen Endzustand geben. Dieser Ansatz wird von einer Service-orientierten Architektur sowie von der aufsteigenden Cloud-Technologie geradezu gefördert. In einer betrieblichen SOA werden die angebotenen Services fortlaufend erneuert. Da diese unfertigen Teile in die eigene Anwendungssoftware eingebaut sind, kann auch die eigene Software nie fertig werden. Sie hat immer nur einen Stand erreicht. Dies gilt umso mehr für Services aus der Cloud, wo der Anwender noch weniger Einfluss auf die Weiterentwicklung seiner Komponente hat. Das ändert wie unsere Systeme geplant und finanziert werden. Die Planung und Kostenkalkulation beschränkt sich auf ein Release, bzw. auf einen Zeitrahmen von maximal drei Monaten. Das Budget für ein Produkt muss offen bleiben und nach jedem Release neu aufgestellt werden. Die IT ist in einer dynamischen Umwelt eingebettet und muss sich dieser anpassen. Dennoch darf diese Anpassung nicht in Chaos ausarten. Die Antwort ist eine gesteuerte Evolution im Einklang mit den wandelnden Anforderungen und den neuesten Serviceangeboten.

Keywords: Projektmanagement, Produktmanagement, Cloud-Computing, Cloud-Services, SOA, Change-Management, Softwarewartung, Softwareevolution

1 Das Wesen von IT-Projekten

IT-Projekte sind zeitlich begrenzte, einmalige Anstrengungen, um ein vorgegebenes Ziel bzw. Ergebnis zu erreichen. Wichtig sind hier die zeitliche Begrenzung und die Zielorientierung. Projekte steuern auf ein bestimmtes Ziel hin und haben dafür nur begrenzt Zeit und Geld. Meistens ist die Zeit wichtiger als das Geld, weil der Anwender damit rechnet, mit dem Ergebnis des Projektes ab einem bestimmten Zeitpunkt arbeiten zu können. Ist das Ziel einmal erreicht, ist das Projekt zu Ende bzw. „Done" [Mira02]. Die

Entwicklungsmannschaft wird aufgelöst und die daraus resultierende Software geht in die sogenannte Wartung.

Diese Denkweise ist verheerend, was die Nachhaltigkeit der Software anbetrifft. Um den Termin zu halten, werden sämtliche anderen Ziele wie Performanz, Sicherheit, Wartbarkeit und Ausbaufähigkeit geopfert. Alles was nicht sofort erkennbar ist, wird unter den Teppich geschoben. Der Anwender wird damit getröstet, dass alles später folgen kann. Dass dies aber nicht möglich ist, weil die Weichen dafür gar nicht gestellt sind, kann der naive Anwender nicht erkennen. Abgesehen davon ist das Personal, das diese Nachbesserung eventuell noch durchführen könnte, nicht mehr verfügbar. Kurzum, der Anwender wird betrogen, damit er glaubt, dass die Software wirklich fertig und das Projekt damit abgeschlossen ist.

Projekte beanspruchen Ressourcen bzw. Betriebsmittel, um zum gewünschten Ergebnis zu gelangen. Das Ausmaß der Ressourcen bestimmt die Größenordnung des Projektes. In IT-Projekten sind die Ressourcen von dreierlei Art:

- Hardware,
- Software,
- Personal.

Die Personalressourcen sind maßgeblich. Die Hardwarekapazität wie auch die Höhe der Softwarenutzungsgebühr hängt von der Anzahl der Köpfe ab. Somit gehen die Projektkosten aus der Anzahl der beteiligten Personen mal die Projektdauer hervor. Da der Anwender bemüht ist, seine Kosten so niedrig wie möglich zu halten, ist er auch bemüht, die Projektlaufzeit so kurz wie möglich zu halten. Der zu geringe Aufwand geht auf Kosten der Nachhaltigkeit. Statt die Software ordentlich zu konstruieren und die Qualität sorgfältig zu prüfen, wird sie in kurzen Sprints ohne unabhängige Qualitätssicherung „zusammengehauen“ und als eine benutzergerechte Lösung verkauft. Der Anwender, als Produkt Owner, sieht die Software nur von außen her über die Benutzeroberfläche. Wie es innen hinter der Oberfläche aussieht, weiß er nicht. Er bekommt es nur später zu spüren, wenn er sie ändern und erweitern möchte. Die Rechnung für die kurze Entwicklungszeit und die geringen Kosten folgt später und sie überwiegt bei Weitem das, was man durch die schnelle Entwicklung eingespart hat. Nicht nur das; die Masse an schlechtem, schwer handhabbarem Code nimmt immer mehr zu. Bald ertrinkt der Anwender in einer Flut von redundanten Codezeilen, aus dem er nicht wieder herauskommt – Code der eigentlich nie hätte geschrieben werden sollen, wenn er nur die Zeit genommen hätte zu prüfen, ob der Code nicht schon existiert.

Das Problem mit IT-Systemen ist, dass solche einmaligen, zeitlich begrenzten Anstrengungen eines Teams besessener Entwickler nicht ausreichen, um ein befriedigendes Ergebnis zu erzielen. Denn während sie darauf hin arbeiten, verschiebt sich das Ziel. Schon allein das Erreichen des Ziels verändert die Bedingungen, unter denen das Ziel angestrebt wurde (siehe das Heisenberg Prinzip). Deshalb gibt es kaum IT-Projekte, mit deren Ergebnissen die angeblichen Nutznießer am Ende wirklich zufrieden sind, auch nicht mit denen der agilen Entwicklungsprojekte [ZhPa11]. Die betriebswirtschaftlichen und

technischen Ausgangsbedingungen ändern sich zu schnell. Kaum ist die erste Version ausgeliefert, da schielen die Anwender schon auf die nächste. Die Ausgangsbedingungen haben sich geändert. Daher ist der Begriff „Projekt“ im herkömmlichen Sinne in Bezug auf IT-Systeme eher irreführend. Er verleitet zu der Annahme, es gebe so etwas wie eine endgültige Lösung, wo in Wirklichkeit nur Zwischenlösungen erzielt werden können [Howa01].

Ein geeigneterer Begriff wäre der Begriff „Produktevolution“. Ein Softwareprodukt durchzieht mehrere Evolutionsphasen. Es beginnt mit der Konzeption und Prototypbildung. Danach folgen mehrere Releases, mit denen der Anwender schon arbeiten kann. Bennett und Rajlich unterscheiden in ihrem Evolutionsmodell zwischen der Entwicklungsphase und der Evolutionsphase, aber diese Unterscheidung ist künstlich. Eigentlich ist alles nach dem Bau des ersten Prototyps eine Evolution bzw. eine permanente Weiterentwicklung bis hin zur Ablösung des Produktes. Laut dem Modell von Bennett und Rajlich folgt nach einigen Jahren Evolution eine ausgesprochene Erhaltungsphase (Maintenance Phase) bei der nur noch Restfehler korrigiert und kleine Änderungen vorgenommen werden [BeRa00] (siehe Abbildung 1).

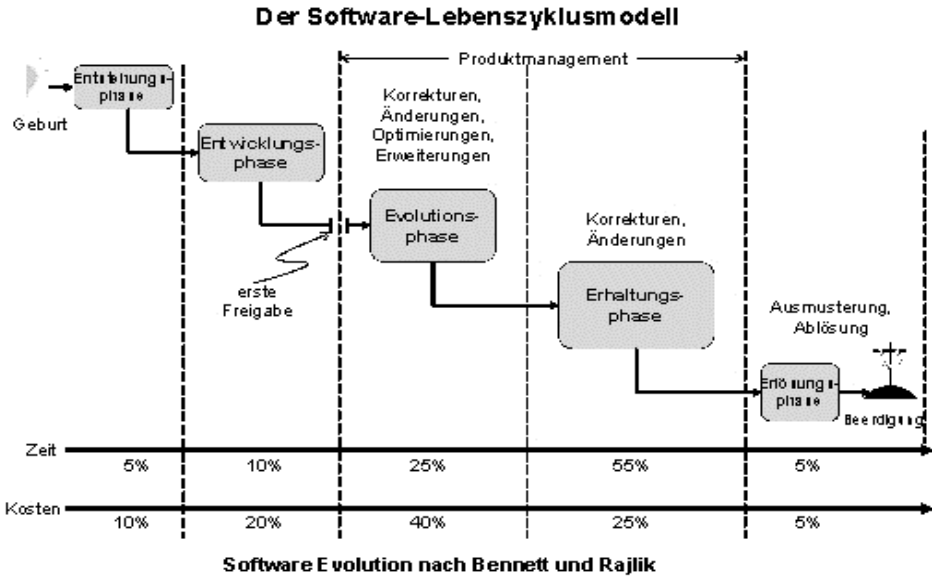

Abbildung 3: Modell der Software-Evolution

Das fünfstufige Phasenmodell trifft in Anbetracht der Dynamik einer Service-orientierten Softwarewelt nicht mehr zu. Ein modernes Service-basiertes System muss bis zu seinem Lebensende ständig weiter entwickelt werden. Am Ende bleiben nur drei Phasen übrig:

1. die Entstehungsphase, bzw. die Prototypentwicklung,

2. die Evolutionsphase, bzw. die Weiterentwicklung und

3. die Erlösungsphase, bzw. die Ausmusterung des Systems.

Wir müssen uns also von dem Begriff „Projekt“ befreien, ebenso wie die Deutschen sich von jeder Menge anderer alter Begriffe aus der NS-Zeit mit einem schlechten Beigeschmack befreit haben. Der Begriff „Projekt“ ist zu sehr mit den Begriffen „Zeit“ und „Aufwand“ assoziiert. Ein Evolutionsvorhaben zum Bau eines Softwareproduktes ist weder Zeit- noch Kostenverbunden. Es findet eben statt und dauert so lange wie das Produkt noch gebraucht wird und kostet so viel wie der Benutzer bereit zu bezahlen ist. Wenn er merkt, dass die Kosten ausufern kann er das Tempo der Evolution verlangsamen, die Mannschaft reduzieren oder das Vorhaben insgesamt aufgeben. Das Ziel des Projektes ist also nicht, ein bestimmtes Problem endgültig zu lösen, sondern ein Produkt bereitzustellen, mit dem die Anwender ihre Probleme immer besser lösen können. Das Produkt hat immer einen aktuellen Zustand. Die Zustände erfolgen in Intervallen. Der Übergang von einem Zustand zum anderen könnte man zwar als Projekt bezeichnen aber der Begriff „Release“ passt besser. Ein Release ist auch zeitlich und kostenmäßig begrenzt, aber im Gegensatz zum Projekt steht nicht das Ziel als Erstes an, sondern die Zeit und die verfügbare Kapazität. Das Ziel eines Release wird der gesetzten Zeit und dem erlaubten Aufwand angepasst. D.h. man setzt erst einen Termin, dann wird entschieden was man in der Zeit mit den vorhandenen Ressourcen erreichen kann. Wenn die Anforderungen mehr oder dringend sind, obliegt es dem Anwender das Budget für das Vorhaben aufzustocken. Insofern darf das Budget nicht fixiert sein. Es wird immer der Situation angepasst.

Software ist letztendlich ein geistiges Produktionsmittel, die eingesetzt wird, um bestimmte menschliche Arbeiten zu erleichtern oder gänzlich zu übernehmen. Derartige IT-Produkte kommen nicht auf Anhieb in einem einzigen einmaligen Projekt zustande, sondern sind das Ergebnis eines langen Reifeprozesses, das sich über einen längeren Zeitrahmen hinstreckt. Laut ISO-Standard 12207 ist dies der Produktlebenszyklus [ISO95]. Bis das Produkt gut nutzbar ist, bedarf es viele Releases. Auch danach folgen immer weitere Releases, allerdings nicht unbedingt in den gleichen kurzen Intervallen wie zuvor. Fertig bzw. „Done“ wird es nie. Der Weg ist das Ziel.

2 Risiken in der Produktentwicklung

Das eigentliche Thema ist demzufolge nicht das Projektmanagement sondern das Produktmanagement. Es müssen zunächst die IT-Produkte definiert und modelliert werden, ehe über ein Projekt nachgedacht wird, denn es sind die Produkte, welche die fachlichen und technischen Anforderungen erfüllen. Produkte dienen einem wirtschaftlichen Zweck, und sie haben eine technische Basis. Sie existieren unabhängig von Projekten. Deshalb müssen wir weg vom Projektdenken hin zum Produktdenken. Produkte und ihre Architektur sollen künftig im Vordergrund der Betrachtung stehen, statt wie bisher Projekte.

Wenn dies gelingt werden die Risiken und damit die vielen Misserfolge der Softwareentwicklung zwar nicht aus der Welt geschafft aber erheblich reduziert. Die fünf Hauptrisiken, die nach DeMarco und Lister immer wieder vorkommen sind

- falsche Einschätzungen der Termine und Aufwände,
- keine Einigung über die angestrebten Ergebnisse,
- ständig veränderte Anforderungen (*creeping requirements*),
- Verlust an Schlüsselpersonen und
- Überschätzung der eigenen Leistung [DeLi03].

Der Verlust maßgeblicher Personen ist ein Risiko, wovor kein menschliches Vorhaben gefeit ist. Nicht nur IT-Projekte, sondern auch IT-Produkte und sogar Softwarefirmen, sind von Schlüsselpersönlichkeiten abhängig. Dies liegt am Wesen von Software als geistige Substanz. Durch eine langfristige Strategie können aber die Folgen vom Personalausfall abgemildert werden. Man hat mehr Zeit um auf den Personalverlust zu reagieren und kann auch mit Personalreserven besser vorbauen. Die anderen vier Risiken haben jedoch mit dem Wesen des Projektmanagements zu tun. Sie sind alle vier Folgen einer falschen zeitlichen und finanziellen Begrenzung der Projekte.

2.1 Falsche Einschätzung der Termine und Aufwände

Das Unvermögen der Menschen, IT-Projekte richtig abschätzen zu können, liegt hauptsächlich daran, dass es unmöglich ist, die Dimensionen eines komplexen IT-Systems im Voraus zu bestimmen. Eine zuverlässige Schätzung ist nur dann möglich, wenn genug Erfahrung mit dem Sachgebiet und der anzuwendenden Technologie vorliegt, d.h., wenn das gleiche Problem mit den gleichen Mitteln schon mehrmals gelöst wurde. Je öfter ähnliche Anwendungen mit denselben technologischen Mitteln entwickelt werden, desto solider die Schätzbasis.

Leider ist dies in der IT Welt nur selten der Fall. Sowohl die Anwendungen als auch die technischen Mittel werden stets komplexer und differenzierter. Selten hat man die Gelegenheit, die gleiche Projektart zu wiederholen. Etwas ist immer anders – das Sachgebiet, die Rahmenbedingungen, die Technologie oder die Menschen. Die Produktivität, die ein Betrieb mit CICS/COBOL auf dem Host oder mit CORBA und C++ in einer Client/Server Umgebung erlangt hat, ist auf neue web-basierte Systeme nur bedingt übertragbar. Dies zwingt dazu, die Erfahrungsbasis immer weiter auszubauen, in der Hoffnung, es könnte vielleicht für das nächste Projekt ausreichen.

Erfahrungswerte können nur aus der Entwicklungspraxis gewonnen werden. Jede Produktentwicklung ist ein Lernprozess. Also muss es ein Release geben, das nur dem Zweck dient, Erfahrungen zu sammeln, die den nachfolgenden Releases zu Gute kommen. Dies setzt aber voraus, dass die Menschen, die jene Erfahrung gewinnen, zusammen bleiben und zwar über den ganzen Lebenszyklus des Produktes hinaus. Mit jedem

Release wachsen die Erfahrungsbasis und die Genauigkeit, mit der das nächste Release geschätzt werden kann. Ausschlaggebend ist, dass auf dem gleichen Fachgebiet mit der gleichen Technologie und der gleichen Mannschaft weitergearbeitet wird.

2.2 Keine Einigung über die angestrebten Ergebnisse

Auch der angestrebte Konsens bezüglich der Ziele eines Produktes, ist nur über einen Annäherungsprozess zu erreichen. Anwender müssen sich erst an das neue System gewöhnen, ehe sie darüber ein endgültiges Urteil abgeben können. Es fällt ihnen schwer, einem abstrakten Modell zuzustimmen, sei es noch so eindringlich präsentiert. Sie brauchen einen lauffähigen Prototyp, mit dem sie sich auseinandersetzen können. Das bedeutet, erst mit Hilfe einer Vorabversion ist ein Konsens über die endgültige Version wirklich möglich.

Die Entwicklung eines funktionsfähigen Prototyps gehört daher als erste Stufe zum Lebenszyklus eines jeden komplexen IT-Produktes, um erstens die Anforderungen der Anwender herauszulocken und zweitens eine Einigung darüber herbeizuführen, wie die Anforderungen mit der Technologie umzusetzen sind. Zu diesem Zweck ist ein eigenes Projekt – ein Prototypbau – unabdingbar.

2.3 Ständig veränderte Anforderungen

Das dritte große Projektrisiko – *creeping requirements* – ist eigentlich eine Folge des zweiten. Solange die potentiellen Anwender keine endgültige Meinung über ihre eigentlichen Anforderungen haben, werden sie ihre Meinung ändern und Neues anfordern. Das Ganze ist auch für sie ein Lernprozess, wofür sie genügend Zeit brauchen. Durch die stufenweise Entwicklung eines IT-Systems haben sie die Möglichkeit, neue Anforderungen einzubringen, ohne auf die Nutzung der bisher realisierten Anforderungen zu verzichten. Da das System offen bleibt, kann es im Sinne von Open Source immer neue Funktionen aufnehmen. Einen Redaktionsschluss gibt es nur für das jeweilige Release.

2.4 Überschätzung der eigenen Produktivität

Die Überschätzung der eigenen Produktivität liegt daran, dass die Projektbeteiligten noch keine Erfahrung mit der vorgesehenen Anwendung in der vorgeschriebenen technologischen Umgebung haben. Sie müssen ihre Produktivität erst entdecken. Erst durch die Messung der Produktivität in den ersten Evolutionsphasen haben die Produktmanager die Daten, die sie brauchen, um die Produktivität in den Folgephasen abzuschätzen. Mit jedem zusätzlichen Release werden die Produktivitätsdaten zu diesem Produkt immer mehr erhärtet und die Schätzgenauigkeit immer besser. Die Team Performance lässt sich immer besser von Release zu Release hochrechnen [PlPo12].

Es bleibt also am Ende nur das fünfte Risiko – der Verlust von Teammitgliedern – übrig. Die restlichen Risiken können über die Einführung des Produktmanagements erheblich gemildert werden. Sie sind alle Folgen eines falsch verstandenen Projektbegriffes mit

starren Terminen und fest vereinbarten Leistungsumfängen in einer Welt, in der es auf Flexibilität und Anpassungsfähigkeit ankommt. Es mag zwar andere Bereiche geben, in denen dies unumgänglich ist, vor allem in der Prozesssteuerung und bei der Entwicklung von Gerätesoftware, aber im Bereich sozio-ökonomischer Systeme sind sie fehl am Platz. Die meisten betriebswirtschaftlichen Systeme sind das Produkt eines langen Evolutionsprozesses. Sie entstehen nicht durch eine einzige, einmalige Anstrengung, sondern durch viele aufeinander aufbauende Entwicklungsschübe, die das IT-Produkt stufenweise quantitativ erweitern und qualitativ verbessern.

3 Das Wesen der Produktevolution

Eins ist klar zu stellen: IT-Produkte sind von einem völlig anderen Wesen als Software-Produkte, die in Geräte eingebaut werden oder die physikalischen Prozesse steuern. Die anderen beiden Produktarten sind statischer Natur. Sie werden einmal erstellt, getestet, ausgebessert und von da an nur geringfügig geändert. Insofern spielt die Qualität der Entwicklung eine entscheidende Rolle. Sie muss den Funktionsumfang auf Anhieb zum größten Teil abdecken. Der Funktionsumfang ist auch einigermaßen fassbar. Das trifft jedoch für einen großen Teil der IT-Systeme nicht zu. Für die, die es zutrifft – die klassischen Backoffice Systeme, bzw. operative Hintergrundprozesse – sind schon längst Standardlösungen im Einsatz. Für die meisten IT-Systeme, die es noch zu entwickeln gibt – die Frontoffice Systeme, bzw. kundennahe Vordergrundprozesse – trifft es nicht zu. Ihr Funktionsumfang ist eine Variable, die sich beliebig einstellen lässt – mal mehr, mal weniger – je nachdem, wie viel er dem Unternehmen im Moment nutzt [Wood99].

Funktionalität von IT-Produkten ist deshalb keineswegs fest. Sie liegt auf einer großen Bandbreite von einer minimal akzeptierbaren zu einer maximal wünschbaren. Welche Funktionalität zu welchem Zeitpunkt realisiert wird, ist eine Frage des Nutzwertes und der verfügbaren Ressourcen. Man kann mit einer minimalen Lösung anfangen und sich allmählich an eine optimale Lösung herantasten. Obwohl, je näher man an das Wunschziel herankommt, desto mehr rückt es in weite Ferne, denn für jedes Problem, das gelöst wird, werden zwei neue geschaffen. Von einer Erfüllung aller Wünsche kann nicht die Rede sein. Allenfalls von einem ausreichenden Zustand, mit dem sich alle Beteiligten abfinden können.

Sowohl Funktionalität als auch Qualität von IT-Produkten sind relativ, relativ zu dem, was man gerne hätte, zu dem, was gerade geläufig ist und zu dem, was man bisher gehabt hat. Darum ist es so schwer, einen Konsens über den Umfang und die Qualität eines Produktes zu finden. Das, was ausreichend ist, stellt sich erst in der Nutzung des Produktes heraus. Daher kann ein IT-Produkt nur stückweise entstehen.

Die Vorausdefinition der erforderlichen Funktionalität und Qualität ist eine reine Hypothese, die erst durch die Benutzung derselben bestätigt wird. Diese Binsenweisheit hat Gilb schon Mitte der 80er Jahre unter dem Begriff „evolutionäre Software Entwicklung“ propagiert [Gilb88]. Kurz darauf folgte Boehm mit seinem sogenannten Spiralen Modell der Produktentstehung [Boeh88]. Es ist also keineswegs neu, dass IT-Produkte das Ergebnis eines evolutionären Prozesses sind. Umso erstaunlicher ist es, dass IT-Manager

immer noch das klassische Projektmanagement als Leitbild pflegen. Das dies immer noch so ist, haben wir einem falsch verstandenen Projektbegriff zu verdanken.

Das Leitbild für das IT-Produktmanagement ist die objektorientierte Denkweise. Im Mittelpunkt steht nicht die Aktivität bzw. das Projekt, sondern das Objekt bzw. das Produkt. Releases sind mit den Methoden zu vergleichen, die einem Objekt zugeordnet sind. Sie versetzen das Produkt von dem einen Zustand in den anderen.

So gesehen sind Releases als Zustandsübergänge auf dem langen Lebensweg eines Produktes zu betrachten. Innerhalb eines einzelnen Releases finden unterschiedliche Aktivitäten statt, die parallel zu einander ablaufen:

- Fehlerbehebung,
- Änderung,
- Sanierung,
- Weiterentwicklung und
- Integration (siehe Abbildung 2).

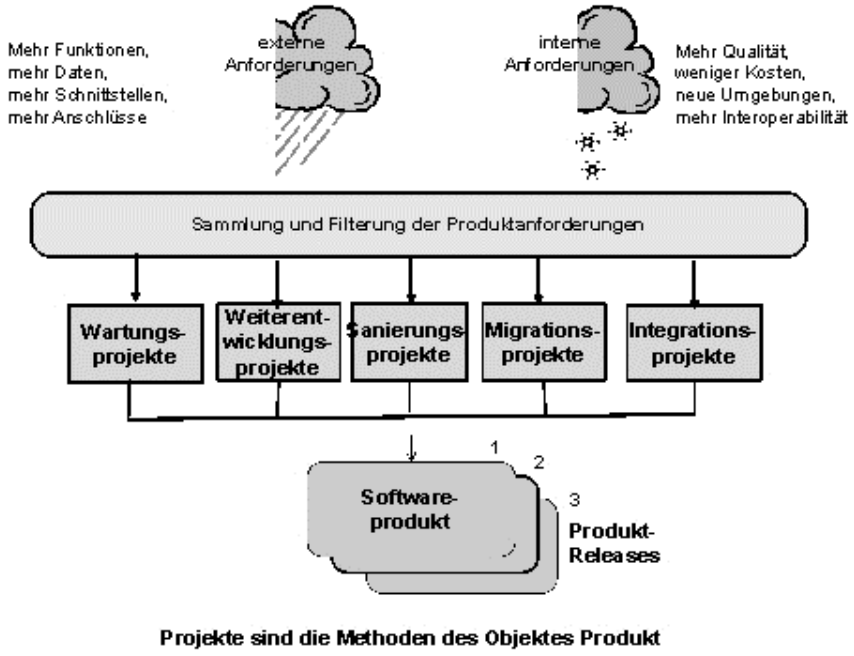

Abbildung 2: Software-Evolutionsprojekte

Fehlerbehebungsaktivitäten korrigieren das Produkt und versetzen es in einen Zustand, den es von Anfang an hätte haben sollen.

Änderungsaktivitäten verwandeln das Produkt, ohne es funktional zu erweitern. Bestehende Komponenten werden fachlich den veränderten fachlichen Anforderungen und technisch den veränderten technischen Anforderungen angepasst.

Sanierungsaktivitäten verbessern die innere Qualität des Produktes. Über diverse Reengineering, bzw. Refactoring Maßnahmen wird die Software in einen technisch besseren Zustand versetzt.

Weiterentwicklungsaktivitäten bauen die Funktionalität des Produktes aus. Es werden zusätzliche Funktionen und Daten eingefügt, die den Nutzwert des Produktes steigern.

Integrationsprojekte verbinden das Produkt mit fremden Produkten, damit es mit ihnen interagieren kann.

Die Auswirkung dieser vielen Aktivitäten auf das Produkt lässt sich am besten mittels eines Zustandsübergangsdiagramms darstellen. Der Ausgangszustand des Produktes vom letzten Release ist der Anfangszustand zum nächsten Release (siehe Abbildung 3).

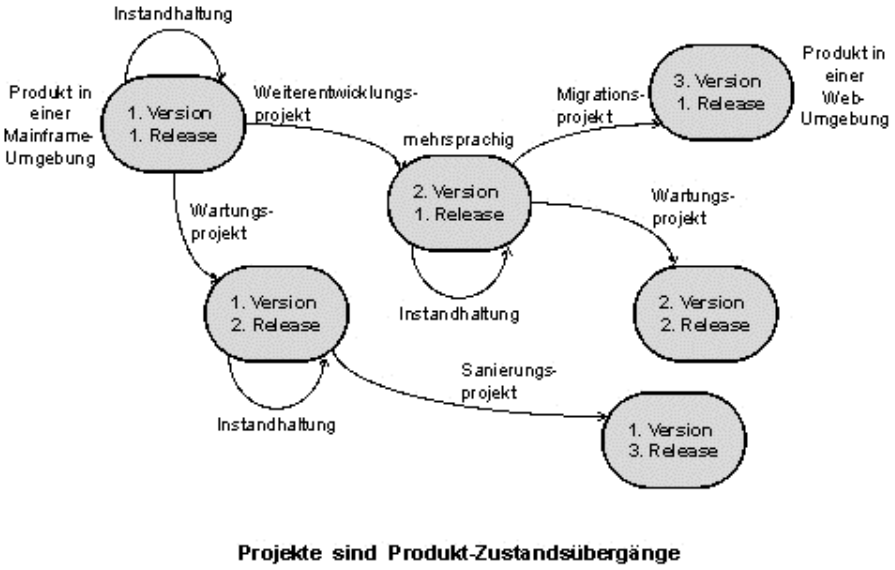

Abbildung 3: Produktzustandsübergänge

4 Planung der Software-Evolution

Die Gesetze der Software-Evolution erfordern einen gesteuerten Evolutionsprozess als Alternative zu einer willkürlichen, ad hoc Weiterentwicklung [Lehm85]. Um den Nutzwert der Software möglichst lange zu erhalten, müssen Wartung und Weiterentwicklung eine konsequente Strategie folgen. Natürlich ist es nicht möglich sämtliche Änderungen vorauszusehen und exakt zu planen. Die Evolution eines Software-Systems ist vom Wesen her stochastisch. Fehler treten unerwartet auf, notwendige Änderungen ergeben sich aus der technischen sowie aus den organisatorischen und marktwirtschaftlichen Umgebungen. Die Produkt-verantwortlichen sind gezwungen zu reagieren auf die Ereignisse, wie Gesetzesänderungen, Kundenwünsche und Konkurrenzsituationen. Sie müssen auch auf technische Erneuerungen in der Hardware und Software reagieren und ihr System anpassen ob sie wollen oder nicht. Sie stehen unter Zugzwang [CMKC03].

Man wäre geneigt zu sagen, es hat keinen Sinn irgendwas zu planen. Die zuständigen Entwickler sollten sich nur zurücklehnen und warten, bis etwas passiert. In dem Fall passte wirklich der Begriff „Wartungsmannschaft“ zu ihnen. Sie warten auf den nächsten Wartungsauftrag. Diese passive, auftragsgetriebene Haltung ist jedoch ein Hauptgrund für den Verfall eines Software-Systems. Damit treten die Gesetze der Software-Evolution in Kraft und das System verliert ständig an Wert. Um dies zu verhindern, müssen die Systemverantwortlichen eine aktive Strategie verfolgen. Die unerwarteten Ereignisse sollten in einen Planungsrahmen hineingezwungen werden. Es werden hierfür keine Kapazitäten vorgesehen, aber der Großteil der Kapazität wird für Weiterentwicklungs- und Sanierungsaufgaben geplant. Sie können, falls erforderlich, davon abgezogen werden aber sie sind zunächst verplant. Die Kunst der Evolutionsplanung liegt darin, das nicht Planbare zu planen.

Der Evolutionsprozess soll zunächst in Release-Intervalle aufgeteilt werden. Neue Releases werden in Intervallen von einem Monat bis zu einem Jahr geplant. Ein neues Release umfasst ein lauffähiges, produktionsreifes System und die dazu gehörige Dokumentation und Testumgebung. Es sollte möglich sein, die Dokumentation mit der Software abzugleichen und den Test der Software jederzeit zu wiederholen. Jedes Release hat diese Mindestkriterien zu erfüllen. Dokumentation, die mit der Produktentwicklung nicht Schritt hält, ist wegzulassen. Es werden nur jene Dokumente fortgeführt, die mit der Software im Einklang stehen. Der Test muss ebenfalls im Gleichschritt mit dem Code fortgeschrieben werden, d. h. Testfälle und Testprozeduren sind ein Spiegelbild des aktuellen Systems [MNS01].

Jedes neue Release ist als Projekt zu betrachten, auch wenn es auf einen Monat beschränkt ist. Es soll dafür zumindest einen groben Plan mit messbaren Planungszielen geben. Es gilt, die Einhaltung der Planungsziele zu kontrollieren, auch wenn sie aus welchen Gründen auch immer nicht einzuhalten sind. Es ist besser, nicht erreichbare Ziele zu haben als gar keine. Die Ziele sollten sowohl qualitativer als auch quantitativer Natur sein. Auf der einen Seite soll die Funktionalität wachsen, auf der anderen Seite die Qualität steigen. Beide Zielarten sind stets im Auge zu behalten. Ohne Evolutionsplan ist es nicht möglich, den Fortschritt der Produktevolution zu verfolgen. Daher müssen die

Evolutionsaktivitäten geplant werden, auch wenn der Plan durch die letzten Ereignisse überholt wird.

5 Ansätze zur Software-Evolution

Ein Problem der Software-Evolution ist, die Beschreibung eines Systems mit dem System selbst synchron zu halten, bzw. den Code mit dem Modell zu synchronisieren, insbesondere dann, wenn das System sich oft und signifikant ändert. Hier werden vier Ansätze zur Lösung dieses Problems geschildert:

- der anforderungsgetriebene Ansatz
- der Top-Down modellgetriebene Ansatz
- der Bottom-Up modellgetriebene Ansatz
- der testgetriebene Ansatz.

5.1 Anforderungsgetriebener Ansatz

Nach dem anforderungsgetriebenen Ansatz zur Software-Evolution bildet das Anforderungsdokument die Basis, auf der das Produkt weiterentwickelt wird. Als Erstes wird das Anforderungsdokument geändert, bzw. erweitert, dann der Code. Jede Anforderung verweist auf den Anwendungsfall, der diese Anforderung erfüllt, und jeder Anwendungsfall verweist auf die Codekomponente, bzw. auf die Services, die diesen Anwendungsfall implementieren. Es bestehen also Links zwischen dem Anforderungsdokument und dem Code. Die Aufrechterhaltung dieser Links ist eine der wichtigsten Aufgaben der Evolution – sie dürfen nicht verloren gehen. Gepflegt werden diese Links über eine System-Repository.

Der Vorteil des anforderungsgetriebenen Ansatzes ist, dass die Anforderungen in natürlicher Sprache beschrieben sind und somit für den Endanwender sowie für den Systemanalytiker verständlich sind. Sie können die Änderungen verfolgen und selbst das Anforderungsdokument fortschreiben. Sie brauchen nur ein Tool, um die Verbindungen zum Code zu erhalten. Eine zusätzliche Modellierungsprache wird nicht benötigt, weil die natürliche Sprache genügt, die Anforderungen und Anwendungsfälle zu beschreiben. Falls das Wartungspersonal einen Überblick über den Code in graphischer Form haben möchte, kann es ein Reverse-Engineering Werkzeug verwenden, um diesen bei Bedarf zu erstellen. Es ist sicher; die Anwender, Manager und Tester werden nicht danach fragen. Sie werden bei ihren Beschreibungen in natürlicher Sprache bleiben. Der Wartungsmannschaft ist in der Regel am besten gedient mit einem Software-Repository und einem flexiblen Abfragedienst, der sie nach Bedarf mit Information versorgt. Keine Studie bezüglich Software-Wartung hat jemals bestätigt, dass UML Diagramme die Wartungskosten wirklich reduzieren. Warum sollte man sie also pflegen [MuNi05]?

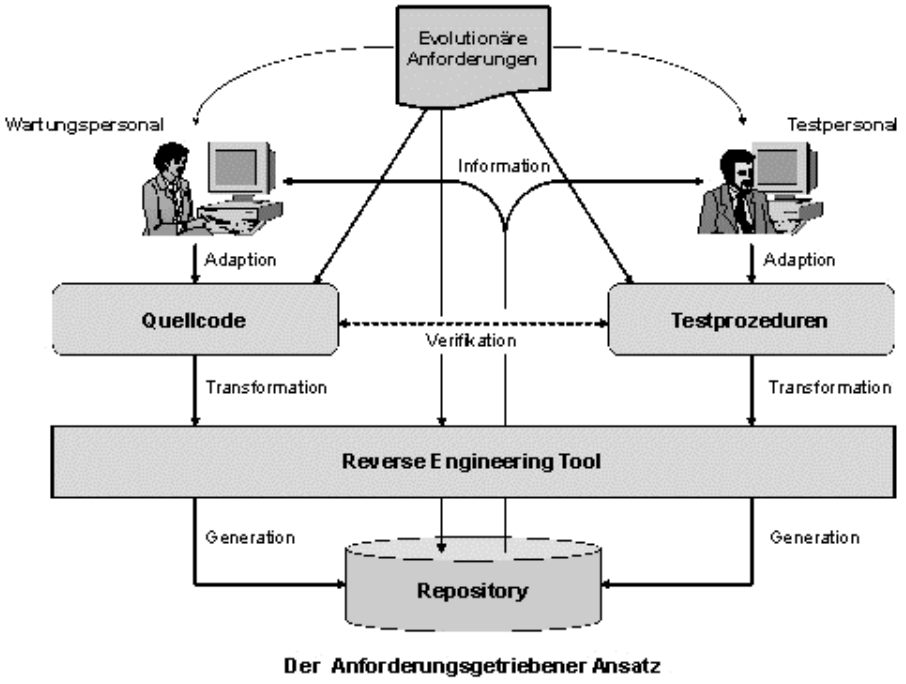

Abbildung 4: Anforderungsgetriebene Evolution

In einem anforderungsgetriebenen Modell werden insgesamt drei getrennte Beschreibungen des Zielsystems nebeneinander existieren

- Das Anforderungsdokument
- Der Source Code und
- Die Testprozeduren (siehe Abbildung 4).

Bei jeder Änderung bzw. Erweiterung wird sowohl der Source Code als auch die Testprozedur angepasst. Wichtig ist, dass diese Anpassungen von zwei verschiedenen Personen mit zwei unterschiedlichen Sichten auf das System ausgeführt werden – dem Programmierer und dem Tester. Auf der Seite des Codes wird der Programmierer die angeforderten Änderungen von den Change-Requests auf die Code-Komponente übertragen. Auf der Seite des Tests wird der Tester bestehende Testfälle ändern und neue Testfälle einfügen, um den Änderungsanforderungen gerecht zu werden. Alle beide – der Programmierer wie auch der Tester – werden Informationen über die Auswirkung ihrer Änderungen benötigen. Zu diesem Zweck wird ein unsichtbares Modell der Anwendung in einem Software Repository mit sämtlichen Systemelementen und deren Beziehungen abgebildet. Eingefügt werden die Elemente und Beziehungen durch eine automatisierte statische Source Code Analyse sowie durch eine automatisierte Analyse der Anforde-

rungstexte und der Testfälle. Durch Querverweise werden die Change-Requests mit den entsprechenden Anforderungselementen und diese wiederum mit den entsprechenden Code- und Testelementen gekoppelt. Auf diese Weise werden Änderungsanforderungen direkt oder indirekt mit den betroffenen Software Elementen verbunden. Über eine Abfragesprache können Programmierer und Tester die Abhängigkeitspfade durch die Repository verfolgen [HDS05].

Im Bereich der Softwarewartung und -weiterentwicklung sind die Tester die Vertreter der Anwender. Es ist ihre Aufgabe, dafür zu sorgen, dass die Anwender das bekommen, was sie verlangen. Zu diesem Zweck arbeitet die Testmannschaft als Gegenpol zur Wartungsmannschaft. Beide holen jedoch ihre Informationen über das System aus der gleichen Quelle – aus der Repository – und zwar über den gleichen Abfragedienst. Ihre Suchfragen sollten möglichst direkt beantwortet werden, ohne dass sie lange Listen von UML Diagrammen durchsuchen müssen. Sie werden auf Anhieb erfahren, wo was zu ändern ist. Dennoch wird die Änderung niemals automatisch durchgeführt. Es obliegt dem Menschen, den Code und den Test fortzuschreiben, denn nur so wird das bewährte Prinzip der doppelten Buchführung aufrechterhalten. Die Kosten werden zwar höher, aber so auch die Qualität der Software. Qualität hat schließlich ihren Preis und der Preis hier ist die Erhaltung zweierlei Systembeschreibungen.

5.2 Top-Down modellgetriebener Ansatz

Das Ziel der modellgetriebenen Software-Evolution wäre, dass die Änderungen am Modell automatisch an den echten Programmcode übertragen werden, wie in Abbildung 5 dargestellt.

Dies setzt eine automatische Transformation zwischen den abstrakten Beschreibungen und den weniger abstrakten voraus. Wird einem Modell eine Funktion hinzugefügt, so sollte sie auch an einer oder vielleicht mehreren Stellen im Code auftauchen. Die Voraussetzung für eine solche Transformation ist, dass die Modellierungssprache eng verbunden ist mit der Implementierungssprache, d.h. die abstrakte Beschreibung ist nicht viel abstrakter als die weniger abstrakte. Desto weiter sich die Modellierungssprache über dem Code erhebt, desto schwieriger und fehleranfälliger ist die Transformation [HaRu04].

Der modellgetriebene Ansatz basiert auf einem klassischen Top-Down-Ansatz der Software-Entwicklung, der an sich schon einen Trugschluss birgt. Die Fürsprecher dieses Ansatzes haben den naiven Glauben an die Fähigkeit durchschnittlicher Entwickler. Sie gehen davon aus, dass diese wüssten, was sie tun. In Wirklichkeit haben sie nicht die geringste Ahnung. Sie spielen mit einem Problem herum, bis sie eine akzeptable Lösung gefunden haben. Wie Balzer schreibt „ist die Implementierung in der Tat weniger eine Verfeinerung der ursprünglichen Spezifikation als vielmehr eine ständige Neudefinition derselben. Es gäbe zwischen Spezifikation und Implementierung eine viel größere Verflechtung als uns die allgemeine Meinung glauben lassen will...“ [BaSw82].

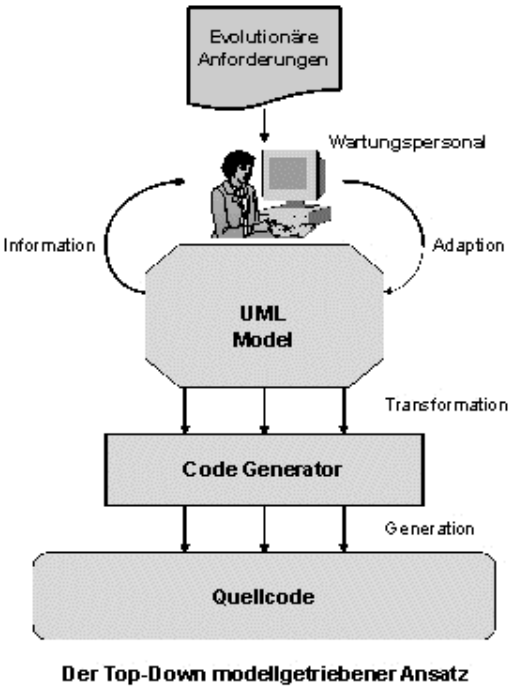

Abbildung 5: Modellgetriebene Evolution

Der Autor konnte in einem Zeitraum von beinahe 40 Jahren zahlreiche Entwickler in verschiedenen Kulturkreisen bei der Arbeit beobachten. Er tut sich schwer, jene Hypothese zu akzeptieren, nach der Entwickler mit UML Werkzeugen die zu lösenden Probleme besser verstehen, als die Entwickler vor 20 Jahren mit CASE Werkzeugen, basierend auf strukturierter Analyse und Entwurf, getan haben. Das Problem damals war der menschliche Anwender dieser Werkzeuge und er ist es immer noch. Die durchschnittlichen Entwickler in der Industrie sind nicht fähig, die Lösung zu einem komplexem Problem zu konzeptionalisieren, unabhängig davon, welche Sprache sie dabei benutzen, um sich auszudrücken oder welches Werkzeug sie für die Implementierung besitzen. Wie Michael Jackson es so treffend formulierte: „Systemanforderungen können niemals ganz im Vorfeld definiert werden, nicht einmal im Prinzip, denn der so genannte Anwender kennt sie nicht im Vorfeld, und dies nicht einmal im Prinzip“ [JaMc82].

5.3 Bottom-Up codegetriebener Ansatz

Ein entgegengesetzter Ansatz ist der Bottom-Up-Ansatz. Die Änderungen werden an der untersten semantischen Ebene der Software, nämlich am Code selbst, vorgenommen und werden nachher über diverse Reverse-Engineering Techniken auf die abstrakteren Beschreibungen übertragen. Wird also eine Schnittstelle im Code implementiert, wird diese Schnittstelle, automatisch übertragen, auch im Modell auftauchen. Dieser Ansatz ge-

währleistet, dass das Modell immer eine aktuelle und wahre Beschreibung des eigentlichen Systems ist. Jedoch muss auch hier die Implementierungssprache mit der Modellierungssprache eng verbunden sein. Alle Konstrukte der Implementierungssprache müssen über ein Äquivalent in der Modellierungssprache verfügen, anderenfalls werden sie übergangen, wie es oft der Fall ist bei der Übersetzung natürlicher Sprachen [Seli03]. Abbildung 6 beschreibt diesen Ansatz.

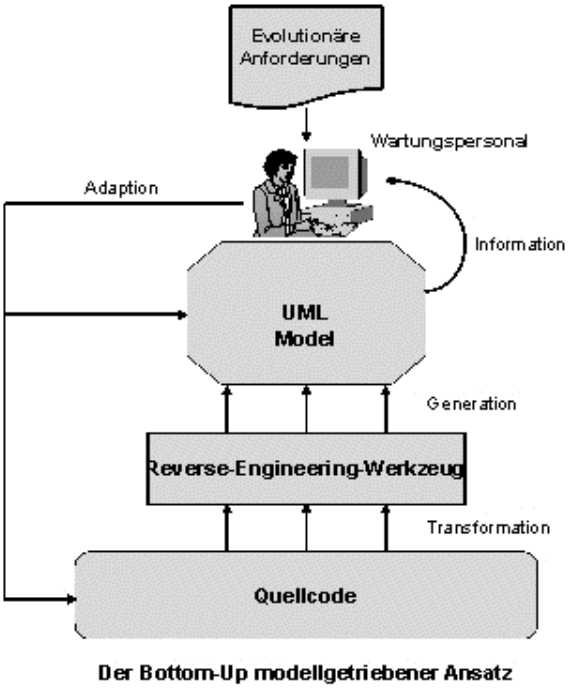

Abbildung 6: Codegetriebene Evolution

Das größte Manko der modellgetrieben Software-Evolution ist aber in beiden Ansätzen, dem Top-Down wie auch dem Bottom-Up-Ansatz, dass es nur eine einzige Beschreibung des Systems gibt. Die andere Beschreibung ist nur eine Übersetzung der ursprünglichen in eine andere Sprache. Im Fall des Top-Down-Ansatzes ist die ursprüngliche Beschreibung ein Modell. Der Code ist nur eine Kopie des Modells in einer anderen detaillierten Form. Im Falle des Bottom-Up-Ansatzes ist wiederum der Code die ursprüngliche Beschreibung und das Modell wird daraus generiert. Als solches ist das Modell nur eine weitere, etwas abstraktere Beschreibung des Codes. In beiden Fällen handelt es sich um eine und dieselbe Beschreibung des echten Systems [Sned89].

Die Frage, die sich hier stellt, ist: Was ist leichter zu ändern – die graphische, abstraktere Beschreibung oder die schriftliche, detaillierte Beschreibung? Theoretiker würden argumentieren, dass es leichter und besser wäre, die Diagramme oder abstrakteren Notatio-

nen zu ändern. Praktizierende Programmierer würden für eine Änderung im Code oder in den detaillierten Beschreibungen plädieren. Beide Gruppen haben gute Gründe für ihre Argumente. Nach 15 Jahren Forschung im Bereich des automatisierten Programmierens kamen Rich und Waters zu dem Schluss, dass „das Verfassen einer vollständigen Spezifikation in einer allgemeinen Spezifikationssprache selten leichter und oft unbeschreiblich viel schwieriger ist als ein Programm zu schreiben. Außerdem gab es nur wenige Erfolge bei der Entwicklung automatischer Generatoren, die effiziente Programme aus Spezifikationen generieren…“ [RiWa88]. Bis heute sieht der Autor keinen Grund, die Programmgeneratoren als wesentlich besser einzuschätzen.

Theoretiker werden behaupten, dass Diagramme leichter zu verstehen sind und einen besseren Überblick bieten. Praktiker werden argumentieren, dass der Code die exakteste Beschreibung von dem ist, was vor sich geht und der Code einen genaueren Einblick in die Softwarekonstruktion ermöglicht. Abgesehen davon wird der Praktiker behaupten, dass er noch nicht weiß, was passieren wird, wenn er eine Änderung macht und dass er erst einmal viele Varianten ausprobieren muss, bis er die richtige findet. Als Praktiker neigt der Autor dieses Papiers dazu, die Sicht des Programmierers zu teilen. Der Teufel liegt in den Details und in den meisten Fällen sind es die letzten 10% der Details, die den Unterschied ausmachen [Math86].

Der Top-Down-Ansatz geht davon aus, dass die Wartungsprogrammierer genau wissen, was sie tun und dass sie in der Lage sind, die Änderungen an ihren Modellen auf den darunter liegenden Code zu projizieren. Die hier zitierten Forscher wussten, dass dies nicht der Fall ist. Wartungsprogrammierer sind geborene Hacker. Wenn sie eine Änderung implementieren müssen, experimentieren sie mit mehreren Alternativen, bis sie eine passende gefunden haben. Konsequent gesehen ist Software-Evolution auf Code-Ebene ein Trial-and-Error Prozess, der so oft wiederholt wird, bis die richtige Lösung gefunden ist. Aus diesem Grund bleibt die Debatte offen, welcher Ansatz wirklich der bessere ist. Es kann vom Systemtyp abhängen, sowie von den Fachkenntnissen des Wartungspersonals [Glas04].

5.4 Testgetriebener Ansatz

Die Frage, ob Top-Down oder Bottom-Up, ist jedoch nicht das Wichtigste. Wesentlicher ist, dass beide Ansätze auf einer einzigen Beschreibung des Software-Systems basieren, da die andere Beschreibung nur eine Übersetzung ist. Dieser Fakt macht beides, modellgetriebene Entwicklung wie auch Evolution, inakzeptabel für die Verifikation und die Validierung. Um ein System zu verifizieren, also um sicher zu stellen, dass das System wahr ist, benötigt man mindestens zwei voneinander unabhängige Beschreibungen des Systems. Testen bedeutet Vergleichen. Ein System wird getestet, indem man das tatsächliche Verhalten gegen das spezifizierte abgleicht [DLP79]. Wenn der Code aus der Spezifikation abgeleitet wurde, ist er nichts anderes als die Spezifikation in einer anderen Form. Der Test des Systems ist dann nichts weiter als ein Test des Übersetzungsvorganges. Um die Qualität eines Systems sicherzustellen, ist es nötig, einen dualen Ansatz zu verfolgen, wie Abbildung 7 veranschaulicht.

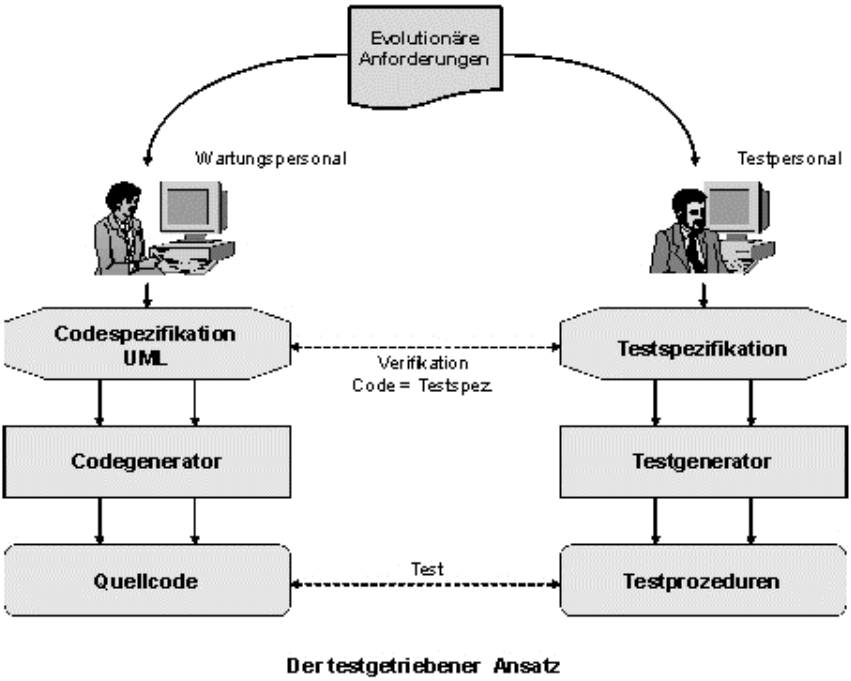

Abbildung 7: Testgetriebene Evolution

Beim Test müssen die Testfälle und Testdaten aus einer anderen Beschreibung des Systems abgeleitet werden, als aus der, von welcher der Code abgeleitet wurde. Dies bedeutet, es muss zwei unabhängige Beschreibungen der endgültigen Lösung geben, vorzugsweise in zwei verschiedenen Sprachen. Eine sollte in der Sprache der Entwickler vorliegen, die andere in der Sprache der Benutzer – und das ist die natürliche Sprache [Fetz88].

6 Die Herausforderung der Software-Evolution

Es ist die Aufgabe des Produktmanagements, ein Anwendungssystem von Release zu Release durch alle fünf Phasen des Lebenszyklus von der Entstehung bis zur Erlösung hindurch zu steuern. Der Produktmanager sorgt für die Kontinuität der Dienstleistung, die das System erbringt, bei gleichzeitig fortdauernder Weiterentwicklung. Dafür braucht er ein langfristiges Produktziel mit einem allgemeinen Evolutionsplan und eine Evolutionsstrategie, wie er dahin kommen will. Dieses Ziel darf jedoch nicht starr sein, es kann sich ändern im Laufe der Zeit. Hingegen muss es für jedes Release ein kurzfristiges und wohldefiniertes Release-Ziel geben. Dieses Ziel beinhaltet die Ergebnisse der parallel laufenden Aktivitäten – die Änderungen, Korrekturen, Erweiterungen, Sanierungen und Integrationsschritte. Nach jedem Release kann neu überlegt werden, wie das nächste Release auszusehen hat. Wichtig ist, dass das Produkt offen bleibt und sich in

jede Richtung weiterentwickeln lässt. Gleichzeitig muss jede Evolutionsstufe eine solide und nachhaltige Basis für die nächste Stufe bieten. Es komme darauf an, sowohl die Flexibilität als auch die Nachhaltigkeit des Softwareproduktes zu bewahren. Jedenfalls sollten wir uns vom dem Begriff „Projekt“ entfernen [Froh02].

Die Herausforderung des Produktmanagers besteht darin, die Erhaltung und Evolution der Anwendung von Release zu Release zu steuern. Er hat dafür zu sorgen, dass kein einziges Release weder die Leistung noch die Nachhaltigkeit des Produktes beeinträchtigt. Im Vordergrund stehen immer die Interessen der aktuellen Benutzer. Sie dürfen in ihren Arbeitsabläufen keineswegs behindert werden – und dies, obwohl das Produkt, mit dem sie arbeiten, stets verändert wird. Ein IT-Leiter verglich dieses Kunststück mit dem Umbau eines Flugzeugs während des Flugs. Es ist wirklich eine echte Herausforderung, die große technische und soziale Kompetenz voraussetzt [Rose03]. Dennoch werden nur solche Software-Anwendungen überleben, mit denen dieses Kunststück gelingt.

Literaturverzeichnis

[BaSw82] Balzert, R.; Swartout, V.: “On the inevitable intertwining of Specification and Implementation”, Comm. of ACM, Vol. 25, No. 7, July, 1982, s. 27

[BeRa00] Bennett, K.; Rajlich, V.: “Software Maintenance and Evolution – A Staged Model”, in the Future of Software Engineering, Proc. of ICSE2000, IEEE Computer Society Press, Limerick, 2000, s. 73

[Boeh88] Boehm, B.: “A Spiral model of Software Development and Enhancement”, IEEE Computer, May 1988, s. 64

[CMKC03] Cusumano, M.; MacCormack, A./ Kemerer, C./ Crandall, B.: “Software Development Worldwide – The State of the Practice”, IEEE Software, Nov. 2003, s. 28

[DeLi03] De Marco, T.; Lister, T.: “Risk Management during Requirements”, IEEE Software, Sept. 2003, s. 99

[DLP79] De Milo: Lipton; Perlis: “Social Processes and Proofs of Theorems and Programs”, Comm. Of ACM, Vol. 22, No. 5, May, 1979.

[Fetz88] Fetzer, J.: “Program Verification – The very Idea”, Comm. Of ACM, Vol. 31, No. 9, Sept. 1988

[Froh02] Fröhlich, A.: Mythos Projekt – Projekte gehören abgeschafft, Galileo Press, Bonn, 2002

[Gilb88] Gilb, T.: Principles of Software Engineering Management, Addison-Wesley Pub., Wokingham G.B., 1988, s. 83

[Glas04] Glass, R.: “Learning to distinguish a Solution from a Problem”, IEEE Software, May, 2004, s. 111

[HaRu04] Harel, D.; Rumpe, B.: “Meaningful Modelling – The Semantics of Semantics”, IEEE Computer, Oct. 2004, s. 64

[Howa01] Howard, A.: “Software Engineering Project Management”, Comm. of ACM, Vol. 44, No. 5, May 2001, s. 23

[HDS05] Hayes,J.; Dekhtyar, A./ Sundarian, S.: “Improving after the Fact Tracing and Mapping of Requirements”, IEEE Software, Dec. 2005, s.30

[Mira02] Miranda, E.: “Planning and Executing Time-Bound Projects”, IEEE Software, March 2002, s. 73

[ISO95] ISO/IEC: ISO Standard 12207 – Software Life cycle processes, International Standard Organization, Geneva, 1995

[JaMc82] Jackson, M.; McCracken, D.: “Life cycle Model considered harmful”, SE-Notes, Vol. 7, No. 1, April 1982, s. 11

[Lehm85] Lehman, M.: Program Evolution, Academic Press, London, 1985, s. 12

[Math86] Mathis, R.: “The last 10%”, IEEE Trans. On S.E., Vol. 12, No. 6, June, 1986, s. 572

[MNS01] Murphy, G., Notkin, D., Sullivan, K.: “Software Reflexion Models - Bridging the Gap between Design and Implementation”, IEEE Trans. on S.E., Vol. 27, Nr. 4, April 2001, s. 364

[MuNi05] Munson, J.; Nikora, A.: “An Approach to the measurement of Software Evolution”, Journal of Software Maintenance and Evolution, Vol. 17, No. 1, Jan. 2005, s. 65

[PlPo12] Plewan, H.J. / Poensgen, B.: Produktive Softwareentwicklung - Acht Faktoren für mehr Produktivität und Qualität, Objektspektrum, 2-2012, s. 12

[RiWa88] Rich, C.; Waters, R.: “The Programmer’s Apprentice”, IEEE Computer, Nov., 1988, s.15

[Rose03] Rosenberg, M.: „Ergo-Töchter leiden unter IT-Bereinigung“, Computerwoche, Nr. 44, Okt. 2003, s. 1

[Seli03] Selic, B.: “The Pragmatics of Model-Driven Development”, IEEE Software, Sept. 2003, s. 19

[Sned89] Sneed, H.: “The Myth of Top-Down Development and its Consequences for Software Maintenance”, Proc. Of 5th ICSM, IEEE Computer Society Press, Miami, Nov. 1989, s. 3

[Wood99] Woodward, S.: “Evolutionary Project Management“, IEEE Computer, Oct. 1999, s. 49

[ZhPa11] Zhang, Y./Patel, S.: “Agile Model-Driven Development in Practice”, IEEE Software, March 2011, s. 84