Einführung und Bewertung eines leichtgewichtigen Entwicklungsprozesses

Raimund Heid

Diplomarbeit

15. November 2002

Fachbereich Informatik
Universität Hamburg

Betreuer
Prof. Dr. Winfried Lamersdorf
(Arbeitsbereich Verteilte Systeme und Informationssysteme)
Prof. Dr. Heinz Züllighoven
(Arbeitsbereich Softwaretechnik)

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Einführung

Ein großer Teil der modernen Computer basiert auf der 1946 von *John von Neumann* vorgeschlagenen Rechnerarchitektur, die die Aufteilung eines Systems in schwer modifizierbare, mechanische Komponenten – *Hardware* – und ein leicht austauschbares Steuerprogramm – *Software* – vorsieht. Die Entwicklung von Software – oft auch als *Programmierung* bezeichnet – war jahrzehntelang einem kleinen Kreis von Spezialisten vorbehalten. In den 70er-Jahren jedoch büßte sie mit der Verbreitung von vergleichsweise kostengünstigen Rechenanlagen ihren elitären Charakter ein. Umfang und Komplexität der Programme stiegen ebenso konstant und rasch an wie die Leistungsfähigkeit der Hardware. Bald konnten die ersten Programme nicht mehr von einem Programmierer allein erstellt werden. Die Einsatzdauer vieler Systeme übertraf die ursprünglichen Annahmen bei weitem. Wie die Bestandsaufnahmen anlässlich des so genannten Jahr-2000-Problems zeigten, sind viele von ihnen seit mehreren Jahrzehnten in Betrieb.

Die steigende Komplexität der Programme und der zunehmende Bedarf an Planungssicherheit bei kommerziellen Software-Projekten führte zur Definition von so genannten *Vorgehens-Modellen,* mit denen die Arbeitsabläufe in Software-Projekten standardisiert werden sollten. Die ersten dieser Modelle orientierten sich am Vorbild von industriellen Fertigungsprozessen. Daher wurde als Synonym für *Vorgehens-Modell* die Bezeichnung *Software-Entwicklungsprozess* (oder kurz: *Prozess*) gebräuchlich.

Es stellte sich jedoch heraus, dass bei der Entwicklung von Software spezielle Gesetzmäßigkeiten gelten. Die Diskussion darüber, wie diesen am besten Rechnung getragen wird, ist noch nicht abgeschlossen und wurde in den letzten Jahren durch neue Prozesse belebt, die sich in ihren Paradigmen radikal von den traditionellen Prozessen unterscheiden. Ein hervorstechendes gemeinsames Merkmal dieser neuen Prozesse ist die vergleichsweise geringe Zahl und der schlichte Charakter ihrer Regeln. Sie werden deshalb als *leichtgewichtige* Prozesse bezeichnet.[1]

In der vorliegenden Arbeit wird untersucht, welche Anforderungen aus heu-

[1] In jüngster Zeit scheint sich die Bezeichnung *agile* Prozesse einzubürgern. Da sich dieser Wechsel erst nach Anmeldung des Titels der vorliegenden Diplomarbeit vollzog, wird hier die alte Bezeichnung *leichtgewichtig* verwendet.

tiger Sicht an einen Entwicklungsprozess für Software-Projekte mittlerer Größe gerichtet werden und inwiefern leichtgewichtige Prozesse diese erfüllen, Letzeres am Beispiel des Prozesses *Extreme Programming*.

1.2 Was ist ein Software-Entwicklungsprozess?

Wie Tom DeMarco ausführt, sind entgegen einer weit verbreiteten Annahme bei der Entwicklung von Software vornehmlich nicht *technische Probleme* sondern *Kommunikations-Probleme* zu lösen.[2] Mit Entwicklungsprozessen sollen Kommunikations-Probleme in Software-Projekten für alle Beteiligten durch Schaffung eines organisatorischen Rahmens reduziert werden.

Viele Software-Projekte wurden erfolgreich abgeschlossen, ohne dass explizit ein Entwicklungsprozess befolgt worden ist, und viele Software-Projekte sind gescheitert, obwohl ein Entwicklungsprozess befolgt wurde. Es muss deshalb die Frage gestellt werden, welchen Nutzen Software-Entwicklungsprozesse haben und wie sie zu interpretieren und anzuwenden sind.

Die Untersuchungen des *Software Engineering Institute* bei der Entwicklung des *Capability Maturity Model*[3] zeigen, dass die erfolgreiche Durchführung von Software-Projekten ohne die Befolgung eines Entwicklungsprozesses in der Regel auf das Geschick und die Erfahrung von einzelnen Personen – in der Regel Projektleitern – zurückzuführen ist. Beim Entwurf von Software-Entwicklungsprozessen werden deren Arbeitsprinzipien analysiert. Ziel ist die Definition einer abstrakten Vorlage, die als Basis für konkrete Projektpläne dienen kann.

Es gibt keine genormte Darstellungsform für Entwicklungsprozesse, gebräuchlich ist aber eine Aufgliederung in folgende Bestandteile:

- *Rollen* legen die Verantwortungsbereiche der Projektteilnehmer fest.
- *Aktivitäten* legen fest, welche Arbeitsschritte zu welchem Zeitpunkt durchzuführen sind.
- *Methoden* beschreiben, wie einzelne Arbeitsschritte durchzuführen sind.
- *Kriterien* definieren die Beschaffenheit der bei einem Arbeitsschritt zu erzeugenden Artefakte.

Entwicklungsprozesse unterscheiden sich erheblich hinsichtlich Anzahl und Ausführlichkeit der Beschreibung ihrer Rollen, Aktivitäten, Methoden und Kriterien. Aber auch ein sehr detaillierter Entwicklungsprozess beziehungsweise Projektplan kann nicht als Algorithmus aufgefasst werden, der bei korrekter Ausführung in endlicher Zeit zu einem definierten Ziel führt. Vielmehr beschreibt ein Entwicklungsprozess, wie bei der Durchführung eines Projekts *grundsätzlich* zu verfahren ist. In jedem Projekt tauchen jedoch tagtäglich sehr spezifische Probleme auf, deren Lösung nicht in einem Entwicklungsprozess beschrieben werden kann.

Bemerkenswert ist in diesem Zusammenhang die Beobachtung Bruce Cockburns, dass Entwickler und Projektleiter einen neuen Entwicklungsprozess zunächst sehr strikt befolgen, dann aber auf der Basis von Erfahrungen einzelne

[2] vgl. [?, S. 6]
[3] s. Abschnitt 3.1.2

Methoden leicht variieren und schließlich bewusst bei Bedarf neue Methoden definieren.[4] Ein Entwicklungsprozess kann demnach auch als *Anleitung zum Erlernen* von neuen Projekt-Organisationsformen verstanden werden.

Die eingangs formulierten Fragen können also folgendermaßen beantwortet werden: der Nutzen eines Software-Entwicklungsprozesses besteht darin, dass durch Befolgung von bewährten Vorgehensweisen die Planungssicherheit in einem Projekt erhöht wird. Ein Prozess beschreibt jedoch nicht *alle* Maßnahmen, die zur erfolgreichen Durchführung eines Projekts notwendig sind.

1.3 Aufbau dieser Arbeit

Die dieser Arbeit zu Grunde liegenden Fragen lauten

- „Welche Anforderungen werden aktuell an einen Software-Entwicklungsprozess für Projekte mittlerer Größe gestellt?“ und
- „Inwiefern wird der leichtgewichtige Prozess *Extreme Programming* diesen gerecht?“

Im Folgenden soll kurz erläutert werden, wie zur Beantwortung dieser Fragen vorgegangen wurde und wie die Ergebnisse der Untersuchungen dargestellt werden.

1.3.1 Klassifikation von Entwicklungsprozessen

Die leichtgewichtigen Prozesse unterscheiden sich in vielerlei Hinsicht von den ersten Prozessen, die Mitte der 50er-Jahre entwickelt wurden. Gleichwohl stehen sie gewissermaßen in deren Tradition, weil bei ihrem Entwurf die Erfahrungen aus vier Jahrzehnten organisierter Software-Entwicklung einbezogen wurden.

Es wird deshalb untersucht, wie sich die Anforderungen an Software-Entwicklungsprozesse im Laufe der Zeit verändert haben und welche weiteren Faktoren zur Entstehung von Entwicklungsprozess-Generationen beigetragen haben. Die charakteristischen Merkmale der jeweiligen Generation werden anhand von prominenten Prozessen erläutert und es wird ein einfaches Klassifikationsschema entworfen.

1.3.2 Auswahlkriterien für Entwicklungsprozesse

Die Kritik an den Entwicklungsprozessen der frühen 90er-Jahre richtete sich vornehmlich gegen deren Detailreichtum und mangelnde Flexibilität. Es stellte sich heraus, dass sie deshalb für kleine und mittlere Projekte nur bedingt geeignet waren. Die Mitte der 90er-Jahre entstandenen leichtgewichtigen Prozesse stellten daher eine Gegenreaktion dar.

Da somit heute eine Auswahl von sehr unterschiedlichen Entwicklungsprozessen zur Verfügung steht, stellt sich die Frage, welche Faktoren einen Einfluss auf die Abwicklung eines Software-Projekts haben und welche Kriterien demnach bei der Auswahl eines Entwicklungs-Prozesses beachtenswert sind. Zur Aufstellung eines Kriterienkatalogs werden zunächst zwei etablierte Modelle zur

[4] vgl. [**?**, S. 14 ff.]

Bewertung von Entwicklungsprozessen erörtert und auf die ihnen zugrunde liegenden Kriterien untersucht. Anschließend wird ein Kritienkatalog erörtert, der insbesondere die Erfahrungen mit leichtgewichtigen Prozessen berücksichtigt.

1.3.3 Das Projekt *Netlife Finance Suite*

Die für die Beurteilung des leichtgewichtigen Prozesses *Extreme Programming* herangezogenen Erfahrungswerte wurden vornehmlich in einem Industrieprojekt bei der *Netlife AG* in Hamburg gewonnen.

In diesem Kapitel werden die Ausgangssituation, die Zielsetzung und der Verlauf dieses Projekts geschildert. Insbesondere wird beschrieben, wie auf Basis der im vorangegangenen Kapitel vorgestellten Kriterien ein Entwicklungsprozess ausgewählt wurde. Schließlich erfolgt eine Beschreibung der angewandten Untersuchungsmethoden.

1.3.4 Extreme Programming in der Praxis

Extreme Programming wurde von dem Unternehmensberater *Kent Beck* entworfen und erstmals 1995 in einem Projekt angewandt. Eine große Popularität erlangte Extreme Programming jedoch erst, nachdem Kent Beck es 1999 in Buchform einer breiten Öffentlichkeit vorgestellt hatte. Seitdem wurde von vielen anderen Autoren ergänzende Literatur – Erläuterungen, Erfahrungsberichte, Verbesserungsvorschläge zu Teilaspekten, etc. – zu Extreme Programming veröffentlicht.

In diesem Kapitel werden die im Projekt *Netlife Finance Suite* gesammelten Erfahrungen mit *Extreme Programming* in Relation zu den Aussagen und Prognosen Kent Becks sowie anderer namhafter Autoren wie Martin Fowler und Ron Jeffries gebracht. Dabei werden Becks Thesen im Wesentlichen bestätigt. Es wird jedoch auch deutlich, dass die erfolgreiche Anwendung von Extreme Programming an einige essenzielle Vorbedingungen gebunden ist.

1.4 Grundbegriffe

Die vorliegende Arbeit befasst sich mit der Entwicklung von Software. Da vornehmlich auf organisatorische Aspekte und weniger auf technische Einzelheiten eingegangen wird, ist die zum Verständnis notwendige Menge an Fachbegriffen aus der Informatik vergleichsweise gering. Um Missverständnisse bei häufig verwandten Begriffen zu vermeiden, sollen die wichtigsten hier kurz definiert werden. Ein vollständiges Glossar und ein Abkürzungsverzeichnis finden sich im Anhang.

1.4.1 Technische Begriffe

Ziel jeder Software-Entwicklung ist die Konstruktion eines informationstechnischen *Systems*. Die meisten informationstechnischen Systeme basieren auf der von-Neumann-Architektur und setzen sich somit aus Hardware- und Software-Komponenten zusammen. In dieser Arbeit wird lediglich die Entwicklung von Software für standardisierte Hardware-Plattformen betrachtet. Das zu erstellende System ist also durch seine Software-Komponenten und eine Angabe der Hardware-Plattform vollständig definiert.

Nach Engesser ist ein System „eine Einheit, die durch die Zusammenfassung mehrerer Komponenten entsteht“.[5] Eine Komponente ist entweder atomar oder wiederum ein System. Die *Architektur* eines Systems legt die Beziehungen zwischen seinen Komponenten fest und ordnet jeder Komponente einen oder mehrere Verantwortungsbereiche zu. Die System-Architektur ist das Ergebnis des *System-Entwurfs* (auch „*System-Design*“). Beim System-Entwurf ist die Orientierung an einer System-Metapher[6] üblich, die eine plastische Vorstellung vom Zweck und der Struktur des Systems liefert und damit die Kommunikation zwischen Entwicklern erleichtert.

Aus Sicht eines Anwenders besteht ein System aus einer oder mehreren Programm-Dateien, die auf einer bestimmten Hardware ausgeführt werden können. Aus Effizienzgründen werden die Programm-Dateien in der Regel nicht direkt von den Software-Entwicklern erstellt, sondern auf Basis so genannter *Quelltext-Dateien* von einem Übersetzungsprogramm („*Compiler*“) erzeugt. Für den Entwickler eines Systems sind die ausführbaren Programm-Dateien von untergeordneter Bedeutung, da sie sich jederzeit ohne Aufwand aus den Quelltexten konstruieren lassen. Der Begriff *Programm* bezeichnet für sie deshalb die Quelltexte eines Systems. Diese Konvention gilt auch für die vorliegende Arbeit.

Bei der Erörterung von programmiertechnischen Details wird grundsätzlich von einem objektorientierten Entwurf und der Verwendung einer objektorientierten Programmiersprache ausgegangen. Eine Definition der elementaren Begriffe *Objekt*, *Klasse*, *Variable* und *Methode* findet sich beispielsweise bei [?, S. 484 ff.].

1.4.2 Organisatorische Begriffe

Die Entwicklung eines Systems wird von einem *Software-Haus* (auch: *Software-Firma*) durchgeführt. Sie stellt entweder eine *Produktentwicklung* für einen Massenmarkt dar oder geschieht im Auftrag eines Kunden des Software-Hauses. Da dieser Kunde das System unter Umständen wiederum seinen Kunden zur Verfügung stellt, wird zur besseren Unterscheidung der Kunde eines Software-Hauses grundsätzlich als *Auftraggeber* bezeichnet. Mit *Kunde* wird demnach immer der Kunde des Auftraggebers bezeichnet.

Die Entwicklung eines Systems geschieht im Rahmen eines zeitlich befristeten *Projekts*, in dem der Auftraggeber mit einem *Team* aus Mitarbeitern des Software-Hauses zusammen arbeitet. In der Regel benennt das Software-Haus einen seiner Mitarbeiter zum *Projektleiter*. Dieser fungiert als hauptsächlicher Ansprechpartner des Auftraggebers und koordiniert die Tätigkeiten innerhalb des Teams, das aus *Software-Entwicklern* besteht.

1.5 Danksagung

Mein Dank gilt den Professoren *Dr. Winfried Lamersdorf* und *Dr. Heinz Züllighoven* für ihre kritische Auseinandersetzung mit der vorliegenden Arbeit. Wertvolle Anregungen erhielt ich von ihren Mitarbeitern *Dr. Frank Griffel* und *Dr. Martin Lippert.*

[5] vgl. [?, S. 717]

[6] beispielsweise „doppelt verkettete Liste“

Mein Dank gilt ebenfalls der Netlife AG, Hamburg, sowie den Mitarbeitern des Netlife Finance Suite Projekts – *Albert Heck, Alexander Temmen*, *Anthony Schmude*, *Arndt-Henning Siebs, Matthias Meyer*, *Thomas Tautenhahn, Thorsten Ende*, *Patrik Uhter*, *Reinhard Letz*, und *Sa Liu* – für ihre Geduld und ihren Unternehmungsgeist.

Weiter gilt mein besonderer Dank *Andreas Jahnke*, der das Netlife Finance Suite Projekt als Abteilungsleiter kritisch begleitete und wertvolle Kommentare zu vielen Abschnitten der vorliegenden Arbeit geliefert hat. Gleiches gilt für die beiden Lektorinnen *Astrid Böhmer* und *Kirsten Laczka*.

Kapitel 2

Klassifikation von Software-Entwicklungsprozessen

Obwohl bis zum Anfang der 70er-Jahre Computerprogrammierung vielerorts als von wenigen beherrschte „Kunst“ betrieben wurde[1], stammen die ersten Vorschläge für systematische Vorgehensweisen bereits aus den 50er-Jahren. Systematik war überall dort notwendig geworden, wo

- Software in Teams entwickelt wurde
- Programme über Jahre hinweg erweitert und angepasst werden mussten
- einzelne Software-Komponenten in verschiedenen Systemen verwendet wurden
- Systeme fristgerecht zur Verfügung stehen mussten
- der Gesamtumfang des Quellcodes oder die Komplexität der Aufgabenstellung eine Modularisierung und Hierarchisierung erforderlich machten.

Die zunehmende Kommerzialisierung der Software-Entwicklung führte dazu, dass diese Bedingungen schließlich für die Mehrzahl der Software-Projekte galten. Die Standardisierung des Entwicklungsprozesses wurde damit für alle Institutionen relevant, die Software-Entwicklung in größerem Umfang betrieben. Von staatlicher Seite wurde diese Entwicklung insbesondere durch das amerikanische Verteidigungsministerium vorangetrieben.

Kategorisierungskriterien

Die im Folgenden vorgenommene Kategorisierung von Entwicklungsprozessen ist chronologisch begründet und betont die Hauptkritik der jeweils jüngeren Generation von Prozessen an ihren Vorgängern. Ein anderes Kategorisierungskriterium ist die Art des „treibenden Moments“ im Entwicklungsprozess. Die traditionellen Entwicklungsprozesse können als *dokumentengetrieben* bezeichnet werden, da sie vornehmlich Form und Inhalt der Dokumente festlegen, die

[1] [?, S. 313 ff]

in den einzelnen Prozessphasen zu erstellen sind. Dagegen können viele modernere Prozesse als *tätigkeitsgetrieben* charakterisiert werden, da sie hauptsächlich eine Menge von Aktivitäten umfassen, jedoch keine präzisen Aussagen über das zu erreichende Ergebnis einer Aktivität machen.

Skopus und Tailoring

Ein weiteres Unterscheidungsmerkmal stellt der so genannte *Skopus* dar. Er ist ein Maß für die Ausführlichkeit, mit der die Rollen, Aktivitäten und Methoden eines Prozesses beschrieben sind. Traditionelle Prozesse weisen in der Regel einen großen Skopus auf, sind also vergleichsweise ausführlich beschrieben – die Spezifikation des weiter unten vorgestellten Vorgehensmodells umfasst beispielsweise über 500 DIN-A4-Seiten. Dagegen weisen modernere Prozesse einen geringen Skopus auf, ihre wesentlichen Merkmale lassen sich häufig auf wenigen Seiten zusammenfassen.

Der Grund für diese großen Unterschiede ist, dass ältere Prozesse als *vollständige* Prozesse verstanden wurden, die alle standardisierbaren Aspekte der Software-Entwicklung abdecken sollten. Vor einem Einsatz in einem konkreten Projekt ist ein so genanntes *Tailoring* vorzunehmen, bei dem alle nicht benötigten Bestandteile des Prozesses entfernt werden. In der vorliegenden Arbeit wird diese Form des Tailoring als *Abwärts-Tailoring* bezeichnet.

Dagegen können viele moderne Prozesse als *unvollständig* bezeichnet werden, weil sie nicht alle standardisierbaren Aspekte der Software-Entwicklung ansprechen, sondern nur die nach dem Urteil ihrer Urheber *unverzichtbaren* Aspekte. Vor dem praktischen Einsatz eines derartigen Prozesses ist zu definieren, mit welchen konkreten Methoden und Werkzeugen die einzelnen Aktivitäten durchzuführen sind. Dieser Vorgang wird in der vorliegenden Arbeit als *Aufwärts-Tailoring* bezeichnet.

2.1 Lineare Prozesse

Das von Herbert D. Benington 1956 publizierte *Stagewise Model*[2] orientierte sich am Vorbild industrieller Produktionsprozesse und behandelte die Probleme, die bei unorganisierter Software-Entwicklung auftreten

- schlechte Wartbarkeit der Quelltexte
- hohe Fehlerhäufigkeit
- Systeme, die den Anforderungen der Anwender nicht gerecht werden

indem es den Prozess der Software-Entwicklung in die Phasen

1. Analyse
2. Entwurf
3. Codierung
4. Test

[2] [?]

5. Installation und Betrieb

einteilte, die streng nacheinander zu durchlaufen sind. Durch eine methodische Top-Down-Vorgehensweise sollte ein System „aus einem Guss“ entstehen.

Die konsequente Anwendung des Stagewise Model ist nur bei kleinen Systemen ratsam bzw. möglich. Sein eigentlicher Wert ist darin zu sehen, dass es einen Idealtyp von Entwicklungsprozess verkörpert, der die Basis für viele komplexere Modelle bildet. Alle Modelle, bei denen wie im Stagewise Model die obige Phasenfolge bis zur Fertigstellung des Endprodukts nur einmal durchlaufen wird, werden als *lineare Prozesse* bezeichnet.

2.1.1 Wasserfall-Modell

Das Wasserfall-Modell stellt eine Erweiterung des Stagewise Model dar. Es wurde 1970 von Walker Royce vorgestellt.[3] Die Bezeichnung „Wasserfall-Modell“, die auf den nicht-zyklischen Charakter des Modells hinweist, geht auf eine Publikation von Barry W. Boehm aus dem Jahre 1981 zurück.[4] Royce erweiterte das Stagewise Model um Rückkopplungsschleifen zwischen den Phasen. Erweisen sich die Ergebnisse einer vorangegangenen Phase als unzureichend, so wird die aktuelle Phase abgebrochen und der Prozess wieder in der vorigen Phase fortgesetzt. Dieses Vorgehen wird allerdings nicht beliebig tief rekursiv fortgesetzt, sondern bleibt auf die unmittelbar vorangegangene Phase beschränkt.

Das Wasserfall-Modell ist dokumentengetrieben. Es macht wenig Aussagen darüber, welche Maßnahmen in einer Phase durchzuführen sind, sondern definiert stattdessen Form und Inhalt der zu erstellenden Dokumente:

1. Das Ergebnis der Analyse-Phase ist ein *Anforderungskatalog*, in dem das für die Nutzer erfahrbare Verhalten des zu realisierenden Systems beschrieben wird.

2. Das Ergebnis der Entwurfsphase ist der *technische Feinentwurf*, der das zu realisierende System in Komponentendarstellung zeigt.

3. Das Ergebnis der Codierungsphase ist der *Quelltext*, der eine Umsetzung des technischen Feinentwurfs in eine Programmiersprache darstellt.

4. Das Ergebnis der Testphase ist ein *Abnahmezertifikat*. Es bescheinigt dem Hersteller, dass das System die im Anforderungskatalog definierten Eigenschaften besitzt.

5. Das Ergebnis der Installationsphase ist das beim Auftraggeber installierte *betriebsfähige System*.

Für das Wasserfall-Modell gelten in der Praxis ähnliche Einschränkungen wie für das Stagewise Model. Es ist in seiner Reinform nur dort einsetzbar, wo die Anforderungen an das System bereits in der Analyse-Phase weitgehend erkannt und festgelegt werden können.

[3] [?]
[4] [?, S. 35 ff.]

2.1.2 Vorgehensmodell[5]

Das von Boehm 1981 vorgestellte V-Modell erweitert das Wasserfall-Modell um Qualitätssicherungsmaßnahmen, die in allen Phasen durchzuführen sind. Dabei wird unterschieden zwischen *Validation* und *Verifikation*. Bei der Validation eines Produkts wird sein Nutzen für den Anwender überprüft, bei einer Verifikation die Einhaltung einer Spezifikation. Eine Validation findet in der Analyse-Phase statt, eine Verifikation in den späteren Phasen.

Der Name des Modells erklärt sich daraus, dass Boehm die Phasen des Wasserfallmodells mit den zugehörigen Validations- bzw. Verifikationsmaßnahmen grafisch in V-Form dargestellt hat.

Das bundesdeutsche Vorgehensmodell

Das V-Modell wurde für die Bundeswehr und für deutsche Behörden detailliert ausgearbeitet und 1992 als *Vorgehensmodell* im Bundesanzeiger veröffentlicht, 1997 erschien die derzeit gültige Fassung. Das Vorgehensmodell beschreibt *Vorgehensweisen* („Was ist zu tun?“), *Methoden* („Wie ist etwas zu tun?“) und *Werkzeuganforderungen* („Womit ist etwas zu tun?“) für die Erstellung von Hard- und Software sowohl für eigenständige wie auch für eingebettete Systeme.

Es umfasst die vier Submodelle

1. Systemerstellung (SE; Erstellung des Systems durch Hard- und Software-Entwicklung)
2. Qualitätssicherung (QS; Vorgabe von Qualitätsanforderungen, Prüffällen und -kriterien. Untersuchung der Produkte und Prüfung auf Einhaltung der Standards)
3. Konfigurationsmanagement (KM; Verwaltung der erzeugten Produkte)
4. Projektmanagement (PM; Planung, Kontrolle und Information der anderen Submodelle)

Produkt-Lebenszyklus

Die einzelnen Komponenten des zu konstruierenden Systems werden generell als *Produkt* bezeichnet. Ein Produkt muss zunächst nach einem bestimmten *Produktmuster* beschrieben werden und durchläuft in der Folge den Zustandszyklus

1. geplant (Folgezustand: 2)
2. in Bearbeitung (Folgezustand: 3)
3. vorgelegt (Folgezustände: 2, 4)
4. akzeptiert (Folgezustand: 2).

Beim Übergang vom Zustand *in Bearbeitung* zum Zustand *vorgelegt* verlässt ein Produkt den Entwicklungsbereich und wird an die Qualitätssicherung übergeben, die es verifiziert. Bei erfolgreicher Prüfung erhält das Produkt den Zustand *akzeptiert* und darf von nun an bis zur Planung einer neuen Produkt-Version nicht mehr verändert werden (Übergang von *akzeptiert* nach i*n Bearbeitung*).

[5] Die Darstellung folgt [Balzert 1998, S. 101 ff.].

Terminologie von Systemen

Das Vorgehensmodell strukturiert die Begriffswelt der Produktentwicklung, indem es die Bezeichnungen für den hierarchischen Aufbau von Systemen vorgibt. Ein *System* ist in *Segmente* unterteilt, die sich wiederum aus *Einheiten* zusammensetzen. Einheiten bestehen aus *Komponenten,* die wiederum aus *Modulen* und *Datenbanken* zusammengesetzt sind.

Im Submodell Systemerstellung gibt es für jede Systemebene eine Reihe von zu erstellenden Produkten (Analysen, Entwürfe, Dokumentation etc.) und zugehörigen Aktivitäten zu ihrer Validation bzw. Verifikation.

Rollen

Um eine sachgerechte Ausführung von Aktivitäten sicherzustellen, werden im Submodell Projektmanagement *Rollen* definiert, die jeweils bestimmte Erfahrungen, Kenntnisse und Fähigkeiten bei den Personen voraussetzen, die sie ausfüllen sollen. Die folgenden Rollentypen sind in jedem Submodell zu besetzen.

- Der *Manager* legt die Rahmenbedingungen für das Submodell fest und bildet die absolute Entscheidungsinstanz.
- Der *Verantwortliche* plant, steuert und kontrolliert die Aktivitäten des Submodells.
- Einer oder mehrere *Durchführende* bearbeiten die im Submodell anfallenden Aufgaben.

Die Rollen sind konkrete Ausprägungen dieser Rollentypen. Beispielsweise wird bei den Durchführenden zwischen *System-Analytikern*, *System-Designern*, *Software-Entwicklern*, *Hardware-Entwicklern* und so fort unterschieden. Insgesamt sind 25 verschiedene Rollen definiert.

Tailoring

Das Vorgehensmodell erhebt den Anspruch, für jede Art von Produktentwicklung geeignet zu sein. Um insbesondere in kleineren Projekten unnötigen Arbeitsaufwand zu vermeiden, ist eine zweistufige Anpassung (*Tailoring*) vorgesehen:

- Beim *ausschreibungsrelevanten* Tailoring werden vor dem Projektstart alle Aktivitäten gestrichen, die im konkreten Projekt nicht von Nutzen sind.
- Beim *technischen* Tailoring werden im Entwicklungsverlauf situationsabhängig Aktivitäten und Produktinhalte beim Beginn jeder Hauptaktivität festgelegt.

Alle Tailoring-Maßnahmen sind schriftlich im Projekthandbuch zu begründen. Dadurch sollen Qualitätseinbußen auf Grund von zu rigorosem Streichen von Aktivitäten vermieden werden.

Resümee

Sein linearer Charakter lässt es ratsam erscheinen, das Vorgehensmodell nur zur Konstruktion von Systemen einzusetzen, deren Eigenschaften in einem frühen Projektstadium weitgehend vollständig bestimmt werden können.

Die aktuelle Version des Vorgehensmodells enthält im Unterschied zur ersten Version aus dem Jahr 1992 auch Methoden für die inkrementelle Erstellung von Systemen. Auch wenn hierdurch in Verbindung mit der Möglichkeit des Tailoring die universelle Verwendbarkeit des Vorgehensmodells verbessert wurde, darf angesichts der großen Zahl von zu erstellenden Dokumenten und zu besetzenden Rollen nicht übersehen werden, dass die Verwendung des Vorgehensmodells einen vergleichsweise hohen Verwaltungsaufwand erfordert. Da die gesamte Spezifikation über 500 DIN-A4-Seiten umfasst, ist bereits für das Tailoring ein Zeitraum von mehreren Wochen zu veranschlagen. Das Vorgehensmodell erscheint deshalb nur sinnvoll für große Projekte mit langer Laufzeit.

2.2 Zyklische Prozesse

Bei der Anwendung von linearen Prozessen zeigte sich immer wieder, dass die in industriellen Fertigungsprozessen bewährten Vorgehensweisen nur sehr bedingt auf die Software-Entwicklung übertragen werden können. Der Grund ist vor allem die immaterielle Natur des Endprodukts Software.

Der fundamentale Unterschied zwischen Software und Produkten aus anderen Industriezweigen besteht darin, dass die Kosten für die Vervielfältigung von Software im Verhältnis zu den Entwicklungskosten gering sind. Da dies gleichermaßen für die zu Grunde liegenden Quelltexte gilt, ist Software-Entwicklung ihrem Wesen nach immer innovativ. Routinetätigkeiten wie etwa bei der Fließbandproduktion von Automobilen sind ihr fremd. Dieser Umstand erschwert insbesondere eine zuverlässige zeitliche Planung von Software-Projekten.

Kosten bei wechselnden Anforderungen

Lange Zeit waren Software-Firmen darauf bedacht, die Anforderungen an ein neu zu realisierendes System so früh und umfassend wie möglich zu ermitteln. Der Grund hierfür war die Erfahrung, dass die Kosten für eine Änderung des Systems auf Grund neuer Anforderungen mit der Projektlaufzeit bzw. mit dem Fortschreiten von Projektphase zu Projektphase exponentiell zunehmen (s. Abbildung 2.1[6]).

Aus verschiedenen Gründen lassen sich jedoch die Anforderungen an ein System in vielen Projekten nicht vollständig und nicht exakt vorab ermitteln.

- Die Entwickler müssen zum Verständnis des Anwendungsgebiets zunächst einen Lernprozess durchlaufen.
- Gleiches gilt für den Auftraggeber bzw. die Anwender in Bezug auf die eingesetzten informationstechnischen Anlagen.
- Das Marktumfeld – bedingt duch neue Mitbewerber, modische Trends, die ökonomische Situation, etc. – verändert sich im Laufe des Projekts.

[6] zitiert nach: [?, S. 8]

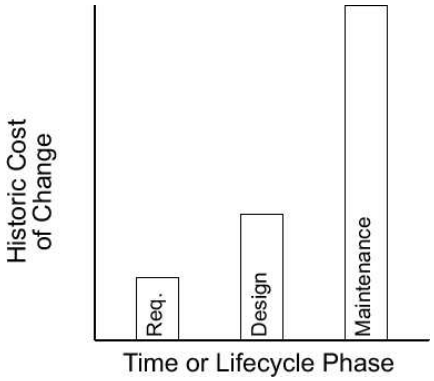

Abbildung 2.1: Historische Sicht für Kosten von Änderungen

- Notwendige Veränderungen von Arbeitsabläufen werden nicht richtig antizipiert. Die dynamische Entwicklung im Bereich der IT-Basistechnologien wie Kryptologie, Mikroelektronik, Nachrichtentechnik, Massenspeichertechnologie und dergleichen verändern die Anforderungen an ein Produkt unter Umständen während der Projektlaufzeit.
- Es gibt wechselseitige Einflüsse zwischen System und Umfeld, die nicht präzise antizipiert werden können: Das neue System verändert Arbeitsabläufe und Benutzergewohnheiten, die wiederum Änderungen des Systems sinnvoll machen, usw.
- Die Mensch-Maschine-Schnittstelle lässt sich bei neuen Systemen erst an einem Prototypen auf Handhabbarkeit prüfen.

Eine Vielzahl von Projekten scheiterte schließlich daran, dass das fertige System zwar der anfangs erstellen Spezifikation entsprach, aber dennoch nicht den realen Anforderungen des Auftraggebers gerecht wurde. Die Versuche, die Anforderungen des Auftraggebers besser zu berücksichtigen, liefen ausnahmslos darauf hinaus, die oben beschriebene Phasenfolge *Anforderungsdefinition – Entwurf – Fertigung – Test – Auslieferung* bis zur Erstellung des endgültigen Produkts – ganz oder teilweise – mehrmals zu durchlaufen. Die resultierenden Prozesse werden deshalb als *zyklische Prozesse* bezeichnet.

Lösungen zur Abflachung der Kostenkurve

Mit der Einführung von Zyklen in den Software-Entwicklungsprozess entstand die Notwendigkeit, die Kosten für nachträgliche Änderungen gering zu halten und von der Projektlaufzeit zu entkoppeln. Ziel war eine Kostenkurve wie sie in Abbildung 2.2 skizziert ist. Über die Maßnahmen, mit denen dieses Ziel erreicht werden kann, gab es im Laufe der Zeit verschiedene Ansichten. Klassische Ansätze sind:

- Komponentenbasierte Entwicklung
- Einsatz von CASE-Tools
- Stringente Programmier-Richtlinien.

Diesen Lösungen ist gemeinsam, dass sie das Ziel einer flexiblen System-Architektur durch einen gesteigerten Planungsaufwand zu erreichen versuchen. Sie eignen sich gut für die Erstellung von wiederverwendbaren Software-Komponenten („Klassen-Bibliotheken“). Bei der Konstruktion von Anwendungen macht sich jedoch der hohe zeitliche Aufwand für Entwurf und Planung negativ bemerkbar. Im Rahmen der leichtgewichtigen Prozesse wurden deshalb neue Lösungsansätze verfolgt, die weiter unten erörtert werden.

2.2.1 Spiral-Modell[7]

Das *Spiral-Modell* wurde 1986 von Barry Boehm vorgestellt.[8] Als so genanntes *Meta-Modell* enthält es nicht alle Elemente eines Entwicklungsprozesses, son-

[7] Die Darstellung folgt [Balzert 1998, S. 129 ff.].
[8] [?]

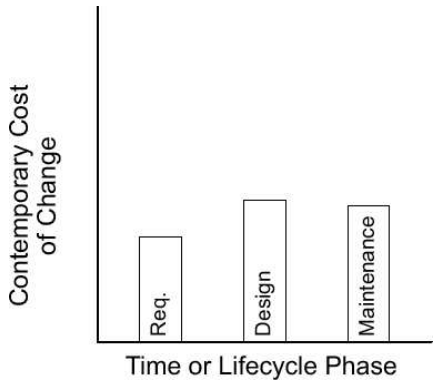

Abbildung 2.2: Aktuelle Sicht für Kosten von Änderungen

dern nur Methoden, die für das wiederholte Durchlaufen von klar definierten und voneinander abgegrenzten Entwicklungsphasen benötigt werden.

Das Spiral-Modell unterteilt einen Entwicklungszyklus in vier Schritte:

1. Die im aktuellen Zyklus angestrebten Ziele werden in Form von System-Anforderungen definiert. Anschließend werden jeweils mehrere Realisierungsmöglichkeiten inklusive der zu beachtenden Randbedingungen entwickelt.

2. Die verschiedenen Realisierungsmöglichkeiten werden evaluiert und eventuell vorhandene Risiken identifiziert. Gegebenenfalls werden zur besseren Einschätzung von Risiken Prototypen konstruiert sowie Simulationen und Benutzerbefragungen durchgeführt.

3. In Abhängigkeit von den verbleibenden Risiken wird ein Prozess-Modell für den aktuellen Zyklus ausgewählt und die Entwicklung der neuen Systemfunktionen durchgeführt.

4. Die Planung des nächsten Zyklus umfasst gegebenenfalls eine Aufteilung des Systems in Teilsysteme, die getrennt weiterentwickelt werden können, sowie eine Analyse der Schritte 1 bis 3 des aktuellen Zyklus zwecks Verbesserung des Entwicklungsprozesses.

Resümee

Das Spiral-Modell stellt eine harmonische Erweiterung für den praktischen Einsatz von linearen Entwicklungsprozessen dar, die im Rahmen des Spiral-Modells zur Durchführung einzelner Entwicklungsphasen eingesetzt werden können.

Auf Grund der ausführlichen Risikoabschätzungen und der zeitaufwändigen Evaluierung von Implementierungs-Alternativen erscheint das Spiral-Modell wenig geeignet für die Realisierung von kurzen Entwicklungszyklen, wie sie heute üblich sind. Sein Einsatzgebiet sind deshalb eher Projekte, in denen ein mittleres bis großes System mit weitgehend stabilen Anforderungen schrittweise erweitert wird.

2.2.2 Rational Unified Process[9]

Der *Rational Unified Process (RUP)* basiert auf dem *Rational Objectory Process*, der 1997 nach dem Zusammenschluss der beiden Firmen *Rational Software Corporation* und *Objectory AB* durch Verschmelzung der zwei Prozesse *Rational Approach* und *Objectory Process* entstand. Er ist eine Instanz eines allgemeineren Prozesses, den Ivar Jacobson, Grady Booch und James Rumbaugh 1999 in ihrem Buch *The Unified Software Development Process* vorgestellt haben.[10]

Ein zentrales Konzept des RUP ist die Erzeugung und Verwaltung von *Modellen*, die auf einer abstrakten Ebene die System-Architektur darstellen. Die Beschreibung der Modelle erfolgt in der *Unified Modeling Language* (*UML*), die von Rational Software entwickelt wurde, inzwischen aber von der *Object Management Group* (*OMG*) weiter gepflegt wird. Rational Software bietet verschiedene kommerzielle Produkte an, die als Werkzeuge für einzelne Arbeitsabläufe

[9] Die Darstellung folgt [?].
[10] [?]

innerhalb des RUP eingesetzt werden können. Ähnlich wie das Vorgehensmodell muss der RUP vor einem praktischen Einsatz für das jeweilige Projekt angepasst werden. Als Ausgangspunkt können dabei vorkonfigurierte RUP-Inkarnationen für verschiedene Projektgrößen und -typen verwendet werden.

Der RUP unterscheidet zwei Sichten auf ein Software-Projekt:

- Aus der Perspektive des Managements stehen finanzielle, strategische und kommerzielle Aspekte im Vordergrund.
- Aus der Perspektive der Entwickler stehen dagegen technische Aspekte wie die System-Architektur und die Qualität der erstellten Komponenten im Vordergrund.

Management-Perspektive

Aus der Perspektive des Managements läuft ein Projekt in *Zyklen* ab, die jeweils in vier Phasen unterteilt sind. Für jede Phase wird eine Liste von Eingangs- und Ausgangskriterien angegeben, die ein eindeutiges Urteil darüber erlauben, ob die Voraussetzungen für den Start bzw. den Abschluss der Phase gegeben sind.

- In der *Startphase* eines Zyklus wird eine *Vision* des zu realisierenden Systems bzw. seiner neuen Funktionen und Eigenschaften entworfen. Am Ende der Startphase muss dem Auftraggeber eine Grundlage für die Entscheidung vorliegen, ob das Projekt in die nächste Phase übergehen soll.
  In der Startphase werden folgende Dokumente erstellt:
  - eine *System-Vision*, die die wichtigsten Anforderungen, Eigenschaften und Grenzen des zu konstruierenden Systems beschreibt
  - Ein *Use-Case-Modell*, das die wichtigsten Anwendungsfälle enthält
  - Eine erste grobe *Risikoeinschätzung*
  - Ein *Projektplan*
  - Gegebenenfalls einen oder mehrere *Prototypen*.
- In der *Ausarbeitungsphase* wird der grobe System-Entwurf aus der Startphase detailliert ausgearbeitet, eine System-Architektur entworfen und eine Risikoanalyse durchgeführt. Dazu wird ein Prototyp des Systems realisiert, der die wesentlichen Eigenschaften des Zielsystems aufweist und eine provisorische Implementierung für die technischen Komponenten mit der höchsten Risikoeinstufung enthält. Am Ende der Ausarbeitungsphase muss der Auftraggeber wiederum entscheiden, ob das Projekt in die nächste Phase gehen soll.
  In der Ausarbeitungsphase entstehen folgende Dokumente und Produkte:
  - Ein weitestgehend vollständiges *Use-Case-Modell*
  - Eine Beschreibung der *System-Architektur*
  - Ein ausführbarer *Prototyp*
  - Ein detaillierter und weitgehend vollständiger *Projektplan*.

- In der *Konstruktionsphase* erfolgt schrittweise eine Implementierung der System-Komponenten, die in der Ausarbeitungsphase nur provisorisch umgesetzt worden sind. In der Regel geschieht dies parallel durch mehrere Entwickler bzw. Entwickler-Teams. Daneben erfolgt die Erstellung von Dokumentation für die Entwickler und für die System-Benutzer sowie die Definition von Testsuites.
- In der *Überführungsphase* wird die neue System-Version beim Auftraggeber installiert und in einem Testbetrieb von den Anwendern auf ihre Tauglichkeit geprüft. Die Erfahrungen der Anwender werden von den Entwicklern ausgewertet. Neben Fehlerkorrekturen werden noch geringfügige Systemänderungen vorgenommen und in regelmäßigen Abständen in Form von Updates neu installiert. Ist der Auftraggeber mit der Qualität der aktuell installierten System-Version einverstanden, wird diese anschließend regulär in Betrieb genommen.

An eine Überführungsphase schließt sich in der Regel die Startphase des nächsten Zyklus an, wobei es allerdings in der Praxis zu leichten zeitlichen Überschneidungen zwischen den Zyklen kommen kann.

Entwickler-Perspektive

Aus der Perspektive der Entwickler läuft ein Prozess in *Iterationen* ab, in denen jeweils eine neue System-Version erstellt wird, die entweder aus technischer Sicht oder aus Sicht der Benutzer einen Fortschritt gegenüber der vorigen System-Version darstellt. Bei der Projektplanung wird das Ende einer Phase immer mit dem Ende einer Iteration synchronisiert.

Statische Strukturen

Der RUP beschreibt, *was* die Projektteilnehmer *wann* und *wie* tun sollen. Daraus ergeben sich folgende statische Strukturen:

- *Rollen* beschreiben die Zuständigkeiten der Personen, denen sie zugewiesen wurden.
- *Aktivitäten* beschreiben, wie der Inhaber einer bestimmten Rolle eine bestimmte Tätigkeit durchführen soll.
- *Artefakte* beschreiben Umfang und Form von Produkten, die bei der Durchführung von Aktivitäten entstehen.
- *Arbeitsabläufe* beschreiben Folgen von Aktivitäten, die schließlich bestimmte Produkte hervorbringen.

Arbeitsabläufe

Der RUP definiert sechs *Haupt-Arbeitsabläufe*:

1. Bei der *Anwendungsgebiet-Modellierung* werden Anwendungsszenarien in Form von Use Cases beschrieben. Diese bilden in der Folge die wichtigste gemeinsame Datenbasis für den Auftraggeber und das Entwicklerteam.

2. Bei der *Anforderungsdefinition* wird das Verhalten des Systems in den einzelnen Use Cases festgelegt.

3. Bei *Analyse und Design* wird die System-Architektur entworfen bzw. angepasst. Sie bildet die Basis für die Implementierung des Systems in der Konstruktionsphase.

4. Bei der *Implementation* werden die im Analyse-und-Design-Workflow definierten Klassen implementiert und als Komponente getestet.

5. Beim *Testen* werden Regressionstests entwickelt, die die korrekte Interaktion aller Komponenten überprüfen.

6. Bei der *Inbetriebnahme* wird das System für die Installation beim Auftraggeber bzw. Kunden vorbereitet, ausgeliefert und schließlich installiert.

Daneben gibt es drei so genannte *unterstützende Arbeitsabläufe*:

1. Durch das *Projektmanagement* wird der gesamte Entwicklungsprozess kontrolliert und gesteuert.

2. Durch das *Konfigurations- und Änderungsmanagement* wird sichergestellt, dass der Konstruktionsprozess für das System reproduzierbar bleibt, und dass Änderungen systematisch verwaltet werden.

3. Das *Umgebungsmanagement* stellt zu Beginn eines Projektes sicher, dass alle benötigten Ressourcen verfügbar sind und gibt Anleitungen für die Konfigurierung des RUP.

Resümee

Der RUP ist kein Entwicklungsprozess im engeren Sinne, sondern eine Sammlung von etablierten Methoden („best practices“) in Verbindung mit einem generellen Ablaufschema für Software-Projekte. Vor einem praktischen Einsatz muss eine Auswahl von RUP-Methoden zu einer RUP-Inkarnation zusammengestellt werden. Alternativ kann je nach Projektgröße und -typ eine vorkonfigurierte RUP-Inkarnation gewählt werden – animiert durch die Popularität von Extreme Programming sind in jüngster Zeit beispielsweise auch diverse Vorschläge für leichtgewichtige RUP-Inkarnationen publiziert worden.

Die Verwendung des RUP bietet sich deshalb vor allem an für Entwicklerteams, die Projekte sehr unterschiedlicher Art und Größe durchführen. Ist diese Voraussetzung nicht gegeben, schlägt im Vergleich mit leichtgewichtigen Entwicklungsprozessen die Komplexität des RUP negativ zu Buche.

2.3 Leichtgewichtige Prozesse

Mit den zyklischen Prozessen war es möglich geworden, auch in großen Projekten anforderungskonforme Software mit relativ hoher Termintreue zu erstellen. Allerdings orientieren sich diese Prozesse in der Regel an den Anforderungen für sehr große Projekte und sind deshalb sehr umfassend und detailliert. Insbesondere für System-Analyse, Entwurf und Dokumentation ist unabhängig von der Komplexität des zu konstruierenden Systems ein hoher zeitlicher Aufwand zu veranschlagen. Zusätzlich wurde oft die Erfahrung gemacht, dass

- das Ergebnis der Entwurfsphase in der anschließenden Implementierungsphase nicht ohne gravierende Einschränkungen umgesetzt werden konnte

- aufwändig erstellte Dokumentationen der System-Architektur auf Grund von Änderungen schnell veralteten und damit wertlos wurden.

Zur Beschreibung dieser Symptomatik wurde der Begriff der *Software-Bürokratie* geprägt. Mitte der 90er-Jahre wurden Versuche unternommen, die Phänomene der Software-Bürokratie zumindest in kleineren Projekten zu vermeiden. Statt der Frage „Was *kann* standardisiert werden?“ stand die Frage „Was *muss* standardisiert werden?“ im Vordergrund. Ausgangspunkt für diesen Paradigmenwechsel war das Bestreben, etablierte Methoden zur System-Analyse, zum System-Entwurf und zur Dokumentation zu verwenden und lediglich um

Methoden zu ergänzen, die für die Steuerung von Projekten notwendig sind. Ein weiterer Vorteil dieses Ansatzes besteht in der besseren Akzeptanz durch die Entwickler und im geringen zeitlichen Aufwand für die Vermittlung von Prozessen.

Den nach diesen Prinzipien entwickelten Prozessen ist gemeinsam, dass sie nur wenige Regeln umfassen und dem Entwicklerteam große Freiheiten bei der konkreten Ausgestaltung einzelner Disziplinen wie Entwurf, Dokumentation, Qualitätssicherung etc. lassen. Sie werden deshalb als *leichtgewichtige Prozesse* bezeichnet.

Die „Leichtbauweise" dieser Prozesse beruht auf der *Kombination* von ausgewählten Methoden, die sich gegenseitig unterstützen. Es ist deshalb nicht ratsam, leichtgewichtige Prozesse inkrementell einzuführen oder einzelne Methoden nicht anzuwenden, weil damit die „Tragfähigkeit" eines Prozesses gefährdet wird.

Darüber hinaus weisen leichtgewichtige Prozesse folgende Gemeinsamkeiten auf:

- **Kurze Iterations-Zyklen**. Um möglichst häufig Rückmeldungen der Anwender zur aktuellen System-Version zu erhalten, werden kurze Iterationszyklen von wenigen Wochen oder Monaten angestrebt.
- **Orientierung am Kundennutzen**. Die während einer Iteration umzusetzenden neuen System-Funktionen und -Eigenschaften stellen aus der Sicht des Kunden bzw. System-Nutzers einen sichtbaren Mehrwert dar.
- **Adaptive Zeitplanung**. Die Zeitplanung erfolgt nicht für das Gesamtsystem, sondern bezieht sich nur auf einzelne System-Funktionen. Sie orientiert sich an der Arbeitsgeschwindigkeit des Entwicklerteams in vorangegangenen Iterationen.
- **Teamorientierung**. Die Vorteile erfolgreicher Teamarbeit – hohe Motivation der Teammitglieder, gezieltere Nutzung von individuellen Stärken, stabilere Zeitplanung bei krankheitsbedingten Ausfällen – werden gezielt genutzt.

Einige leichtgewichtige Prozesse stützen sich so sehr auf die Vorteile von Teamarbeit, dass sie nur schlecht funktionieren, wenn die Teambildung nicht gelingt. Allerdings darf nicht übersehen werden, dass ein Projekt unter diesen Voraussetzungen auch mit herkömmlichen Prozessen nur sehr mühevoll abgewickelt werden kann.

2.3.1 Extreme Programming[11]

Der Name *Extreme Programming* (*XP*) wurde von Kent Beck geprägt, der diesen Entwicklungsprozess erstmals 1995 in einem Projekt für die Chrysler Corporation verwendete und später als Buchautor vorstellte.[12] Wie durch den Namen bereits angedeutet, stellt XP die Tätigkeit des Programmierens in den Mittelpunkt der Software-Entwicklung. Alle anderen Tätigkeiten haben dagegen bei XP nur unterstützenden Charakter.

[11] Die Darstellung folgt [?].
[12] [?]

Rollen

In einem XP-Projekt werden drei Rollen unterschieden:

- Der *Auftraggeber* definiert die Anforderungen an das zu realisierende System.
- Die *Entwickler* entwerfen und implementieren das System gemäß den Anforderungen des Auftraggebers. Sie bilden ein intensiv kommunizierendes Team.
- Der *Teammanager* überwacht den Fortschritt des Projekts und kümmert sich um die Lösung unvorhergesehener Probleme.

Planung

Die Software-Entwicklung läuft in verschachtelten Zyklen ab. Ein *Releasezyklus* dauert einige wenige Monate und hat die Erstellung einer neuen System-Version zum Ziel. Während eines Release-Zyklus implementieren die Entwickler *Userstories*. Eine Userstory wird vom Auftraggeber mit Unterstützung der Entwickler formuliert und beschreibt eine Systemeigenschaft, die für den Auftraggeber einen sichtbaren Nutzen darstellt.

Die Entwickler schätzen den für die Realisierung einer Userstory erforderlichen Aufwand in so genannten *idealen Entwicklertagen*. Ein idealer Entwicklertag ist ein Arbeitstag, an dem ein Entwickler sich ohne Unterbrechung der Programmierung widmen kann. Bei der *Release-Planung* wählt der Auftraggeber eine Menge von Userstories aus, die für die nächste System-Version umgesetzt werden sollen. Diese Menge ist allerdings begrenzt durch die Anzahl der idealen Entwicklertage, die im letzten Release-Zyklus zur Verfügung standen.

Ein Release-Zyklus ist in zweiwöchige *Iterationen* unterteilt. Zu Beginn jeder Iteration unterteilen die Entwickler die für die Umsetzung einer Userstory notwendigen Arbeitsschritte in *Tasks*, die jeweils innerhalb von drei Tagen erledigt werden können. Anschließend übernimmt jeder Entwickler des Teams die Verantwortung für die Realisierung mehrerer Tasks.

System-Entwurf

Der *System-Entwurf* beginnt mit der Definition einer System-Metapher, einzelne Komponenten des Systems können anschließend im Grobentwurf mit Hilfsmitteln wie CRC-Karten beschrieben werden. Von einem Feinentwurf im klassischen Sinne wird abgesehen. Stattdessen wird sofort eine funktionsfähige Minimalversion des Systems implementiert. Die dabei entstandene System-Architektur wird im Laufe des Projekts erweitert, verfeinert und den aktuellen Anforderungen angepasst. Diese Anpassung wird als *Refactoring* bezeichnet.

Zur Risikominimierung werden bei technischen Unklarheiten im Rahmen so genannter *Spike Solutions* Prototypen erzeugt, die eine fundierte Aufwandsabschätzung erlauben.

Bei Entwurf und Implementierung des eigentlichen Systems wird konsequent das *Prinzip der Einfachheit* verfolgt, indem keine Methoden oder Attribute definiert oder implementiert werden, die für die aktuelle System-Version nicht benötigt werden. Auch eine Code-Optimierung unterbleibt, bis sie vom Auftraggeber explizit für eine bestimmte System-Version angefordert wird.

Programmierung

Entwurf und Implementierung führen die Entwickler jeweils paarweise an einem Arbeitsrechner durch. Während ein Entwickler die Tastatur bedient, achtet der andere auf die Einhaltung der Programmierkonventionen und stellt strategische Überlegungen über die nächsten Arbeitsschritte an. Diese Vorgehensweise wird *Pair Programming* genannt. Bei der Umsetzung einer Task dürfen die Entwickler alle Teile des Systems verändern, es gibt keine besonders ausgewiesenen Verwalter für einzelne Klassen oder Teilsysteme.

Ein Entwicklerpaar arbeitet jeweils an einer Task, für den einer der beiden Entwickler die Verantwortung übernommen hatte. Falls die zugehörige Userstory nicht eindeutig spezifiziert sein sollte, steht der Auftraggeber vor Ort für Rückfragen zur Verfügung.

Nach erfolgreicher Implementierung einer Task werden die Änderungen unverzüglich und in vollem Umfang in das Konfigurationsmanagement übertragen. Die neue System-Version wird regelmäßig vom Auftraggeber getestet. Gegebenenfalls ändert er im Anschluss die Anforderungen an das System.

Testen

Der Auftraggeber definiert die sich aus einer bestimmten Userstory ergebenden Anforderungen an das System in Form von so genannten *Akzeptanztests*. Akzeptanztests sind *automatisierte Tests*, d.h. sie müssen vom Tester nicht interpretiert werden, sondern machen nur eine Aussage darüber, ob ein Testlauf das erwartete Resultat geliefert hat. Die Entwicklung eines Release gilt genau dann als abgeschlossen, wenn das System alle Akzeptanztests fehlerfrei absolviert. Die Akzeptanztests dienen somit auch als Indikator für den Projektfortschritt.

Eine andere Form von automatisierten Tests wird verwendet, um die Eigenschaften einzelner Komponenten wie z.B. Klassen zu beschreiben und vor unbeabsichtigten Änderungen zu schützen. Diese Komponententests werden als *Unittests* bezeichnet und sind eine wichtige Voraussetzung für die Durchführbarkeit von Refactorings. Die Übertragung von Änderungen in das Konfigurationsmanagement darf nur erfolgen, wenn die neue System-Version alle bisherigen und alle neuen Unittests erfolgreich absolviert. Ein nachträglich gefundener Fehler wird zunächst durch einen Unittest codiert und erst anschließend behoben.

Resümee

Extreme Programming ist derzeit der populärste leichtgewichtige Prozess und wird kontrovers diskutiert. Der wichtigste Grund dürfte seine Kombinierung von sehr radikalen Methoden und Paradigmen sein:

- Pair Programming wird von manchen Entwicklern argwöhnisch beurteilt, weil sie sich dabei observiert fühlen.
- Collective Code Ownership ist eine logische Konsequenz aus der starken Teamorientierung von Extreme Programming. Es wird von manchen Entwicklern skeptisch beurteilt, weil sie ein Absinken der Software-Qualität befürchten.
- Der ausdrückliche Verzicht auf die Erstellung schriftlicher Dokumentation erweist sich zwar als praxisnah, wird aber ebenfalls oft skeptisch beurteilt.

Der erfolgreiche Einsatz von XP ist von einigen Faktoren abhängig, die in Abschnitt 4.2.1 erörtert werden. Eine ausführliche Diskussion von XP findet sich in Kapitel 5.

2.3.2 Scrum[13]

Der Name dieses Entwicklungsprozesses ist der Sportart Rugby entlehnt, bei der ein *Scrum* ein Team aus acht Mitspielern bezeichnet. Damit soll auf den Charakter von erfolgreicher Teamarbeit hingewiesen werden, bei der die Teammitglieder trotz unterschiedlicher Aufgabenbereiche ein gemeinsames Ziel verfolgen.

Entwicklungsphasen

Scrum unterteilt den Entwicklungsprozess in Phasen, die *Sprints* genannt werden und gewöhlich eine Dauer von einer bis vier Wochen haben. Vor der ersten Sprintphase entwirft ein *Chef-Architekt* eine initiale System-Architektur, die vom Team während der Sprints in Rücksprache mit dem Chef-Architekten weiterentwickelt wird.

Das Team verwaltet alle aktuell identifizierten Aufgaben in einer so genannten Rückstandsliste (*Backlog*). Die Priorität der Aufgaben wird vor jedem Sprint neu festgelegt; während eines Sprints dürfen die Aufgaben im Backlog nicht mehr vom Auftraggeber verändert werden.

Zu Beginn eines Sprints übernimmt jedes Team eine Reihe von Aufgaben aus dem Backlog und verteilt die Verantwortung für diese auf die Teammitglieder. Der Umfang wird so gewählt, dass sie alle während des Sprints abgeschlossen werden können. Das Ergebnis eines Sprints ist eine voll funktionsfähige System-Version, die für den Auftraggeber gegenüber der Vorgänger-Version einen Mehrwert bietet.

Während eines Sprints finden regelmäßig Meetings statt, auf denen alle Teammitglieder über den Stand ihrer Arbeit berichten. Die Länge eines Sprints ist festgelegt und wird auch dann nicht verändert, wenn sich abzeichnet, dass nicht alle übernommenen Aufgaben umgesetzt werden können. Stattdessen erfolgt in diesem Fall eine Reduktion des Aufgabenumfangs.

Planung und Abschätzung

In der Regel wird der Auftraggeber

- den Termin für eine neue System-Version und
- die Priorität von bestimmten System-Funktionen und -Eigenschaften

vorgeben. Es obliegt den Entwicklern, die Aufwandsabschätzung für die zugehörigen Aufgaben vorzunehmen und in Kooperation mit dem Auftraggeber den Umfang für die nächste System-Version festzulegen. Anschließend werden die Aufgaben aus dem Backlog nach Priorität auf die einzelnen Entwicklerteams verteilt. Ein elementares Risikomanagement wird dadurch realisiert, dass Aufgaben mit hohem technischem Risiko eine hohe Priorität erhalten.

[13] Die Darstellung folgt [Rising 2000].

Projektmanagement

Ein Entwicklerteam umfasst höchstens zehn Mitglieder, bei Bedarf können jedoch mehrere Teams parallel arbeiten. Ein *Scrum master* ist verantwortlich für die

- Einberufung der regelmäßigen Meetings
- Verwaltung des Backlog
- Erfassung des Projektfortschritts
- Kommunikation mit dem Auftraggeber
- Koordination der verschiedenen Entwicklerteams.

Am Ende eines Sprints wird dieser von allen Projektbeteiligten analysiert und der Auftraggeber entscheidet, ob das System weiter entwickelt werden soll oder ob die aktuelle System-Version seinen Bedürfnissen und Ansprüchen genügt.

Resümee

Scrum weist bei der Releaseplanung und beim Projektmanagement viele Gemeinsamkeiten mit XP auf. An grundlegenden Unterschieden ist zu nennen:

- Während der System-Entwurf bei XP als Gemeinschaftsaufgabe verstanden wird, erfolgt bei Scrum der initiale System-Entwurf durch einen Chef-Architekten. Vor- und Nachteile dieser Praxis werden in Abschnitt 5.7 erörtert.
- Scrum macht keine Aussage darüber, ob die Implementierung im Pair-Programming-Stil erfolgen soll.
- Der Projektablauf erfolgt bei Scrum scheinbar in einfachen Zyklen. Durch getrennt verwaltete Backlogs lässt sich jedoch ein übergeordneter Zyklus mit variabler Länge realisieren, der große Ähnlichkeit mit dem Releasezyklus von XP hat.

Obwohl die Größe eines Scrum-Entwicklerteams auf maximal zehn Mitglieder begrenzt ist, wurde Scrum bereits in Projekten eingesetzt, die eine wesentlich größere Zahl von Entwicklern erforderten. Dazu wurden die Aufgaben aus dem Backlog zu Beginn eines Sprints nach technischen Kriterien unter mehrere Scrum-Teams aufgeteilt. Es ist zu prüfen, unter welchen Voraussetzungen diese Vorgehensweise angewendet werden kann und ob sie sich auf ähnliche Entwicklungsprozesse wie zum Beispiel XP übertragen lässt.

2.3.3 Feature Driven Development[14]

Der zentrale Begriff bei *Feature Driven Development* (*FDD*) ist das *Feature*, also die System-Eigenschaft. Ein Feature ist bei FDD dadurch charakterisiert, dass es

- vom System-Nutzer als eigenständige Eigenschaft wahrnehmbar ist

[14] Die Darstellung folgt [**?**].

- mit einer Imperativ-Aussage der Form „Berechne die Summe der Rechnungsposten“ beschrieben werden kann
- von den Entwicklern innerhalb von zwei Wochen implementiert werden kann.

Features können zwecks Strukturierung zu *Feature Sets* und *Major Feature Sets* zusammengefasst werden.

Ein Projekt wird nach FDD in fünf Schritten abgewickelt. Für jeden Schritt existieren wie beim Rational Unified Process Eingangs- und Ausgangskriterien.

1. *Entwicklung eines grundlegenden Systemmodells.* Die Entwickler entwerfen unter Anleitung des Chef-Architekten ein Systemmodell in Form von Klassendiagrammen und stellen eine erste unvollständige *Feature-Liste* auf.
2. *Aufstellung einer Feature-Liste.* Die Entwickler vervollständigen die Feature-Liste, gruppieren die Features hierarchisch und ordnen jedem Feature eine Priorität zu.
3. *Projektplanung.* Der Projektmanager und die Hauptentwickler bestimmen für jedes Feature Set und jedes Major Feature Set ein vorläufiges Fertigstellungsdatum sowie einen verantwortlichen Hauptentwickler. Anschließend wird jeder Klasse ein Besitzer (*Class-Owner*) zugeordnet, der für ihre Implementation und Wartung verantwortlich ist.
4. *Entwurf.* Ein Hauptentwickler bestimmt für das aktuell zu implementierende Feature alle betroffenen Klassen und informiert die jeweiligen Class-Owner über die zu implementierenden Methoden. Anschließend entwirft das für das Feature verantwortliche Team ein Sequenzdiagramm, bei dem das aktuelle Feature im Mittelpunkt steht, und fügt es dem System-Modell hinzu.
5. *Implementierung.* Die Class-Owner implementieren die erforderlichen Methoden und erweitern die zugehörigen Komponententests. Das verantwortliche Team inspiziert den Quellcode. Nach erfolgreicher Absolvierung aller Tests werden alle Klassen im Quelltext-Versionsverwaltungs-System aktualisiert.

Die Schritte vier und fünf werden jeweils in zweiwöchigen Iterationen durchlaufen bis alle Eigenschaften implementiert sind.

Rollen und Strukturen

Feature Driven Development kennt hauptsächlich zwei Rollen und eine soziologische Struktur.

- *Hauptprogrammierer* übernehmen jeweils bis zu seiner vollständigen Umsetzung die Verantwortung für ein Feature.
- *Class-Owners* sind verantwortlich für das Design und die Implementierung einer oder mehrerer Klassen.

Die Hauptprogrammierer bilden nach Übernahme der Verantwortung für eine bestimmte Eigenschaft ein *Feature-Team*, das alle Class-Owner umfasst, deren Klassen für die Umsetzung der Eigenschaft verändert werden müssen. Die Class-Owner sind also gleichzeitig Mitglied in mehreren Feature-Teams.

Resümee

FDD unterscheidet sich in vielerlei Hinsicht grundlegend von XP und Scrum:

- Es wird ein aufwändiger initialer System-Entwurf durchgeführt.
- Es wird ein Systemmodell entwickelt, das im Laufe des Projekts immer wieder aktualisiert wird.
- Auch bei der Erweiterung des Systems wird sehr formell vorgegangen: Die Deklaration von neuen Klassen-Variablen und -Methoden erfolgt in einer anderen Projektphase als ihre Implementation.
- Das Konzept des Class-Owners steht in krassem Gegensatz zum Prinzip des Collective Code Ownership bei XP.

Feature Driven Development unterscheidet sich stärker als XP und Scrum von den informellen Vorgehensweisen in vielen Software-Firmen und dürfte deshalb schwerer einzuführen sein als diese. Auf Grund seines stringenten Charakters erscheint es insbesondere für die Konstruktion von Systemen mit mittlerer bis hoher Kritikalität[15] geeignet.

[15] s. Abschnitt 3.2.1.

Kapitel 3

Auswahlkriterien für Entwicklungsprozesse

Bei größeren mit Software-Entwicklung befassten Institutionen ist oft zu beobachten, dass der Software-Entwicklungsprozess von zentraler Stelle vorgegeben wird und nur durch ein Tailoring an die jeweiligen Projekte angepasst werden kann. Angesichts der Vielfalt der in der Softwareindustrie verwandten Entwicklungsprozesse muss jedoch angenommen werden, dass es keinen Prozess gibt, der für alle Arten von Projekten geeignet ist.

Bei erstmaliger Anwendung eines Prozesses müssen Kosten für die Auswahl und das Tailoring des Prozesses, für Mitarbeiter-Schulungen und für die fehlerhafte Anwendung einzelner Methoden veranschlagt werden. Insbesondere bei Projekten mit längerer Laufzeit sind diese jedoch geringer als die Kosten, die für überflüssige Arbeitsschritte oder für die Erstellung von nicht benötigten Dokumenten anfallen. Aus ökonomischen Überlegungen gilt es deshalb abzuwägen, mit welchem Prozess ein System unter bestimmten Qualitätsanforderungen am schnellsten entwickelt werden kann. Als weitere ökonomisch relevante Einflussfaktoren sind dabei ggf. Kosten für Wartung, Erweiterungen, Einarbeitung neuer Entwickler usw. zu berücksichtigen.

Nach diesen Überlegungen rückt die Frage in den Vordergrund, wie ein angemessener Prozess für ein Projekt bestimmt werden kann. Beim in Kapitel 4 beschriebenen Projekt *Netlife Finance Suite* wurde zu diesem Zweck ein Kriterien-Katalog zur Bewertung von Kandidaten-Prozessen definiert. Zur Aufstellung dieses Kriterien-Katalogs wurden zunächst etablierte Modelle zur Bewertung von Prozessen hinsichtlich der von ihnen verwandten Kriterien untersucht.

3.1 Etablierte Bewertungsmodelle

Die Erfahrung der Software-Firmen und ihrer Auftraggeber zeigt, dass die Qualität von komplexer Software maßgeblich vom Prozess abhängt, mit dem sie erstellt wird.[1] Deshalb entstand Mitte der 80er-Jahre bei beiden ein Interesse an Modellen zur Bewertung und Verbesserung der verwendeten Software-Entwicklungsprozesse.

[1] [Balzert 1998, S. 328]

Die Vorschläge für solche Modelle stammen von verschiedenster Seite. Das *Capability Maturity Model* (CMM) wurde im Auftrag des US-Verteidigungsministeriums von der Carnegie-Mellon-Universität entwickelt, während die *ISO-9000-Norm* und *Software Process Improvement Capability Determination* (SPICE) von der *International Standards Organisation* (ISO) stammen; *Business Engineering* wurde von den Unternehmensberatern Michael Hammer und James Champy in einem viel beachteten Buch als Methode zur Umstrukturierung von ganzen Unternehmen vorgeschlagen.

Während CMM und SPICE auf eine schrittweise Verbesserung von bestehenden Prozessen abzielen und deshalb als *evolutionäre* Modelle bezeichnet werden, sehen ISO 9000 und Business Engineering die zeitgleiche Einführung von neuen Verfahrensweisen in allen Unternehmensbereichen vor und werden deshalb als *revolutionäre* Modelle bezeichnet.[2]

In der vorliegenden Arbeit werden die ISO-9000-Norm und das Capability Maturity Model näher analysiert. Da SPICE im Wesentlichen eine Erweiterung des CMM darstellt und Business Engineering weniger auf eine Bewertung als auf eine Veränderung von Entwicklungsprozessen abzielt, wurde bei beiden von einer näheren Betrachtung abgesehen.

3.1.1 ISO 9000[3]

Im Herbst 1985 erschien von der ISO die Normenserie ISO 9000 bis 9004, die Qualitätsmanagementsysteme international vereinheitlichen sollte. Diese Normen wurden 1987 als DIN-Normen und 1987 vom CEN (*Comité Européen de Normalisation*) ohne Änderungen übernommen. Das ISO-9000-Normenwerk legt für das Auftraggeber-Lieferanten-Verhältnis einen allgemeinen organisatorischen Rahmen zur Qualitätssicherung(QS) von materiellen und immateriellen Produkten fest. Während die Normen ISO 9001 bis 9003 Modelle zur Darlegung der QS bei Prozessen mit verschiedenen Fertigungstiefen beschreiben, erläutert die ISO-9004-Norm die verwendeten QS-Elemente und gibt Hinweise für den Aufbau bzw. die Verbesserung eines Qualitätsmanagementsystems.

In der Richtlinie ISO 9000-3 wird unter Verwendung von informationstechnischer Terminologie erläutert, wie die sehr abstrakt formulierte Norm ISO 9001 bei der Software-Entwicklung anzuwenden ist. Sie besteht aus einer Reihe von Verfahrensanweisungen zur Entwicklung, Lieferung und Wartung von Software. Neben einmalig durchzuführenden (und regelmäßig zu überprüfenden) Maßnahmen gibt es solche, die bei jedem Projekt durchzuführen sind. Zu den einmalig durchzuführenden Maßnahmen gehören Maßnahmen der Geschäftsführung

- Verpflichtung zu einer Qualitätspolitik
- ständige Überwachung der Normeinhaltung durch einen Beauftragten der Geschäftsführung
- regelmäßige Überprüfung und ggf. Anpassung des Qualitätsmanagementsystems

sowie Maßnahmen der Mitarbeiter der Qualitätssicherung

[2] vgl. [Balzert 1998, S. 328]. Die ISO-9000-Norm wird in Abweichung zu Balzert aus genanntem Grund als revolutionär klassifiziert.

[3] Die Darstellung folgt [Balzert 1998, S. 328 ff].

- Festlegung von Verantwortlichkeiten und Befugnissen
- Bereitstellung von Mitteln und Mitarbeitern zur Überprüfung von Zwischenergebnissen
- Einrichtung eines Qualitätsmanagementsystems.

ISO 9000-3 schreibt kein spezielles Vorgehensmodell vor, es werden jedoch implizit folgende Annahmen gemacht:

- Die Software-Entwicklung findet in Phasen statt.
- Die Vorgaben für jede Phase sind festgelegt.
- Die geforderten Ergebnisse und durchzuführenden Verifizierungsverfahren jeder Phase sind festgelegt.

Im Laufe eines Projekts sind folgende Dokumente zu erstellen:

- Vertrag Auftraggeber-Lieferant
  - Annahmekriterien
  - Behandlung von Änderungen der Auftraggeberforderungen
  - Lieferumfang
  - Pflichten des Auftraggebers
- Spezifikation
  - vollständiger und eindeutiger Satz von funktionalen Forderungen
  - Leistung, Ausfallsicherheit, Zuverlässigkeit, Datensicherheit
  - externe Schnittstellen
- Entwicklungsplan
  - Projektziele
  - Projektmittel (Teamstruktur, Verantwortlichkeiten, Unterlieferanten)
  - Entwicklungsphasen
  - Managementpflichten (Terminüberwachung, Fortschrittsüberwachung)
  - Entwicklungsmethoden und -werkzeuge
  - Projektplan (Termine, Ressourcen)
- Qualitätssicherungsplan
  - messbare Qualitätsziele
  - Kriterien für die Entwicklungsphasen
  - Test-, Verifizierungs- und Validierungsmaßnahmen
- Testplan
  - Testarten

  - Testfälle
  - Testdaten
  - Testumgebung, Werkzeuge
  - Vollständigkeitskriterien
- Wartungsplan
  - Umfang
  - Identifikation des Produkt-Ausgangszustands
  - Wartungstätigkeiten
  - Wartungsaufzeichnungen und -berichte
- Konfigurationsmanagement-Plan
  - Verantwortlichkeiten der beteiligten Organisationen
  - Konfigurationsmanagement-Tätigkeiten
  - Werkzeuge

Beurteilung

Die ISO-9000-Norm macht keine Aussagen darüber, wie Dokumente zu erstellen oder Software-Komponenten im Detail zu konstruieren sind. Durch die Vielzahl von Pflicht-Dokumenten hat sie jedoch fast den Charakter eines dokumentengetriebenen Software-Entwicklungsprozesses. Die ursprünglich beabsichtigte Unabhängigkeit der ISO-Norm vom verwandten Software-Entwicklungsprozess kann also nur bedingt konstatiert werden.

Kriterien

Da die ISO-9000-Norm lediglich die Erstellung von Dokumenten und Maßnahmen zur Qualitätssicherung definiert, lässt sich ihr eine Liste von Kriterien zur Beurteilung von Entwicklungsprozessen nicht direkt entnehmen. Es lässt sich jedoch feststellen, dass sie die folgenden Disziplinen als Kernbereiche der Software-Entwicklung identifiziert:

- **Anforderungsmanagement**. Bestimmung der Anforderungen an das zu realisierende System; Management von Anforderungsänderungen
- **Release-Planung**. Aufstellung, Überwachung und Anpassung eines Zeitplans für die Realisierung von System-Versionen
- **Qualitätssicherung**. Definition von nachprüfbaren Qualitätskriterien
- **Testen**. Definition und Anwendung von Testkriterien
- **Konfigurationsmanagement**. Definition und Realisierung von reproduzierbaren Produktionsverfahren.

3.1.2 Capability Maturity Model[4]

Das *Capability Maturity Model* (*CMM*) wurde Mitte der 80er-Jahre vom *Software Engineering Institute* (*SEI*) im Auftrag des US-Verteidigungsministeriums entwickelt. Es dient dazu, von US-amerikanischen Behörden beauftragte Software-Firmen zu bewerten und ihnen Richtlinien für die Verbesserung ihrer Software-Entwicklungsprozesse an die Hand zu geben.

Das Capability Maturity Model unterscheidet fünf Reifegrade von Software-Entwicklungsprozessen und gibt Hinweise darauf, welche Schritte eingeleitet werden müssen, um den jeweils nächsthöheren Reifegrad zu erreichen. Je höher der Reifegrad des verwandten Software-Entwicklungsprozesses ist, umso zuverlässiger können Vorgaben für Termine, Kosten und Qualität von einem Software-Unternehmen eingehalten werden.

1. Ein *initialer Prozess* erlaubt keine zuverlässigen Aussagen über Kosten, Zeit und Qualität bei der Erstellung von Software. Zur Erreichung des zweiten Reifegrads sind in jedem Projekt folgende Aktivitäten durchzuführen:

   - Planung (Kosten- und Zeitschätzung, Terminplanung)
   - Fortschrittsüberwachung
   - Änderungsmanagement
   - Qualitätssicherung.

2. Bei einem *wiederholbaren Prozess* sind Kosten und Qualität Schwankungen ausgesetzt. Die Termin-Kontrolle funktioniert zufriedenstellend, der Erfolg von Projekten ist jedoch von einigen wenigen Mitarbeitern mit Schlüssel-Qualifikationen abhängig. Um den dritten Reifegrad zu erreichen, sollten folgende Maßnahmen durchgeführt werden:

   - Entwicklung von Prozess-Standards
   - Einführung von Methoden für Definition, Entwurf, Inspektion und Test.

3. Ein *definierter Prozess* erlaubt eine zuverlässige Kosten- und Terminkontrolle. Die Qualität der erstellten Software ist gegenüber einem Prozess des zweiten Reifegrades zwar verbessert, aber im Einzelfall schwer prognostizierbar. Folgende Maßnahmen müssen für die Erreichung des vierten Reifegrades durchgeführt werden:

   - Analyse und Vermessung der Prozesse
   - Quantitative Qualitätssicherung.

4. Ein *gesteuerter Prozess* erlaubt eine statistische Kontrolle der Qualität der erstellten Produkte. Zur Erreichung des fünften Reifegrades sind folgende Aktionen durchzuführen:

   - Erstellung quantitativer Produktivitätspläne und Produktivitätsüberwachung

[4] Die Darstellung folgt [Balzert 1998, S. 362 ff.].

- Instrumentierte Prozessumgebung
- Ökomisch fundierte Investitionen in Technologien.

5. Ein *optimierender Prozess* stellt eine quantitative Basis für andauernde Kapitalinvestitionen in die Prozess-Automatisierung und -Verbesserung zur Verfügung. Es werden kontinierlich Maßnahmen zur Prozess-Vermessung und zur Fehlervermeidung durch Prozess-Methoden durchgeführt.

Die Ermittlung des Reifegrades eines Prozesses geschieht anhand eines Fragebogens, der so genannte Hauptdisziplinen (*key process areas*) definiert, die zur Erreichung eines Reifegrades zufriedenstellend durch den Prozess abgedeckt sein müssen. Die Reifegrade bauen aufeinander auf, ein Prozess erreicht einen bestimmten Reifegrad also nur, wenn er alle darunterliegenden Reifegrade ebenfalls erreicht. Die jeweils zu behandelnden Hauptdisziplinen sind wie folgt definiert:

- Wiederholbarer Prozess (Reifegrad 2)

  1. Anforderungsmanagement
  2. Projektplanung
  3. Projektverfolgung und -überwachung
  4. Unterauftragsmanagement
  5. Qualitätssicherung
  6. Konfigurationsmanagement

- Definierter Prozess (Reifegrad 3)

  1. Konzentration auf Prozessorganisation
  2. Definieren von Prozessen
  3. Aufstellen eines Trainingsprogramms
  4. Integration von Software-Entwicklung und -Management
  5. Software-Produkt-Engineering
  6. Koordination aller beteiligten Gruppen
  7. Frühzeitige Fehlerbehebung

- Gesteuerter Prozess (Reifegrad 4)

  1. Quantitatives Prozessmanagement
  2. Quantitatives Qualitätsmanagement

- Optimierender Prozess (Reifegrad 5)

  1. Institutionalisierte Fehlervermeidung
  2. Innovationsmanagement
  3. Prozessverbesserungsmanagement

Beurteilung

Das Capability Maturity Model gibt sinnvolle Hinweise für die Analyse und Verbesserung von Entwicklungsprozessen. Die Einteilung von Prozessen in Reifegradstufen liefert eine Hilfe bei der Priorisierung von Maßnahmen zur Prozessverbesserung.

Bei den Kritikern des CMM gibt es einen Konsens darüber, dass die Maßnahmen zur Erreichung des dritten Reifegrades weitgehend als sinnvoll einzustufen sind. Die Wirksamkeit der Maßnahmen zur Erreichung der vierten und der fünften Stufe wird jedoch von manchen Autoren bezweifelt.[5]

Kriterien

Auch dem CMM lassen sich auf Grund seiner Zielsetzung nicht direkt Kriterien für die Beurteilung der Eignung von bestimmten Prozessen für bestimmte Projekte entnehmen. Es lässt sich jedoch feststellen, dass bezüglich der zu absolvierenden Kerndisziplinen bei der Software-Entwicklung zwischen dem CMM der ISO-9000-Norm weitgehend Einigkeit herrscht. Dieser Liste sind ergänzend noch folgende Disziplinen zuzufügen:

- **Unterauftragsmanagement**. Maßnahmen und Kriterien für die Kooperation mit Sub-Unternehmen
- **Mitarbeiterschulung**. Vermittlung von Wissen über Aktivitäten und Methoden des verwandten Prozesses
- **Prozessadaption**. Überwachung und Korrektur der Prozess-Anwendung.

3.2 Auswahlkriterien für Prozesse

Die ISO-9000-Norm und das Capability Maturity Model erlauben die Beurteilung eines Entwicklungsprozesses nach allgemeingültigen Kriterien. Sie gehen beide davon aus, dass ein Software-Haus *einen* Software-Entwicklungsprozess für alle Projekte verwendet. Dessen Schwächen sollen identifiziert und systematisch beseitigt werden.

In vielen Software-Firmen gibt es heute jedoch weitgehend autonome Abteilungen mit unterschiedlichen Aufgabenbereichen und unterschiedlichen Arbeitskulturen. Außerdem erscheint es angesichts einer Vielzahl von etablierten, gut dokumentierten Entwicklungsprozessen nur noch in Ausnahmefällen sinnvoll, einen neuen Entwicklungsprozess zu entwerfen und bis zur Praxistauglichkeit weiterzuentwickeln.

Vielmehr liegt es nahe, zu untersuchen wie ein angemessener Entwicklungsprozess für ein konkretes Projekt bzw. eine Projektform bestimmt werden kann. Die im Folgenden erörterten Kriterien wurden zur Bestimmung eines Entwicklungsprozesses im Projekt *Netlife Finance Suite* verwendet, das in Kapitel 4 beschrieben ist. Der Maßnahmenkatalog und die Dokumentliste von ISO 9000 sowie die Hauptkriterien des Capability Maturity Model dienten dabei als Ausgangspunkt.

[5] vgl. [Balzert 1998, S. 377] und [?, S. 209 ff.]

3.2.1 Projekteigenschaften

Die im Folgenden untersuchten Kriterien beziehen sich auf die Eigenheiten des anstehenden Projekts. Alistair Cockburn betrachtet die *Größe des Entwicklerteams* und die *Kritikalität* des zu erstellenden Systems – bestimmt durch den maximalen Schaden, den ein Systemfehler verursachen kann – als die beiden wichtigsten Faktoren bei der Auswahl eines Entwicklungsprozesses für ein Software-Projekt.[6] Die Anwendung von Prozessen für größere Teams oder höhere Kritikalitäten als tatsächlich benötigt ist zwar ökonomisch nicht sinnvoll, jedoch unkrititisch bezüglich der geforderten Qualitätsmaßstäbe.

Neben der Größe des benötigten Entwicklerteams und der Kritikalität sollte bei der Auswahl eines Entwicklungsprozesses auch die voraussichtliche *Dynamik des Anforderungskatalogs* und der *Auftraggebertypus* beachtet werden.

Die voraussichtliche *Laufzeit eines Projekts* hat in der Regel einen Einfluss auf die Fluktuation der Entwickler und den relativen Anteil von stabilen System-Komponenten. Diese beiden Faktoren bedingen, dass mit zunehmender Projektlaufzeit der Stellenwert schriftlicher System-Dokumentation steigt. Da jedoch Prozesse, die die Erstellung der entsprechenden Dokumente nicht vorsehen, durch ein Aufwärts-Tailoring angepasst werden können[7], sollte der Projektlaufzeit bei der Wahl des Entwicklungsprozesses nur eine untergeordnete Rolle beigemessen werden. Auf eine Aufnahme in die Kriterienliste wurde deshalb verzichtet.

Dasselbe gilt für die *Art* des zu konstruierenden Systems. Vielfach wird angenommen, dass für die Entwicklung von *Produkten* andere Bedingungen gelten als bei der Entwicklung von maßgeschneiderten *Anwendungen*, weil bei letzteren zeitliche und inhaltliche Vorgaben des Auftraggebers beachtet werden müssen. Bei vielen „technikgetriebenen“ Entwicklungen wurden jedoch Produkte kreiert, die in mancher Hinsicht nicht den Bedürfnissen der Anwender entsprachen. Es empfiehlt sich deshalb, auch bei der Produktentwicklung die so genannte Geschäftsperspektive nicht zu vernachlässigen. Bei näherer Betrachtung erweisen sich deshalb die Unterschiede zwischen Produkt- und Anwendungsentwicklung als marginal.

Größe des Entwicklerteams

Die Zahl der erforderlichen Entwickler ist eine der wichtigsten Kenngrößen eines Projekts. Sie hängt u.a. vom Zeitplan und den Anforderungen an das zu erstellende System ab und steht deshalb oft erst nach Abschluss der Analyse-Phase fest, die bei vielen Prozessmodellen bereits Teil des eigentlichen Projekts ist. Häufig wird die Anforderungs- und Bedarfsanalyse auch bei großen Projekten von einigen wenigen Entwicklern bzw. Systemanalytikern durchgeführt. Gleiches gilt bei manchen Prozessmodellen für die Entwurfsphase. Das hier betrachtete Kriterium ist jedoch die volle Stärke des Entwicklerteams, die in der Regel spätestens in der Implementationsphase erreicht wird.

Erfahrungen und Fähigkeiten der Entwickler. Ein weiterer wichtiger Faktor für die erforderliche Größe des Entwicklerteams sind die Erfahrungen und Fähigkeiten der Entwickler. Cockburn berichtet von einem Projekt, das sechs

[6] [?, S. 162]
[7] [?, S. 168]

erfahrene Entwickler in einem bestimmten Zeitraum hätten abwickeln können. Da diese jedoch nicht verfügbar waren, ergab sich schließlich ein Bedarf an insgesamt 24 Entwicklern, von denen vier die Qualifikationskriterien für das ursprüngliche Sechser-Team erfüllten.[8]

Aufstockung des Entwicklerteams. Kommt es zu zeitlichen Verzögerungen während eines Projekts, so ist nach Brooks vor einer Aufstockung des Entwicklerteams zu bedenken, dass die Einarbeitung der neuen Entwickler die Produktivität der erfahrenen Entwickler vermindert und sich deshalb erst nach Monaten amortisiert.[9] Hinzu kommt, dass der verwendete Prozess unter Umständen nicht mehr der neuen Teamgröße angemessen ist und angepasst bzw. ausgetauscht werden muss.[10]

Organisations- und Kommunikationformen. Die Zahl der Entwickler ist ausschlaggebend für die möglichen Organisations- und Kommunikationsformen des Entwicklerteams. Der Erfolg der leichtgewichtigen Prozesse ist zum Teil auf die sehr effiziente Kommunikation und Arbeitsweise in überschaubaren, homogenen Teams ohne feste Rollenverteilung zurückzuführen. Diese Organisationsform kommt jedoch für größere Teams wegen der exponentiell ansteigenden Zahl an direkten Kommunikationswegen nicht in Betracht. Alternative Organisationsformen wie Hierarchien mit Gruppen- und Teamleitern oder die Gruppierung nach Funktionen bzw. Märkten haben jedoch diverse Nachteile:

- Bei der Verteilung der Aufgabenblöcke werden eventuell Interdependenzen übersehen, die sich negativ auf die Homogenität des Gesamtsystems auswirken.
- Die Zeitplanung hängt stark von einzelnen Entwicklern mit Spezialkenntnissen ab und ist deshalb schwieriger und unzuverlässiger.
- Da einzelne Entwickler nur für Teilsysteme verantwortlich sind, ist ihr Interesse an der Weiterentwicklung des Gesamtsystems geringer.
- Durch die indirekte Übermittlung von Informationen können diese verfälscht werden; klärende Rückfragen sind oft nicht möglich oder mit zeitlichen Verzögerungen verbunden.

Maximale Größe homogener Teams. Die maximale Größe für homogene Teams liegt bei etwa zwölf bis vierzehn Mitgliedern. Dieser Erfahrungswert aus Projekten mit *Extreme Programming* als Entwicklungsprozess[11] ist in etwa doppelt so groß wie ältere Empfehlungen.[12]

Dies ist bemerkenswert, weil durch die bei Extreme Programming angewandte Methode des *Pair Programming* die Zahl der aktuell in Bearbeitung befindlichen *Tasks* in etwa der halben Teamstärke entspricht. Somit liegt die Vermutung nahe, dass nicht die Zahl der Entwickler, sondern die Zahl der gleichzeitig zu beobachtenden Informationsquellen den limitierenden Faktor für die Größe von Entwicklerteams darstellt.

[8] [?, S. 160]
[9] [?, S. 75]
[10] [?, S. 163]
[11] [?, S. 156] und [?, S. 163]
[12] [Balzert 1998, S. 79]

Aufteilung von größeren Teams. Bei zu großen Gruppen gelangen nicht mehr alle relevanten Informationen zu den einzelnen Teammitgliedern. Bei einem größeren Entwicklerstab muss deshalb eine Aufteilung in mehrere kleine Teams vorgenommen werden. Bewährt hat sich hierfür unter anderem die *Holistic Diversity Strategy,*[13] bei der Teams mit einer Stärke von zwei bis fünf Entwicklern die Verantwortung für die Realisierung einer bestimmten Funktion oder einer Gruppe von Funktionen übertragen wird. Eine ähnliche Vorgehensweise wird bei Scrum angewandt.[14]

Prozessbeurteilung. Zur Beurteilung eines bestimmten Prozesses sollte zunächst festgestellt werden, mit welchen Teamgrößen ein Prozess nachweislich erfolgreich verwendet worden ist. Das Verhältnis der tatsächlichen Entwicklerzahl TS_{akt} zur maximalen Entwicklerzahl des jeweiligen Prozesses TS_{max} gibt einen Hinweis darauf, wie gut der Prozess sich für das Projekt eignet.

- Ist TS_{akt} deutlich geringer als TS_{max} ($\frac{TS_{akt}}{TS_{max}} < 0.5$), so erscheint der Prozess zu „schwer“ für das anstehende Projekt. Ggf. ist zu erwägen, den Prozess durch ein Tailoring zu verschlanken.
- Die Bedingung $0.5 \leq \frac{TS_{akt}}{TS_{max}} \leq 1.0$ charakterisiert den Normalfall. Ist TS_{akt} geringer als TS_{max}, so sind Tailoring-Maßnahmen zu erwägen.
- Für den Bereich $1.0 < \frac{TS_{akt}}{TS_{max}} < 1.25$ ist nach Cockburn zu erwägen, den Prozess durch zusätzliche Kommunikationsmethoden und Qualitätssicherungsmaßnahmen anzureichern, statt einen Prozess zu verwenden, für den $\frac{TS_{akt}}{TS_{max}} < 1.0$ gilt.[15]

Nach Cockburn ist es empfehlenswert, im Zweifelsfall einen leichtgewichtigeren Prozess mit zusätzlichen Kommunikationsformen oder Qualitätssicherungsmaßnahmen anzureichern statt einen schwergewichtigen Prozess durch einen Tailoring-Prozess effizienter gestalten zu wollen. Er führt dazu als Argument an, dass beim Tailoring erfahrungsgemäß zu zaghaft vorgegangen wird, während andererseits die Defizite bei etwas zu „leichten“ Prozessen in der Regel rechtzeitig erkannt werden.[16]

Kritikalität des Projekts

Die Kritikalität eines Projekts wird bestimmt durch den größtmöglichen Schaden, den das im Projekt zu erstellende System bei einem Defekt verursachen kann. Cockburn definiert für die Kritikalität eine vierstufige Skala[17]:

1. *Verlust von Komfort*: Ein Fehlverhalten des Systems richtet keinen nennenswerten materiellen Schaden an.
2. *Verlust von entbehrlichen Geldsummen*: Ein Fehlverhalten des Systems führt zu begrenzten materiellen Verlusten.

[13] [?, S. 214f]
[14] vgl. S. 28.
[15] [?, S. 164]
[16] [?, S. 164]
[17] [?, S. 152]

3. *Verlust von unentbehrlichen Geldsummen*: Ein Fehlverhalten des Systems kann zum Bankrott von Einzelpersonen oder Firmen führen.

4. *Verlust von Leben*: Ein Fehlverhalten des Systems gefährdet Menschenleben.

Je nach Projektgröße können beliebige Zwischenstufen definiert werden. Bei großen Systemen ist es sinnvoll, einzelnen Subsystemen unterschiedliche Kritikalitäten zuzuordnen. In diesem Fall muss durch den Entwicklungsprozess eine der jeweiligen Kritikalität angemessene Verfahrensweise sichergestellt werden.[18]

Der Kritikalität eines Systems kann durch spezielle Maßnahmen beim Entwurf und bei der Qualitätssicherung Rechnung getragen werden. Beispielhaft seien genannt:

- Anforderungsanalyse nach SADT[19]
- Vorgabe von Programmier-Richtlinien
- Vorgabe einer bestimmten Testabdeckung
- Korrektheitsbeweise
- Statistische Auswertungen über die Einhaltung von Programmier-Richtlinien.

Die konkrete Zuordnung von erforderlichen Maßnahmen bei vorgegebener Kritikalität muss projektspezifisch erfolgen, entweder durch Auswahl eines bestimmten Entwicklungsprozesses oder durch Tailoring des ausgewählten Prozesses.

Prozessbeurteilung. Die Prüfung der Eignung eines Prozesses für die Konstruktion eines Systems bestimmter Kritikalität sollte sich vor allem daran orientieren, ob der Prozess bereits für die Konstruktion von Systemen vergleichbarer Kritikalität erfolgreich verwendet wurde. Beim Aufwärts-Tailoring eines zu „leichten“ Prozesses zwecks Anpassung an eine höhere Kritikalität ist zu bedenken, dass der Einsatz eines neu konstruierten bzw. modifizierten Prozesses ebenfalls einen Risikofaktor darstellt.

Die Identifikation geeigneter Kandidatenprozesse wird erleichtert durch Prozess-Familien wie der Crystal-Gruppe, die verschieden mächtige Prozesse für Projekte mit unterschiedlicher Kritikalität enthält.[20] Eine Alternative zur Auswahl eines vorgefertigten Prozesses stellt das Abwärts-Tailoring von universellen Prozessen nach strengen Regeln[21] und unter Orientierung an Erfahrungswerten dar, s. beispielsweise: [V-Modell 1997, Teil 3 (Handbuchsammlung). Sicherheit und Kritikalität, S. 6]

[18] [Balzert 1998, S. 295]

[19] **S**tructured **A**nalysis and **D**esign **T**echnique: ein Verfahren zur hierarchischen Analyse und graphischen Darstellung bestehender Systeme und Hilfsmittel für den Entwurf von Systemen [**?**, S. 608ff]

[20] [**?**, S. 197ff]

[21] Zum Streichen von Arbeitsschritten sind bestimmte Kriterien zu überprüfen, die Streichung ist schriftlich im Projekthandbuch zu begründen.

Dynamik des Anforderungskatalogs

Die Dynamik des Anforderungskatalogs ist ein Maß für die Zahl von neuen oder geänderten Anforderungen pro Release im Verhältnis zu den Anforderungen des vorhergehenden Release. Als Auslöser für neue oder geänderte Anforderungen kommen in Frage:

- Der Auftraggeber wünscht sich zusätzliche Funktionen.
- Bei der Anwendung werden Mängel in der Spezifikation entdeckt.
- Das neue Release eines Konkurrenzproduktes enthält wichtige neue Funktionen.
- Die Performance einer Funktion ist ungenügend und erfordert eine Neuimplementation.

Es lassen sich grob drei Stufen der Dynamik unterscheiden. Die Dynamik des Anforderungskatalogs ist voraussichtlich unter folgenden Bedingungen ...

- *niedrig*:
  - Der Auftraggeber hat klare Vorstellungen von den Eigenschaften des Systems, beispielsweise weil es sich um eine Re-Implementation handelt.
  - Das Entwicklerteam hat bereits ein vergleichbares System erfolgreich entworfen und konstruiert.
  - Das System steht nicht in Konkurrenz zu Produkten anderer Softwarehersteller.
- *mittel*:
  - Der Auftraggeber kann die Anforderungen an das System spezifizieren, hat aber noch keine Vorstellung, wie diese umgesetzt werden können.
  - Das Entwicklerteam ist mit dem Anwendungsbereich vertraut, hat aber noch kein vergleichbares System realisiert.
  - Auf dem Marktsegment des Systems ist nicht mit großen Innovationsschüben zu rechnen.
- *hoch*:
  - Der Auftraggeber kann die Anforderungen an das System nicht vollständig spezifizieren.
  - Das Entwicklerteam hat keine oder nur wenig Erfahrung auf dem Anwendungsgebiet.
  - Das System ist ein Produkt in einem dynamischen Marktumfeld mit innovativem Charakter.

Ältere Software-Entwicklungsprozesse basierten auf der Annahme, dass sich die Anforderungen an das zu realisierende System weitgehend vor der Entwurfs- und Implementierungsphase ermitteln lassen und dass die Dynamik des Anforderungskatalogs im obigen Sinne mit der Projektlaufzeit stark abnimmt. Ist dies nicht der Fall, so muss bei ihnen in Kauf genommen werden, dass grundlegende System-Änderungen sehr zeitaufwändig und risikobehaftet sind.

Seit dem Beginn der Kommerzialisierung des Internet Mitte der 90er-Jahre hat die Zahl der Projekte im Bereich des elektronischen Handels stark zugenommen. Sie weisen in der Regel wegen ihres experimentellen Charakters eine hohe Dynamik des Anforderungskatalogs auf. Hinzu kommt, dass neue Anforderungen wegen der globalen Konkurrenzsituation unter hohem Zeitdruck umgesetzt werden müssen. Die in den vergangenen Jahren entstandenen Software-Entwicklungsprozesse sind deshalb darauf ausgelegt, durch eine Kombination verschiedener Methoden die Kosten für Systemänderungen über die Projektlaufzeit konstant zu halten und eignen sich damit besser für Projekte mit hoher Anforderungskatalog-Dynamik.

Prozessbeurteilung. *Lineare Prozesse* sind nur für Projekte mit dauerhaft *niedriger* Dynamik des Anforderungskatalogs geeignet, bei Projekten mit höherer Dynamik führen sie zu großen zeitlichen Verzögerungen.

Zyklische Prozesse sind geeignet für Projekte mit *mittlerer* Dynamik. Sie besitzen gegenüber den linearen Prozessen den Vorteil, dass sie für die systematische Reaktion auf Anforderungs-Änderungen ausgelegt sind. Im Vergleich zu leichtgewichtigen Prozessen darf erwartet werden, dass in der Regel schneller eine höhrere Stabilität im Bereich der System-Architektur erreicht wird.

Leichtgewichtige Prozesse erscheinen am besten geeignet für Projekte mit *hoher* Dynamik. Sie kompensieren ihre Nachteile gegenüber den zyklischen Prozessen in Bezug auf die Dynamik der System-Architektur durch die Eigenschaft, in kurzen Zeiträumen neue nutzbringende Funktionen in das System zu integrieren.

Auftraggebertypus

Obwohl viele gescheiterte Großprojekte eine andere Sprache sprechen, ist bei vielen Entscheidungsträgern ohne explizites Wissen über neuere Entwicklungen im IT-Sektor die Auffassung anzutreffen, dass das Risiko bei großen Software-Projekten durch eine sorgfältige Planung nach dem Muster „Anforderungsanalyse – Entwurf – Implementierung“ beliebig reduziert werden kann.

Andererseits sind sich manche Auftraggeber der Probleme bewusst, die damit verbunden sind, dass sie die Anforderungen an das zu realisierende System nicht genau spezifizieren können. Sie sind deshalb unter Umständen bereit, eng mit den Software-Entwicklern zusammenzuarbeiten und auch befristete Experimente zur Evaluierung von Lösungsalternativen zu finanzieren.

Es lassen sich drei Typen von Auftraggebern unterscheiden, die jeweils sehr unterschiedliche Anforderungen an den verwendeten Entwicklungsprozess stellen:

- der *konservative* Auftraggeber
  - kennt die Anforderungen an das zu realisierende System sehr genau und vollständig

  - möchte unter Umständen Grob- und Feinentwurf des Systems erläutert bekommen und beide kommentieren
  - hat wenig Interesse an einer permanenten Kooperation mit dem Entwicklerteam. Er entfaltet seine Haupttätigkeit in der Entwurfs- und Testphase.

- der *indifferente* Auftraggeber
  - kennt und benennt die wesentlichen Anforderungen an das System und überlässt dem Entwicklerteam die Ausarbeitung der Details
  - hat wenig oder kein Interesse an Implementierungsdetails
  - ist zur begrenzten Zusammenarbeit mit dem Entwicklerteam im Rahmen der Evaluierung von Prototypen bereit.

- der *kooperative* Auftraggeber
  - kann die wesentlichen Anforderungen an das System benennen und erarbeitet den Rest in enger Kooperation mit dem Entwicklerteam
  - hat kein Interesse an Implementierungsdetails
  - ist zur permanenten Zusammenarbeit mit dem Entwicklerteam bereit: er steht für Rückfragen zur Verfügung und evaluiert ständig die aktuelle System-Version.

Prozessbeurteilung. Ein *konservativer* Auftraggeber sollte darauf aufmerksam gemacht werden, dass bei größeren Software-Projekten in der Regel ein iteratives Vorgehen bzw. die Verwendung eines zyklischen Prozesses unumgänglich ist, weil die Anforderungen nicht vollständig bzw. korrekt in der Analyse-Phase erfasst werden können. Die Verwendung eines linearen Prozesses ist nur bei kleinen Systemen sinnvoll.

Bei *indifferentem* Auftraggeber-Typus bietet sich die Verwendung eines zyklischen Prozesses an. Bei Verwendung eines leichtgewichtigen Prozesses ist zu erwägen, die Rolle des Auftraggebers vor Ort durch einen Mitarbeiter des Software-Hauses zu simulieren.

Bei *kooperativem* Auftraggeber-Typus bietet sich je nach Anforderungskatalog-Dynamik[22] die Verwendung eines zyklischen oder eines leichtgewichtigen Prozesses an.

3.2.2 Das Entwicklerteam

Ein Team ist nach Balzert eine Gruppe, in der Mitarbeiter unterschiedlicher Qualifikation miteinander arbeiten, um eine gemeinsame Aufgabe zu erledigen.[23] Charakteristisch für Teamarbeit ist laut Balzert:

- Regelmäßige und kontinuierliche Kommunikation untereinander
- Von Fall zu Fall gegenseitige Abstimmung

[22] s. Seite 42
[23] [Balzert 1998, S. 168]

- Gleichberechtigte Mitbestimmung aller Teammitglieder bei der Diskussion von Methoden, Inhalten und Zielen der Arbeit und ihrer Durchführung
- Alle Teammitglieder sind gleichranging und agieren auch gleichrangig
- Verschiedene Teammitglieder übernehmen zeitweise die Führungsrolle, jeweils auf dem Gebiet, auf dem sie ihre Stärken haben.

Der Erfahrungshorizont, die Kenntnisse und die Arbeitseinstellung der Mitglieder eines Projektteams limitieren die erfolgreich einsetzbaren Entwicklungsprozesse. Neben dem *Teamgeist* der einzelnen Entwickler kommt dem *Selbstorganisationsgrad* und der *Homogenität* des Teams eine wichtige Rolle zu.

Teambildung und Teamgeist

Eine erfolgreiche Teamarbeit zeichnet sich dadurch aus, dass alle Teammitglieder ihre individuellen Tätigkeiten an dem Ziel ausrichten, das für das gesamte Team proklamiert wurde. Es ist Aufgabe des Teammanagers, eine entsprechende Arbeitsmoral bei den einzelnen Teammitgliedern zu fördern und über die Projektlaufzeit zu erhalten. Dabei hängt es vom Kooperationswillen der einzelnen Mitglieder und vom Geschick des Teammanagers ab, inwieweit diese Teambildung gelingt. Sie wird dadurch erschwert, dass die individuellen Vorlieben der einzelnen Teammitglieder für Methoden, Werkzeuge und dergleichen nicht zugunsten eines Strebens nach größtmöglicher Konformität ignoriert werden dürfen. Ein gut funktionierendes Team ist daran erkennbar, dass die einzelnen Mitglieder

- sich bei ihren Aktivitäten am Erfolg des Gesamtteams orientieren
- teamintern beschlossene Kompromisse akzeptieren
- individuellen Vorlieben tolerieren, solange diese nicht die Erreichung des Teamziels behindern
- andere Teammitglieder bei der Lösung von Problemen unterstützen.

In schlecht kooperierenden Teams ist dagegen zu beobachten, dass die einzelnen Teammitglieder – bewusst oder unbewusst – gegeneinander arbeiten. Der Vorteil der Teamarbeit wird hierdurch beeinträchtigt.

Viele moderne Entwicklungsprozesse profitieren von den Vorteilen effektiver Teamarbeit, sind aber andererseits auch darauf angewiesen, dass die Teambildung gelingt. Viele traditionelle Entwicklungsprozesse mit fester Rollenverteilung und tiefen Hierarchien sind dagegen weniger anfällig in dieser Hinsicht. Sie setzen stattdessen auf die Fähigkeit von Systemdesignern, isolierte Teilsysteme zu entwerfen, die von einzelnen Entwicklern implementiert und anschließend in das Gesamtsystem integriert werden.

Prozessbeurteilung. Ein Prozess ist daraufhin zu prüfen, ob er ein gut harmonierendes Team voraussetzt. Dies ist insbesondere bei Prozessen der Fall, in denen sämtliche Entwickler eine große persönliche Verantwortung für das Gesamtsystem übernehmen müssen. Haben die für ein Projekt vorgesehenen Entwickler bisher noch nicht erfolgreich als Team zusammengearbeitet, so muss

der Verlauf der Teambildung kritisch beobachtet werden. Gegebenfalls sind personelle Veränderungen vorzunehmen, wenn sich einzelne Mitglieder nicht in das Team integrieren.

Vor der Auswahl eines traditionellen Prozesses ist darauf zu achten, dass die Rolle des System-Architekten mit einem fachlich geeigneten und erfahrenen Kandidaten besetzt werden kann. Anderfalls kann eine mangelnde Isolation zwischen den Systemteilen Koordinations-Aktivitäten erforderlich machen, die im Prozess nicht vorgesehen sind.

Selbstorgansationsgrad

Die Entwicklung einer komplexen Software erfordert die Koordinierung verschiedenster Aktivitäten wie System-Analyse, System-Entwurf, Implementierung, Qualitätssicherung und Konfigurationsmanagement. Mit zunehmender Projektgröße werden diese Aufgaben mehr und mehr an Spezialisten übertragen, der individuelle Zuständigkeitsbereich also immer weiter eingeschränkt. Die Fähigkeit zur Selbstorganisation dieser Spezialisten ist in großen Projekten von untergeordneter Bedeutung, da ihre Koordination durch die Projektleitungsebene erfolgt.

In kleineren Teams ohne Hierarchien sind dagegen zumindest einige Entwickler erforderlich, die sich für alle anfallenden Aufgaben verantwortlich fühlen, idealerweise trifft dies auf alle Teammitglieder zu.

Prozessbeurteilung. Die folgenden Prozesstypen erfordern, dass ein maßgeblicher Teil der Entwickler einen hohen Selbstorganisationsgrad besitzt:

- Prozesse, die explizit auf kleine, homogene Teams angewiesen sind
- Prozesse, die ein Aufwärts-Tailoring während der Projektlaufzeit vorsehen
- Prozesse, die ein hohes Maß an „Chaos“ akzeptieren.

Homogenität des Entwicklerteams

In Prozessen, die für Entwicklerteams keine Hierarchien vorsehen, erweist sich eine Homogenität der Entwickler als erforderlich in Bezug auf

- Programmierkenntnisse
- Programmiererfahrung
- Programmierstil
- Präferenzen für bestimmte Werkzeuge
- Präferenzen für bestimmte Methoden
- Arbeitsmoral.

Bei zu großen Unterschieden in den oben genannten Punkten kommt es andernfalls immer wieder zur Durchführung von Arbeitsschritten, die dem Endprodukt nicht direkt zugute kommen:

- Grundsatzdiskussionen über Programmier-Techniken und -Richtlinien

- Konvertierung der Ausgabedateien exotischer Werkzeuge in kanonische Formate.

Verschärfend kommt hinzu, dass einige Arbeitsmethoden wie z.B. das bei *Extreme Programming* angewandte *Pair Programming* nur sinnvoll bei Entwicklern mit in etwa gleichen Programmierkenntnissen angewendet werden können. Die Homogenität eines Entwicklerteams lässt sich – insbesondere bei neu zusammengestellten Teams – nur schwer objektiv bestimmen. Sie kann darüber hinaus im Laufe eines Projekts schwanken. Beobachtungen aus teamintensiven Prozessen lassen vermuten, dass die Homogenität eines Teams sich durch die Stärkung des Teamgeistes während eines Projekts steigern lässt.

Prozessbeurteilung. Die Homogenität des Entwicklerteams ist vor allem von großer Bedeutung bei Prozessen, die

- keine Hierarchien für das Entwicklerteam vorsehen
- keine feste Zuständigkeitsverteilung für das Entwicklerteam vorsehen
- Pair Programming und Refactoring vorsehen.

Werden derartige Prozesse bei inhomogenen Teams angewandt, so droht die Isolierung einzelner Teammitglieder. Deren Arbeitsergebnisse können dann nicht nur wertlos, sondern im schlimmsten Fall kontraproduktiv für das Gesamtprojekt sein (s. Kapitel 5).

3.2.3 Firmenkultur

Vorgaben durch das Management

Die von Entwicklern zu beachtenden Anforderungen seitens des Managements unterscheiden sich von Software-Haus zu Software-Haus beträchtlich. Die Bandbreite reicht von kurz gehaltenen Berichten über den Status der einzelnen Projekte über die Darstellung von System-Architekturen bis zu Vorgaben für Werkzeuge, Methoden und Prozesse.

Das Management eines Software-Hauses muss auf die Außendarstellung des Unternehmens gegenüber seinen Kunden und potenziellen Neukunden achten und verordnet aus diesem Grunde unter Umständen die Verwendung von bestimmten Prozessen bzw. Qualitätssicherungsmaßnahmen wie der ISO-9000-Norm. Je nach Kundentypus ist diese Vorgehensweise sinnvoll oder auch kontraproduktiv, da sie einen Mangel an Flexibilität offenbart.

Prozessbeurteilung. Vorgaben für Werkzeuge, Methoden und Prozesse können unmittelbar bei der Beurteilung eines Kandidatenprozesses berücksichtigt werden. Besteht für ein Entwicklerteam die Maßgabe, ohne zeitlichen Vorlauf eine gültige System-Architektur präsentieren zu können, so wird es eine beträchtliche Zeit für die Aktualisierung der entsprechenden Dokumente aufbringen müssen und demzufolge bei der Weiterentwicklung des Systems langsamer vorankommen. Die Verwendung eines leichtgewichtigen Prozesses, bei dem ein wesentlicher Teil des Wissens über die System-Architektur im Gedächtnis der Entwickler bis zur endgültigen schriftlichen Fixierung „zwischengespeichert“ wird, ist unter diesen Umständen nicht sinnvoll.

Besteht hingegen nur eine Informationspflicht bezüglich des Projektstatus, so können beispielsweise bei der Verwendung von automatisierten Integrationstests mit geringem technischem Aufwand sehr aktuelle und aussagekräftige Reports auf einer Intranetseite angeboten werden. Diese können dann sowohl dem Management wie auch dem Entwicklerteam als Informationsquelle dienen (s.a.: [**?**, S. 84]).

Qualitätssicherungsmethoden

Viele Software-Firmen besitzen eine so genannte Qualitätssicherungs-Abteilung, deren Mitarbeiter sich im sportlichen Sinne als „Gegner" der Entwicklungsteams verstehen.[24] Ihre Aufgabe besteht traditionell darin, die von den Entwicklerteams gelieferte und grob vorgetestete Software einer stichprobenartigen Prüfung zu unterziehen und je nach Ergebnis entweder zu akzeptieren oder mit einem Hinweis versehen an das Entwicklerteam zurück zu geben. Grund für diese Art der Aufgabenteilung ist die Beobachtung, dass ein Entwickler ein selbst geschriebenes Programm häufig nur sehr oberflächlich testet. Als Grund wird vielfach eine Art innere Hemmung des Programmierers vermutet, die eigenen Programmierfehler zu entdecken. Ob dies wirklich zutrifft, sei dahingestellt. Die gute Akzeptanz von automatisierten Tests in Verbindung mit dem Test-First-Ansatz weist eine jedenfalls neue Lösung für die beschriebene Problematik auf.

Bei der Kooperation mit einem Entwicklerteam, das seine Software durch automatisierte Tests absichert, übernimmt die Qualitätssicherungs-Abteilung idealerweise die Aufgabe, Integrationstests zu spezifizieren, die schließlich vom Entwicklerteam codiert und regelmäßig durchgeführt werden. Die Aufgabe der Entwickler ist in diesem Szenario ebenfalls klar definiert: sie besteht darin, das System so zu konstruieren, dass es schließlich alle Integrationstests fehlerfrei absolviert. Bei konsequenter Anwendung dieser Methode ergibt sich als zusätzlicher Vorteil, dass die wichtige Kenngröße *Projektstatus* sich objektiv als „Anteil der fehlerfrei absolvierten Integrationstests im Verhältnis zur Gesamtzahl der Integrationstests" definieren und jederzeit ohne Aufwand ermitteln sowie publizieren lässt.[25]

Prozessbeurteilung. Die Spezifikation von Integrationstests durch die Qualitätssicherungsabteilung eines Software-Hauses stellt eine harmonische Art der Kooperation mit Entwicklerteams dar, die ihrerseits automatisierte Tests zur Qualitätssicherung einsetzen. Obwohl sie keine notwendige Bedingung ist, wird das Entwicklerteam durch diese Form der Arbeitsteilung entlastet. Prozesse, die intensive automatisierte Tests vorsehen, werden durch sie also unterstützt.

3.2.4 Arbeitsbedingungen

Die räumlichen und zeitlichen Bedingungen, unter denen das Entwicklerteam arbeitet, sollten bei der Auswahl des Entwicklungsprozesses ebenfalls berücksichtigt werden. Prozesse, die direkter mündlicher Kommunikation den Vorzug vor anderen Kommunikationsformen einräumen, lassen sich nicht wie vorgesehen umsetzen, wenn die Entwickler räumlich getrennt voneinander arbeiten müssen. Beobachtungen von Cockburn zeigen, dass

[24] vgl. [**?**, S. 144 ff.]
[25] vgl. [**?**, S. 84]

1. bereits geringe Unterschiede bei der Gestaltung der Arbeitsplätze große Auswirkungen auf das kommunikative Verhalten von Entwicklern haben.[26]

2. der Versuch, räumliche Distanzen durch Kommunikationsmedien wie E-Mail, Telefon oder Bildtelefon zu überbrücken, keinen vollwertigen Ersatz für direkte mündliche Kommunikation darstellt.[27]

Gestaltung der Arbeitsräume

Bei der Gestaltung von Entwickler-Arbeitsräumen muss – je nach den bevorzugten Kommunikationsformen eines Prozesses – ein Kompromiss zwischen zwei wichtigen Faktoren gefunden werden:

1. *Schutz vor Störungen von außen*
   Es ist seit langem bekannt, dass ein Mensch etwa 15 Minuten benötigt, bevor er bei der Lösung einer anspruchsvollen geistigen Aufgabe zur Tat schreitet und dass diese „Eintauch-Phase“ nach jeder Störung des Gedankenflusses von außen wieder von Neuem durchlaufen werden muss.[28] Es ist deshalb wünschenswert, Entwickler von Störungen durch die Außenwelt weitgehend abzuschotten.

2. *Möglichkeit der spontanen Kommunikation zwischen den Entwicklern im Bedarfsfall*
   Fehlt einem Entwickler die Möglichkeit, ohne großen Aufwand einen Teamkollegen wegen einer kurzen Nachfrage kontaktieren zu können, so muss er stattdessen Vermutungen über bestimmte Sachverhalte aufstellen oder die Bearbeitung der aktuellen Aufgabe aufschieben.

Des weiteren sind gegebenenfalls folgende Aspekte zu berücksichtigen:

- Erlauben die Arbeitsplätze Pair Programming?
- Stehen den Entwicklern persönliche Workstations für die Bearbeitung von E-Mails zur Verfügung?

Prozessbeurteilung. Es ist zu überprüfen, ob die im Prozess vorgesehenen Kommunikationformen sich in den verfügbaren räumlichen Gegebenheiten umsetzen lassen bzw. ob eine Anpassung der Räume möglich ist. Bei der Beurteilung eines Prozesses ist ggf. hilfreich, dass wegen der Abhängigkeit der favorisierten Kommunikationsformen von den räumlichen Gegebenheiten bei manchen Prozessen angegeben wird, in welcher Umgebung sie erfolgreich angewandt worden sind.[29]

Zeitliche Bedingungen

Ebenso wie die räumlichen Bedingungen schränken auch die zeitlichen Arbeitsbedinungen die Möglichkeiten der Kommunikation zwischen den Entwicklern unter Umständen ein und sollten deshalb bei der Prozessauswahl beachtet werden. Folgende Faktoren sind in diesem Zusammenhang zu überprüfen:

[26] vgl. [?, S. 77 ff.]
[27] vgl. [?, S. 95]
[28] vgl. [?, S. 71]
[29] vgl. [?, S. 89] und [?, S. 77 ff.]

- Gibt es eine Kernarbeitszeit, zu der alle Teammitglieder für individuelle Nachfragen oder für Meetings verfügbar sind?
- Sind einzelne Entwickler – beispielsweise wegen Teilzeitbeschäftigung – nicht permanent für Rückfragen verfügbar?
- Wird seitens der Geschäftsleitung von den Entwicklern die Ableistung von Überstunden erwartet?
- Ist das Entwicklerteam über mehrere Zeitzonen verteilt und kann deshalb nur eingeschränkt telefonisch kommunizieren?

Prozessbeurteilung. Es ist zu überprüfen, ob die im Prozess vorgesehenen Kommunikationsformen mit den zeitlichen Bedingungen des Projekts vereinbar sind.

Kapitel 4

Das Projekt Netlife Finance Suite

4.1 Aufgabenstellung

Die Netlife AG wurde 1996 gegründet und ist seit Juni 1999 börsennotiert. Der Firmensitz befindet sich in Hamburg, daneben gibt es Niederlassungen in Frankfurt, Leipzig und Zürich. Der Kundenstamm liegt im Bereich Banken, Sparkassen und Finanzrechenzentren.

Die Wurzeln von Netlife liegen in der Konstruktion von Systemen für *Homebanking per Internet*. 1997 wurde als erster Kunde die Postbank AG gewonnen. 1998 kam das Geschäftsfeld *Brokerage*[1] hinzu, das sich inzwischen zum umsatzstärksten Bereich entwickelt hat. Für die Zukunft sind multikanalfähige Projekte im Allfinanzbereich[2] geplant.

Die Software-Entwicklung bei Netlife ist nach Märkten in die zwei Abteilungen Banking und Brokerage aufgeteilt. Den beiden Abteilungsleitern unterstehen direkt die Leiter der einzelnen Kundenprojekte.

Die Banking-Abteilung stand im Frühjahr 2001 vor der Aufgabe, vier unterschiedliche Systeme pflegen zu müssen, die in den vergangenen Jahren in verschiedenen Projekten entstanden waren. Obwohl der Einsatzzweck der Systeme sehr ähnlich war, wurde bis auf einige wenige Utilities kein gemeinsamer Quellcode verwendet. Ein internes Projekt mit folgenden Zielsetzungen wurde ins Leben gerufen:

- Analyse und Vergleich der vier bestehenden Systeme
- Definition bzw. Entwurf und Implementierung eines vereinheitlichten Systems für zukünftige Projekte
- Aufwandsabschätzung für die Migration der bestehenden Systeme auf das vereinheitlichte System
- Auswahl und Etablierung eines Software-Entwicklungsprozesses für zukünftige Projekte.

[1] Durch elektronische Plattformen unterstützter Wertpapierhandel für Privatkunden
[2] Integration von Bank- und Versicherungsdienstleistungen

4.2 Projektverlauf

Das Projekt startete im April 2001 und wurde nach einem Jahr erfolgreich im Sinne der obigen Zielsetzungen abgeschlossen. Chronologisch lassen sich fünf Phasen unterscheiden:

1. Auswahl eines Entwicklungsprozesses
2. Analyse der bestehenden Systeme
3. Entwurf eines vereinheitlichten Systems
4. Implementierung des vereinheitlichten Systems
5. Migration der Altsysteme

4.2.1 Auswahl eines Entwicklungsprozesses

Um bereits in der Analyse-Phase nach einem etablierten Modell vorgehen zu können, wurde zu Beginn des Projekts in Abstimmung mit dem Abteilungsleiter entschieden, mit welchem Prozess das vereinheitlichte System und ggf. zukünftige Projekte entwickelt werden sollten. Als Kandidaten wurden *Extreme Programming* (im Folgenden kurz: *XP*) und der *Rational Unified Process* (im Folgenden kurz: *RUP*) in Betracht gezogen und auf Basis der in Abschnitt 3.2[3] erläuterten Kriterien auf ihre Eignung für das anstehende Projekt untersucht.

Projekteigenschaften

- *Größe des Entwicklerteams.* Das Team umfasste ursprünglich vier Entwickler und wurde nach einem Monat auf sieben Entwickler aufgestockt. Die maximale Größe eines Entwicklerteams in der Banking-Abteilung betrug in der Vergangenheit zehn Entwickler.
  - XP wurde als *geeignet* beurteilt, da es bereits in Projekten mit bis zu 14 Entwicklern erfolgreich eingesetzt worden war.
  - Nach Aussage der *Rational Software Corporation* ist der RUP für Entwicklerteams jeder Größe geeignet, erfordert jedoch für kleinere Projekte ein Tailoring. Der RUP wurde deshalb als *bedingt geeignet* beurteilt.
- *Kritikalität des Systems.* Auf Cockburns Skala wurde der maximale Schaden, den das zu realisierende System bewirken kann, als „Verlust von entbehrlichen Geldsummen“ eingestuft.
  - Sowohl XP wie auch der RUP wurden bereits erfolgreich in Projekten mit dieser Kritikalität eingesetzt und wurden deshalb als *geeignet* eingestuft.

[3] s. Seite 37 ff.

- *Dynamik des Anforderungskatalogs.* Die Dynamik des Anforderungskatalogs wurde als „hoch“ eingestuft, weil neben den bankfachlichen Geschäftsvorfällen später auch Geschäftsvorfälle aus den Bereichen Brokerage und Versicherungswesen zu integrieren waren, mit denen das Entwicklerteam relativ wenig Erfahrung hatte.
  - XP wurde als *sehr geeignet* eingestuft, weil es grundlegende Änderungen bei den Systemanforderungen mit gezielten Maßnahmen adressiert.
  - Der RUP wurde als *eingeschränkt geeignet* eingestuft, da er eine relativ lange Entwurfsphase vorsieht, die bei wechselnden Anforderungen immer wieder zu durchlaufen ist.
- *Auftraggebertypus.* Es handelte sich um ein internes Projekt, bei dem der Abteilungsleiter in Verbindung mit dem Teammanager die Rolle eines *kooperativen Auftraggebers* wahrnahm, der permanent die aktuelle System-Version begutachtete und neue Systemeigenschaften anregte.
  - Sowohl XP wie auch der RUP wurden als *geeignet* eingestuft.

Das Entwicklerteam

- *Teambildung und Teamgeist.* Alle Entwickler des Teams hatten jahrelange Erfahrung als Mitglied in Software-Entwicklungsteams. Das Team arbeitete jedoch zum ersten Mal in dieser Besetzung zusammen. Der Teammanager hatte eine dreijährige Erfahrung in der Leitung von kleineren Teams.
  - XP wurde als *bedingt geeignet* eingestuft, weil es ein gut funktionierendes Team voraussetzt.
  - Der RUP wurde als *geeignet* eingestuft, weil er auch bei nicht harmonierenden Teams nach einen Top-Down-System-Entwurf eine sinnvolle Aufgabenverteilung erlaubt.
- *Selbstorganisationsgrad.* Von einzelnen Teammitgliedern war bekannt, dass sie einen hohen Selbstorganisationsgrad besitzen. Deshalb konnte angenommen werden, dass sie Eigeninitive entwickeln und damit das Projekt vorantreiben werden. Diese Annahme hat sich im Laufe des Projekts bestätigt.
  - Da die für XP notwendige Voraussetzung gegeben war, wurde es als *geeignet* eingestuft.
  - Der RUP stellt geringere Ansprüche an den Selbstorganisationgrad der Entwickler als XP, das Kriterium ist für ihn also weniger relevant. Er wurde ebenfalls als *geeignet* eingestuft.
- *Homogenität des Entwicklerteams.* Die Programmierkenntnisse der Entwickler waren zwar unterschiedlich, differierten aber nicht so stark, dass die Durchführung von Pair Programming in Frage gestellt gewesen wäre.
  - Sowohl XP wie auch der RUP wurden als *geeignet* eingestuft.

Firmenkultur

- *Vorgaben durch das Management.* Seitens des Managements gab es keine Vorgaben für bestimmte Werkzeuge, Methoden oder Prozesse. Nach Fertigstellung der ersten System-Version wurde deren Architektur dem Management in einer kurzen Präsentation erläutert.

  - Die sporadische Erstellung von Dokumenten, die die System-Architektur darstellen, ist – wie bereits an anderer Stelle erläutert – gut mit dem Release-Management von XP vereinbar. XP wurde deshalb als *geeignet* eingestuft.
  - Bei Verwendung des RUP wäre eine aktuelle Darstellung der System-Architektur jederzeit möglich gewesen. Da diese Anforderung jedoch nicht bestand, wurde der RUP als *bedingt geeignet* eingestuft.

- *Qualitätssicherungsmethoden.* Eine Zusammenarbeit mit der Qualitätssicherungsabteilung war nicht vorgesehen. Bei informellen Gesprächen mit deren Leitung stellte sich allerdings heraus, dass sie nicht bereit war, die klassische Rolle der Qualitätssicherung dahingehend zu verändern, dass durch die Qualitätssicherungsabteilung vorwiegend Akzeptanztests spezifiziert werden.

  - Bei einem XP-Projekt ist eine neue Art der Aufgabenteilung zwischen QS-Abteilung und Entwicklerteam wünschenswert, aber nicht unbedingt erforderlich. XP wurde deshalb als *bedingt geeignet* eingestuft.
  - Der RUP erlaubt auch die Einbindung einer konventionellen QS-Abteilung. Er wurde deshalb als *geeignet* eingestuft.

Arbeitsbedingungen

- *Gestaltung der Arbeitsräume.* Dem Team wurden vier unmittelbar benachbarte Arbeitsräume mit je zwei Arbeitsplätzen zur Verfügung gestellt.

  - Die für XP notwendigen räumlichen Arbeitsbedingungen waren damit gegeben. XP wurde als *geeignet* eingestuft.
  - Der RUP ist auch für verteilte Entwicklungsteams geeignet und hat keine speziellen Anforderungen bezüglich der Arbeitsräume. Die für die Unterstützung von verteilten Entwicklerteams notwendigen Kommunikationsmittel bzw. Artefakte sind bei einem lokalen Team per Tailoring zu entfernen. Der RUP wurde deshalb als *bedingt geeignet* eingestuft.

- *Zeitliche Bedingungen.* Auf Grund der Arbeitszeitregelungen war sichergestellt, dass alle Entwickler während einer Kernarbeitszeit gleichzeitig vor Ort sein werden.

  - Die für XP notwendigen zeitlichen Arbeitsbedingungen waren damit gegeben. XP wurde als *geeignet* eingestuft.

– Der RUP ist auch für verteilte Entwicklungsteams geeignet und hat keine speziellen Anforderungen bezüglich der zeitlichen Bedingungen. Die für die Unterstützung von zeitlich versetzt arbeitenden Entwicklerteams notwendigen Kommunikationsmittel bzw. Artefakte sind bei einem Team mit zeitlich parallel arbeitenden Entwicklern per Tailoring zu entfernen. Der RUP wurde deshalb als *bedingt geeignet* eingestuft.

Fazit

Die Wahl des Entwicklungsprozesses fiel nach Durchführung der oben skizzierten Evaluierung auf *Extreme Programming*, weil es bei der überwiegenden Zahl von Kriterien besser geeignet erschien als der *Rational Unified Process*.

Das Projekt startete im April 2001 mit einem Team von vier erfahrenen Entwicklern, von denen einer gleichzeitig die Rolle des XP-Coaches übernahm. In dieser Funktion erstellte er auf Basis der einschlägigen Literatur eine Einführung in XP, die er den anderen Projektteilnehmern in Form einer Schulung vermittelte. Einen Monat nach Projektstart wurde das Team um drei neue und relativ unerfahrene Entwickler aufgestockt. Sie wurden ebenfalls in einer Schulung mit dem Entwicklungsprozess vertraut gemacht.

Da es sich um ein internes Projekt handelte, übernahm der Abteilungsleiter pro forma die Auftraggeber-Rolle. Weil der Funktionsumfang des zu realisierenden Systems durch die Altsysteme vorgegeben war, beschränkte sich seine Aufgabe auf die Priorisierung von Userstories.

4.2.2 Analyse der bestehenden Systeme

Die Analyse der bestehenden Systeme ergab erwartungsgemäß, dass die grundlegende Architektur bei allen gleich war (s. Abbildung 4.1). Es handelt sich um eine klassische Client-Server-Architektur mit vier Schichten auf der obersten Ebene. Der Client läuft als Anwendung auf dem Rechner des Bankkunden, üblicherweise handelt es sich um einen HTML-Webbrowser. Der Client kommuniziert über eine verschlüsselte Internet-Verbindung mit dem Banking-Server. Alle Nachrichten der Kundenseite werden vom Banking-Server im Kontext einer kundenbezogenen *Session* interpretiert. Eine Session wird durch den Kunden eröffnet, indem dieser seine Kundenkennung und ein Passwort auf einer definierten Startseite eingibt. Eine Session endet, wenn der Kunde dies explizit veranlasst oder wenn der Server für eine gewisse Zeit keine Nachrichten von der Client-Seite mehr empfangen hat. Der Banking-Server fungiert als Vermittler zwischen dem Kunden und dem Rechenzentrum der Kundenbank. Im Einzelnen übernimmt er folgende Aufgaben:

- Verwaltung der kundenbezogenen Sessions
- Authentisierung des Kunden
- Prüfung der Eingabedaten des Kunden auf Plausibilität
- Weiterleitung der Kundenaufträge an das Host-System
- Interpretation der Antwortnachricht des Host-Systems

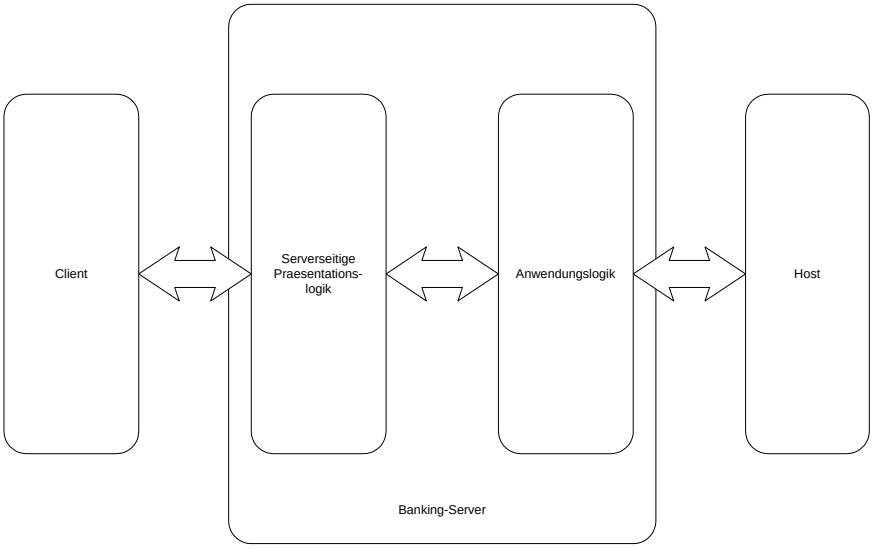

Abbildung 4.1: Banking Server

- Formatierung der Antwortnachricht für den Client.

Die Unterschiede zwischen den bestehenden Systemen offenbarten sich erst bei detaillierterer Betrachtung:

- Die Hostschnittstelle wurde vom jeweiligen Auftraggeber fest vorgegeben und unterschied sich von System zu System auf allen Protokollebenen.
- In allen Systemen sprach die Anwendungslogik direkt die projektspezifische Hostschnittstelle an.
- Die von der Anwendungslogik interpretierten Datenformate orientierten sich in der Regel an den Datenformaten der Hostschnittstelle.
- Die Formate für Standarddatentypen wie Betrag, Währung, Datum und dergleichen unterschieden sich teilweise innerhalb der Systeme von Anwendungsfall zu Anwendungsfall.
- Anwendungslogik und Präsentationslogik waren nicht immer sauber getrennt.
- Das Protokoll zwischen Client und Banking-Server war relativ komplex, weil der Server in einigen Systemen zustandsbehaftet war.
- Die Fehlercodes waren teilweise unübersichtlich strukturiert und unterschieden sich von System zu System.

Zusammenfassend lässt sich sagen, dass alle Systeme für sich genommen eine schlanke und sinnvolle Einheit mit sehr spezifischen internen Schnittstellen bildeten und dass eine Ähnlichkeit lediglich auf der Ebene von abstrakten Zweckbeschreibungen bestand.

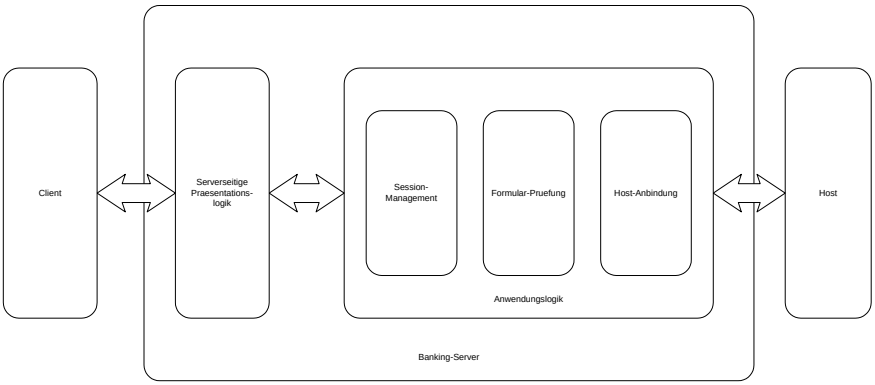

Abbildung 4.2: Architektur Netlife Finance Suite

4.2.3 Entwurf eines vereinheitlichten Systems

Der Entwurf des vereinheitlichten Zielsystems namens *Netlife Finance Suite* startete mit der Beschreibung des von außen beobachtbaren Systemverhaltens in Form von Userstories. Dabei handelte es sich in der Mehrzahl um so genannte *Geschäftsvorfälle* wie „Einzelüberweisung einreichen", „Dauerauftrag einrichten", „Kontoauszug abrufen" und dergleichen.

Um eine Grundlage für die Aufwandsabschätzung der Userstories zu erhalten, wurde nun das lediglich über sein Verhalten definierte System in Einzelkomponenten zerlegt, die ausschließlich über ihre Verantwortlichkeiten bestimmt waren. Als wichtigste Komponenten ergaben sich:

- Sessionmanagement
- Authentisierung
- Formularprüfung
- Datenformatkonvertierung
- Hostanbindung.

Als sehr hilfreich bei dieser Analyse erwies sich die Definition der folgenden System-Metapher: „Ein Bankkunde überreicht einem Kundenbetreuer ein ausgefülltes Formular. Dieser prüft die Eintragungen des Kunden und gibt ihm das Formular zurück, wenn es nicht korrekt ausgefüllt ist. Abschließend leitet der Kundenbetreuer das Formular weiter an das Rechenzentrum der Bank." Diese Metapher spiegelt sich in der Detailsicht der Netlife Finance Suite wider (s. Abbildung 4.2). Dabei entspricht der Client dem Bankkunden und die Anwendungslogik dem Kundenbetreuer. Die vom Client zur Anwendungslogik übertragenen Datenstrukturen sind Spezialisierungen des Typs „Formular". Datenstrukturen, die von der Anwendungslogik zum Client zurückfließen (beispielsweise Kontoauszüge), sind Spezialisierungen des Typs „Dokument".

Man beachte, dass es sich bei der Netlife Finance Suite nicht um ein weiteres Homebanking-System im herkömmlichen Sinne, sondern um ein so genanntes *Framework* handelt, das lediglich als Basis für konkrete Homebanking-Systeme

fungiert. Deshalb wird der Quelltext der Finance Suite getrennt verwaltet. Die Einbindung in ein konkretes System erfolgt über eine Bibliotheksdatei. Gegenüber den bestehenden Systemen hat die Netlife Finance Suite folgende Vorzüge:

- Die Logik zum Prüfen von Formularfeldern muss nur noch einmal implementiert werden. Projektspezifische Anpassungen erfordern in der Regel lediglich Änderungen in einer Konfigurationsdatei.
- Durch die standardisierten Außenschnittstellen kann die Entwicklung des Frontends und der Hostanbindung entkoppelt werden.
- Die Frontendanbindung basiert auf einem einfachen und stabilen Protokoll mit zustandslosem Server.
- Die Hostanbindung besteht im Wesentlichen aus einer Datenkonvertierung.
- Die Kommunikation zwischen den beteiligten Entwicklern wird durch die Standardisierung der Begriffswelt verbessert.

4.2.4 Implementierung des vereinheitlichten Systems

Nachdem eine vollständige Liste der Geschäftsvorfälle aufgestellt und als Userstories beschrieben war, schätzten die Entwickler grob den jeweiligen Realisierungsaufwand, um eine Basis für die Release-Planung zu erhalten. Nach einer initialen Schätzung der *Project Velocity* wurden vom Leiter der Banking-Abteilung Userstories für das erste Release ausgewählt, das in drei Monaten fertig zu stellen war.

Für die Implementierung der Netlife Finance Suite wurden zwei alternative Vorgehensweisen erwogen:

1. Implementierung eines neuen Systems
2. Refactoring desjenigen Systems, das dem vereinheitlichten System am meisten ähnelt.

Aus folgenden Gründen fiel die Entscheidung auf die Implementierung eines neuen Systems:

- Der Entwurf von automatisierten Tests für die bestehenden Systeme wäre durch eine teilweise mangelhafte Dokumentation sehr mühsam gewesen.
- Da alle bestehenden Systeme permanent weiter entwickelt wurden, hätte das Refactoring auf einer Kopie basieren müssen, die zum Zeitpunkt der Fertigstellung des vereinheitlichten Systems nicht mehr dem aktuellen Stand des in Produktion befindlichen Systems entsprochen hätte. Hierdurch wäre der Hauptvorteil der zweiten Variante hinfällig gewesen.
- Das Zielsystem sollte ein Framework sein. Die Trennung zwischen Framework und konkretem System würde bei der zweiten Variante zusätzlichen Aufwand erfordern.
- Ein Entwicklerteam ist deutlich motivierter bei der Konstruktion eines neuen Systems als bei der Änderung eines vorgegebenen Systems.

Das Team vereinbarte, die folgenden Regeln bei der Implementierung des neuen Systems einzuhalten:

- Das Verhalten jeder neu geschaffenen Komponente ist durch einen automatisierten Test abzusichern.
- Die Integration von Änderungen bzw. neuen Komponenten erfolgt in kleinen Einheiten und sehr häufig.
- Eine Integration darf nur erfolgen, wenn im neuen Systemzustand alle automatisierten Tests erfolgreich absolviert werden können.
- Es gibt Collective Code Ownership. *Alle* Entwickler sind befugt, an *allen* Teilen des Systems Änderungen vorzunehmen.
- System-Komponenten sind so einfach wie möglich zu entwerfen und zu implementieren. Es werden ausschließlich aktuell benötigte Variablen, Methoden und Parameter deklariert.
- Beim Entwurf von Komponenten ist die Orientierung an einem etablierten Entwurfsmuster oder einer Metapher anzustreben. Entwurfsmuster finden sich beispielsweise in [?].
- Zentrale System-Komponenten dürfen nicht von einem Entwickler allein entworfen und implementiert werden.
- Die Dokumentation des Systems kann sich auf die formell korrekte Dokumentation des Quelltextes beschränken. Diese ist jedoch unverzichtbar und auch bei Änderungen sorgfältig anzupassen.
- Im Zweifelsfall ist ein umfangreiches Refactoring einer schnellen Lösung vorzuziehen, die nur ein akutes Problem löst, aber nicht zur aktuellen Architektur passt.

Die meisten dieser Regeln entsprachen den Empfehlungen von Kent Beck in [?]. Auf die Abweichungen wird in Kapitel 5 eingegangen.

Drei Monate nach Projektstart stand im Juni 2001 ein funktionsfähiger Prototyp zur Verfügung, der alle für das erste Release vorgesehenen Funktionen aufwies. Der Host war bei diesem Prototypen als so genanntes *Mock-Up* implementiert, simulierte also in realistischer Weise einen tatsächlichen Host. Dies ermöglichte die Definition einer Vielzahl von automatisierten Tests, die später auch in Kundenprojekten verwendet werden konnten.

4.2.5 Migration der Altsysteme

Im Oktober 2001 startete das erste Projekt auf Basis der Netlife Finance Suite, das zweite im Februar 2002. Die Migration des ersten Altsystems startete im November 2001 und wurde wie auch das erste Neuprojekt im Mai 2002 abgeschlossen.

Das mit der Migration des Altsystems betraute Team wurde durch die Entwickler des Finance-Suite-Frameworks in Form von Beratung und gemeinsamer Fehlersuche unterstützt. Als nicht trivial erwies sich die Koordination der Weiterentwicklung bzw. Anpassung des Frameworks mit den diversen Projekten,

auf denen das Framework basierte. Das Release-Management von Extreme Programming musste zu diesem Zweck etwas erweitert werden. Eine eingehende Beschreibung dieses erweiterten Release-Management findet sich in Abschnitt 4.2.6.

Das im zweiten Neuprojekt zu realisierende System wird neben Banking-Geschäftsvorfällen auch Brokerage-Geschäftsvorfälle enthalten. Die Brokerage-Abteilung hat zeitlich parallel zur Entwicklung des Finance-Suite-Frameworks ebenfalls zwei bestehende Systeme vereinheitlicht und zusammengeführt. Die Integration des Finance-Suite-Frameworks mit dem vereinheitlichten Brokerage-System findet zur Zeit statt.

4.2.6 Erweiterung des Release-Management

Das Release-Management von XP ist sehr gut geeignet für Projekte, in denen für einen Kunden ein maßgeschneidertes System entwickelt wird. Dieses Szenario war bei der Entwicklung der Netlife Finance Suite nur in den ersten sechs Monaten gegeben.

Mit dem Start des ersten Neuprojekts entstand das Problem, dass einerseits die Entwicklung des Frameworks weiterlaufen und andererseits eine stabile Grundlage für die spezifischen Komponenten des Neuprojekts verfügbar sein musste. Das Problem wurde verschärft durch die Anforderung, dass bei der Framework-Version, auf der das Projekt aufsetzte, jederzeit Fehlerbehebungen möglich sein mussten, ohne die projektspezifischen Komponenten an die neueste Framework-Version anpassen zu müssen. Durch die folgenden Regeln wurde das XP-Release-Management so erweitert, dass diese Anforderungen erfüllt wurden:

- Mit Hilfe eines Versionsverwaltungssystems wird nach Abschluss jeder Iteration ein Marker gesetzt, der für alle Quelltextdateien des Frameworks einen von der weiteren Entwicklung losgelösten Zustand definiert. Damit ist die nachträgliche Rekonstruktion aller Quelltextstände zu einem bestimmten Zeitpunkt möglich. Darüber hinaus können die durch den Marker definierten Zweitversionen der Quelltexte ohne Auswirkungen auf die Hauptversion verändert werden.

- Ein Projekt auf Basis der Netlife Finance Suite benutzte als Grundlage für die laufende Entwicklung nicht die jeweils aktuelle Version der Netlife Finance Suite, sondern die Version, die durch den letzten Marker definiert war.

- Die auf den Zweitversionen vorgenommenen Fehlerbehebungen werden ggf. vor dem Abschluss einer Iteration auf die Hauptversionen der jeweiligen Quelltexte übertragen.

- Das mit der Weiterentwicklung des Frameworks betraute Team dokumentiert in einer bestimmten Datei alle Änderungen und weist auf die bei einer Portierung evtl. notwendigen Anpassungen hin.

- Nach Abschluss einer Iteration portiert das Projektteam mit Unterstützung des Framework-Teams die projektspezifischen Komponenten auf die nunmehr aktuelle Version der Netlife Finance Suite.

Diese Regeln erwiesen sich als hinreichend, solange sich das im Projekt entwickelte System noch nicht im Einsatz befand. Vor der endgültigen Auslieferung des Systems wurde jedoch eine nochmalige Erweiterung des Release-Management notwendig, denn nachdem an der Netlife Finance Suite alle für das im Projekt zu realisierende System erforderlichen Anpassungen vorgenommen worden waren, bestand seitens des Projektteams kein Interesse mehr an einer regelmäßigen Portierung auf die aktuellste Framework-Version. Stattdessen hatte von nun an die Stabilität des Frameworks für das Projektteam höchste Priorität. Deshalb wurden so genannte *Major Releases* vereinbart, die jeweils am Ende eines Release-Zyklus zu definieren waren. Um den Wartungsaufwand für diese Releases gering zu halten, sollten in der letzten Iteration eines Release-Zyklus keine neuen Funktionen mehr in das Framework integriert, sondern vornehmlich Fehlerbehebungen vorgenommen werden. Die obige Liste ist also um folgende Regeln zu ergänzen:

- Am Ende eines Release-Zyklus ist ein Major Release zu markieren, das sich durch eine besonders niedrige Fehlerrate auszeichnet.
- Eine Endauslieferung an einen Kunden muss auf einem Major Release basieren. Es ist anzustreben, möglichst viele Projekte auf demselben Major Release aufsetzen zu lassen.

Abschließend sei noch darauf hingewiesen, dass die Leiter der jeweiligen Projekte die in XP definierte Rolle des Auftraggebers ausfüllten, bis das Major Release für das jeweilige Projekt bereitstand.

4.3 Untersuchungsmethode

Die in der vorliegenden Arbeit beschriebenen praktischen Erfahrungen mit *Extreme Programming* wurden zunächst vom Autor der vorliegenden Arbeit in Form von Tagebucheintragungen festgehalten. Er fungierte zugleich als XP-Coach und Manager des Entwickler-Teams. Daneben standen die im Rahmen der Release-Planung vorgenommenen Aufwandsabschätzungen und die entsprechenden Effektivwerte für die Realisierung von Userstories in Form von elektronisch erfassten Tabellen zur Verfügung.

Um bei der Beurteilung von Faktoren wie Arbeitsumgebung, Arbeitsbedingungen, Softwarequalität, Systemstabilität objektivere Urteile zu erhalten, wurde mit allen Entwicklern etwa ein Jahr nach Projektstart ein halbstandardisiertes Interview geführt. Die Fragen sowie die anonymisierten Antworten der Entwickler sind im Anhang A zu finden.

Kapitel 5

Extreme Programming in der Praxis

Eine Zielsetzung beim Start des Projekts *Netlife Finance Suite*[1] war die Einführung und Erprobung eines Software-Entwicklungsprozesses, der idealerweise auch in nachfolgenden Projekten verwendet werden und schließlich firmenweit etabliert werden sollte. Im Folgenden werden die dabei gewonnenen Erfahrungen auf ihre Allgemeingültigkeit untersucht und mögliche Lösungsansätze für aufgetretene Probleme erörtert.

Das Kapitel ist thematisch nach den Disziplinen gegliedert, die bei der Abwicklung von Projekten zu absolvieren sind und die deshalb von Software-Entwicklungsprozessen – in sehr unterschiedlicher Ausführlichkeit – in Form von Aktivitäten adressiert werden.

5.1 System-Entwurf

Der System-Entwurf geschieht bei XP im Unterschied zu traditionellen Entwicklungsprozessen nicht in einer gesonderten Projektphase, sondern erfolgt permanent während der Weiterentwicklung des Systems. Eine besondere Situation stellt jedoch der initiale System-Entwurf zu Beginn eines Projekts dar. Die dabei entstandene System-Architektur wird in der Folgezeit fortlaufend durch so genannte Refactorings für die adäquate Umsetzung von neuen Anforderungen vorbereitet.

Dieser Ansatz unterscheidet sich radikal von traditionellen Vorgehensweisen. Seine Begründung findet sich in der neuen Betrachtungsweise von Risiken bei der Software-Entwicklung.

In der Praxis ergibt sich das Problem, dass Entwickler gewöhnlich mit anderen Arten des System-Entwurfs vertraut sind und speziell motiviert werden müssen, die Entwurfsprinzipien von XP zu berücksichtigen.

5.1.1 Initialer System-Entwurf

Der initiale System-Entwurf bei *XP* erfolgt nach Kent Beck in zwei Schritten:

[1] s. Kapitel 4

1. Bestimmung einer passenden System-Metapher[2]
2. Implementierung einer Menge von Userstories, „von denen man erwartet, dass sie dazu zwingen, die gesamte Architektur zu erstellen".[3]

Die Umsetzung der betreffenden Userstories im zweiten Schritt erfolgt mit einfachsten Mitteln. Es wird ausdrücklich vermieden, Klassenhierarchien, Methodenrümpfe, Kommunikationsprotokollebenen und dergleichen zu implementieren, bevor sie tatsächlich für die aktuelle Userstory benötigt werden.

Die Nachteile dieses Vorgehens sind bekannt und werden bewusst in Kauf genommen:

- Die erste Architektur-Version ist in der Regel nicht ausreichend für die Umsetzung der noch nicht realisierten Systemeigenschaften. Sie muss nachträglich Schritt für Schritt erweitert und angepasst werden.
- Unerfahrene Entwickler sind mit der fortlaufenden Umsetzung beider Schritte – die nicht näher beschrieben ist – überfordert.
- Das obige Vorgehen ist über Jahrzehnte – obwohl in der Praxis durchaus üblich – als unsystematisch und anfängerhaft gebrandmarkt worden. Dies führt dazu, dass viele Entwickler sich zunächst sträuben, es „offiziell" anzuwenden.

Diesen Nachteilen stehen folgende Vorteile gegenüber:

- Die Umsetzung einer schlichten System-Architektur ist weniger zeitaufwändig.
- Die System-Architektur bleibt übersichtlich, weil unnötige Komplexität vermieden wird.
- Die Erweiterung einer schlichten System-Architektur ist in der Regel weniger aufwändig als der nachträgliche Umbau einer ungeeigneten System-Architektur.

Im Projekt Netlife Finance Suite beschlossen die Entwickler, den System-Entwurf nicht vollständig während der Implementierungsphase durchzuführen. Zunächst erfolgte die Festlegung der von den Entwicklern als sehr hilfreich empfundenen System-Metapher.[4] Anschließend wurde ein Grobentwurf des Systems mit Hilfe von *Class Responsibility Cards* vorgenommen.[5] Auf eine graphische Darstellung des Systems wurde hingegen verzichet.

Die auf Basis der Class Responsibility Cards entstandene System-Architektur enthielt nur Komponenten, die für die Umsetzung der ersten Userstories erforderlich waren. Sie erwies sich im Laufe des Projekts als relativ stabil. Die hin und wieder durchgeführten Refactorings stellten entweder eine Erweiterung der ursprünglichen Architektur dar oder betrafen periphere Komponenten. Nach Einschätzung der beteiligten Entwickler stellen Class Responsibility Cards ein nützliches Entwurfsinstrument dar, das sie auch in Zukunft verwenden wollen.

[2] vgl. [?, S. 56]
[3] [?, S. 113]
[4] s. Abschnitt 7.2.1, S. 94
[5] s. Abschnitt 4.2.3, S. 57

5.1.2 Refactoring

Das Prinzip der schlichten System-Architektur ist bei XP untrennbar mit dem Prinzip des Refactoring verbunden. Refactoring meint die fortlaufende Anpassung der System-Architektur an die aktuellen Anforderungen.

Traditionelle Entwicklungsprozesse versuchen, die System-Architektur präventiv so zu gestalten, dass Anforderungs-Änderungen mit geringem Zeitaufwand und mit geringen Änderungen an den Quelltexten realisierbar sind. Zu diesem Zweck werden schon beim Entwurf Anstrengungen unternommen, das System so flexibel wie möglich zu gestalten:

- Klassenhierarchien aus der Begriffswelt des Anwendungsgebiets werden vervollständigt.
- Die Interfaces von Klassen werden ergänzt, um ihren Zuständigkeitsbereich vollständig abzudecken.
- Klassen erhalten Attribute, von denen angenommen wird, dass sie später benötigt werden.
- Methoden erhalten Parameter, von denen angenommen wird, dass sie später benötigt werden.

Häufig werden diese Elemente jedoch nicht benötigt oder müssen vor ihrer ersten Verwendung angepasst werden. Die Zeit für ihre Konstruktion wäre also besser anderweitig genutzt worden. Darüber hinaus erhöhen sie unnötigerweise die Komplexität des Systems und erschweren damit seine Weiterentwicklung.

Um diese Nachteile zu vermeiden, wird bei XP versucht, die Umsetzung und sogar die Deklaration aktuell nicht benötigter Systemelemente grundsätzlich zu vermeiden. Dafür wird in Kauf genommen, dass vor der Umsetzung einer Userstory unter Umständen eine Anpassung bzw. Erweiterung der aktuellen System-Architektur erfolgen muss. Dieser Arbeitsschritt wird unterstützt durch das Vorhandensein von automatisierten Tests, mit denen die Integrität der neuen System-Architektur verifiziert werden kann. Die Nachteile dieser Vorgehensweise ähneln denen beim initialen System-Entwurf:

- Die Einschätzung, wann ein grundlegender System-Umbau erforderlich ist, kann nur von erfahreren Entwicklern vorgenommen werden.
- Das Vorgehen steht im Widerspruch zur gängigen Lehrmeinung, der zufolge man bei der Software-Entwicklung möglichst vorausschauend handeln sollte.

Die Vorteile von schlichten Designs in Verbindung mit Refactorings wurden schon beim initialen System-Entwurf erörtert.

5.1.3 Risikoeinschätzung

XP geht von der optimistischen Annahme aus, dass eine erweiterbare Architektur, mit der sich die tatsächlichen Anforderungen an ein System erfüllen lassen, ohne aufwändige Analyse und eigene System-Entwurfsphase geschaffen werden kann. Wegen der erwähnten Vorteile wird bewusst das Risiko eingegangen, dass diese Annahme für das aktuelle Projekt ausnahmsweise nicht gilt.

Diesem Risiko gegenüber steht nämlich das Risiko, durch eine zu aufwändige System-Architektur Zeit zu verlieren. Letzteres Risiko ist nach Erfahrung von Kent Beck höher zu veranschlagen.[6]

Ob der Einsatz von XP in einem bestimmten Projekt sinnvoll ist, hängt also auch davon ab, wie obiges Risiko eingeschätzt wird. Steht die Konstruktion eines sehr komplexen Systems an, so ist es höher zu veranschlagen als bei Systemen, die einfachen Anforderungen genügen müssen und/oder in ähnlicher Form bereits einmal realisiert worden sind.

5.1.4 Umsetzung in der Praxis

Obwohl die Art des System-Entwurfs von XP vor allem für die Entwicklung von Anwendungen und weniger für die Entwicklung von Produkten oder Frameworks angemessen erscheint, hat sie sich im Projekt Netlife Finance Suite als geeignet erwiesen. Insbesondere die strikte Orientierung an den aktuellen Anforderungen erwies sich dabei als hilfreich.

Als Methode für die Motivierung von Entwicklern wird vielfach die Betonung eines Qualitätskults empfohlen. Beziehen die Entwickler diesen auf einzelne Komponenten wie Klassen, Subsysteme und ähnliche Dinge, so kann leicht ein Widerspruch zu den Entwurfsprinzipien von XP entstehen, da diese Komponenten häufig „unvollendet“ belassen werden müssen. Stattdessen ist in einem XP-Projekt die Übertragung des Qualitätsgedankens auf das Gesamtsystem bzw. das Gesamtprojekt erforderlich: das Interesse der Entwickler ist dann nicht mehr auf die „Veredelung“ einzelner Komponenten sondern auf die Weiterentwicklung des Gesamtsystems gerichtet.

5.2 Release-Planung

Die Release-Planung von XP dient dazu, die Anforderungen des Auftraggebers mit den Bedürfnissen der Entwickler in Einklang zu bringen. Dazu müssen die vier voneinander abhängigen Variablen

- Zeit
- Kosten
- Umfang
- Qualität

ausbalanciert werden. In der Vergangenheit wurde gewöhnlich dem Auftraggeber gestattet, Vorgaben für alle vier Variablen zu machen. Wegen der gegenseitigen Abhängigkeit führte dies in der Regel dazu, dass der Vorgabewert für die am schwierigsten messbare Variable – Qualität – verfehlt wurde. Die Praxis hat jedoch gezeigt, dass der Auftraggeber im Zweifelsfall eher bereit ist, Einschränkungen im Umfang als bei der Qualität zu tolerieren.[7] Dieser Umstand und die Abhängigkeiten zwischen den Variablen werden bei der Release-Planung von XP im so genannten *Planungsspiel* berücksichtigt.

[6] vgl. [?, S. 110]
[7] vgl. [?, S. 15 ff.]

Das Planungsspiel dient dazu, in der *Erforschungsphase* neue Anforderungen an das System zu definieren, in der *Verpflichtungsphase* den Umfang der nächsten System-Version in einem Release-Plan festzulegen und in der *Steuerungsphase* den Fortschritt bei der Implementierung der neuen Systemfunktionen zu überwachen und ggf. Plankorrekturen vorzunehmen.

Ein XP-Projekt läuft in verschachtelten Zyklen ab. Der längste Zyklus ist der Release-Zyklus, der jeweils mit der Fertigstellung einer neuen System-Version abgeschlossen wird. Kent Beck empfiehlt als Dauer für einen Release-Zyklus einen Zeitraum von ein bis zwei Monaten. Ein Release-Zyklus enthält mehrere Iterations-Zyklen, die jeweils eine bis vier Wochen dauern.[8]

In der Planung eines Releasezyklus wird durch die Fokussierung auf die neuen Systemfunktionen die Sicht des Auftraggebers eingenommen. Dagegen steht bei der Planung eines Iterations-Zyklus mit der Definition von Aufgabenpaketen die Perspektive der Entwickler im Vordergrund. Wie Kent Beck und Martin Fowler ausführen, kann die Rolle des Auftraggebers in einem XP-Projekt durchaus von einem Mitarbeiter des beauftragten Software-Hauses wahrgenommen werden.[9] Wichtig ist jedoch, dass für den Betreffenden statt technischer Einzelheiten der Nutzwert des zu konstruierenden Systems im Mittelpunkt steht.

5.2.1 Erforschungsphase

In der Erforschungsphase definiert der Auftraggeber neue Anforderungen an das System und notiert diese in Form von Userstories. Dabei wird er vom Entwicklerteam unterstützt.

Userstories

Eine Userstory ist die kurz und prägnant formulierte Beschreibung einer Interaktion zwischen Benutzer und System. Im Unterschied zu den aus dem *Rational Unified Process* bekannten *Use Cases* wird eine Userstory nicht so detailliert ausformuliert, dass sie als Spezifikation für eine Implementierung ausreicht. Sie stellt vielmehr eine Notiz dar, die an die erforderliche Systemfunktion erinnert. Die Entwickler klären die für die Implementierung erforderlichen Details mit dem Auftraggeber erst, wenn die Umsetzung der Userstory ansteht. Für die damit verbundene Zeitersparnis sind unter Umständen folgende Nachteile in Kauf zu nehmen:

- Ist der Kunde nicht permanent vor Ort für die Entwickler ansprechbar, so kann es zu zeitlichen Verzögerungen kommen.
- Bei thematisch verwandten Userstories können evtl. Synergieeffekte durch gemeinsam verwendbare Komponenten nicht realisiert werden.

Das Verfahren nach dieser Regel führte im Netlife Finance Suite Project zur beabsichtigten Zeitersparnis ohne dass die erwähnten Nachteile zum Tragen kamen.

[8] Eine Befragung von 37 Entwicklerteams ergab, dass 70% von ihnen entweder zwei- oder dreiwöchige Releasezyklen verwenden, vgl. [?, S. 77]

[9] [?, S. 17 f.]

Aufwandsabschätzung

Der zweite Schritt bei der Erfassung einer Userstory ist die Schätzung des Aufwands für ihre Realisierung. Sie wird grundsätzlich von den Entwicklern vorgenommen. Als Einheit dienen dabei so genannte *ideale Entwicklertage*. Ein idealer Entwicklertag ist ein Tag, an dem ein Entwickler sich acht Stunden lang der Implementierung einer Userstory ohne Unterbrechung durch Telefonate, Besprechungen und dergleichen zuwenden kann.

Um zu betonen, dass es sich um eine sehr grobe Schätzung handelt, empfehlen manche Autoren, zur Abschätzung von Userstories stattdessen ideale Entwicklerwochen[10] oder sogar nur einen Schwierigkeitsgrad auf einer Skala von eins bis drei[11] zu verwenden. Gleichzeitig werden die Entwickler hierdurch darauf hingewiesen, dass sie auch bei der Abschätzung keine detaillierte technische Planung durchführen sollen – dies steht erst bei der Iterationsplanung an.

Im Projekt Netlife Finance Suite entschied sich das Entwicklerteam zunächst für die Abschätzung der Userstories in Entwicklerwochen. Es kam jedoch immer wieder zu Diskrepanzen zwischen diesen sehr groben Schätzungen und den später vorgenommenen Schätzungen auf der Basis von kleinen Arbeitspaketen (Tasks). Aus diesem Grund wurde schließlich eine Userstory zur Aufwandsabschätzung grundsätzlich in Tasks aufgeteilt und der Gesamtaufwand durch Aufsummierung der Aufwändsschätzungen für die Tasks bestimmt.

Spike Solutions

Lässt sich eine Userstory auf Grund mangelnden Detailwissens nicht hinreichend genau abschätzen, so empfiehlt sich nach XP die Implementierung einer *Spike Solution,* also die schnelle Umsetzung einer Lösung, die alle kritischen technischen Fragen adressiert. Eine Spike Solution ist als Element einer Risikomanagementstrategie zu betrachten. Es gilt abzuwägen, ob das Risiko einer Fehleinschätzung den Aufwand für die Durchführung einer Spike Solution rechtfertigt. Dabei ist außerdem zu berücksichtigen, dass die in der Spike Solution gefundenen Lösungsansätze in vielen Fällen für die Implementierung des eigentlichen Sytems verwendet werden können.

Im Projekt Netlife Finance Suite wurden zu Beginn drei Spike Solutions erstellt, die in relativ kurzer Zeit Fragen zu Architekturdetails beantworteten. Die dabei konstruierten Lösungen konnten zum überwiegenden Teil entweder unverändert oder mit leichten Modifikationen in das System integriert werden.

Aufteilung von Userstories

Ergibt die Abschätzung einer Userstory, dass sie nicht innerhalb eines Releasezyklus implementiert werden kann, so muss sie aufgeteilt werden. Dabei muss die Regel beachtet werden, dass die resultierenden Userstories immer noch Systemfunktionen beschreiben, die aus der Sicht des Benutzers einen Mehrwert gegenüber den aktuellen Systemfunktionen bieten. Dies hat zur Folge, dass bei der Aufteilung einer Userstory zunächst immer die Benutzerschnittstelle vor der Realisierung einer internen Funktion umgesetzt wird. Hiermit wird den Entwicklern indirekt eine einfache Form von Prototyping nahegelegt. Vor- und Nachteile dieser Vorgehensweise lassen sich folgendermaßen skizzieren.

[10] vgl. [?, S. 58]
[11] vgl. [?, S. 39]

- Vorteile

    1. Vor der zeitaufwändigen Realisierung einer Funktion erhalten die Entwickler ein Feedback von der Benutzerseite.
    2. Der Nutzen von komplexen Systemfunktionen kann vor ihrer vollständigen Umsetzung besser evaluiert werden.
    3. Die Entwickler erlangen ein besseres Verständnis für eine Systemfunktion vor ihrer Implementierung.

- Nachteile

    1. Nach der Aufteilung einer Userstory muss bei der Umsetzung für das erste Release ein gewisser Teil des zukünftigen Systems mit Mock-Ups simuliert werden. Deren Implementierung kostet Zeit, die ohne die Aufteilung der Userstory nicht aufgewendet werden müsste.
    2. Die zweckmäßige Aufteilung einer Userstory erfordert Zeit und setzt Erfahrung voraus.

Die im Projekt Netlife Finance Suite gemachten Erfahrungen deuten darauf hin, dass die zur Abschätzung und Aufteilung von Userstories notwendigen Lernprozesse von erfahrenen Entwicklern in wenigen Tagen absolviert werden können.

5.2.2 Verpflichtungsphase

In der Verpflichtungsphase werden die für das nächste Release in Frage kommenden Userstories vom Auftraggeber priorisiert, ausgewählt und in einen Release-Plan übertragen. Aus diesem geht hervor, zu welchem Zeitpunkt eine Userstory implementiert sein muss. Dabei muss der Auftraggeber allerdings eine wesentliche Bedingung akzeptieren: Das Entwicklerteam gibt eine obere Schranke für die Zahl der idealen Entwicklerstunden pro Iteration vor. Der Auftraggeber kann nur entscheiden, ob er

- das Release zu einem bestimmten Termin benötigt und deshalb ggf. Einschränkungen am Umfang vornimmt oder ob er
- den Umfang der neuen System-Version fest vorgibt und den erwarteten Fertigstellungstermin vom Entwicklerteam berechnen lässt.

Als Basis zur Berechnung der erwähnten Schranke dient der Wert der idealen Entwicklerstunden aus vorangegangenen Iterationen, er wird mit der Zahl der Iterationen für das aktuelle Release multipliziert. Für die Berechnung der idealen Entwicklerstunden pro Iteration gibt es verschiedene Empfehlungen, sie werden in Abschnitt 5.2.3 erörtert.

Im Projekt Netlife Finance Suite entschied sich der Auftraggeber für die Einhaltung eines festen Zeitrahmens. Ein Release-Zyklus umfasste vier Iterations-Zyklen zu je drei Wochen. Bereits die erste System-Version war vollständig lauffähig und konnte zu Demonstrationen verwendet werden. Auch die nachfolgenden Versionen waren jeweils stabil und lieferten gegenüber ihren Vorgängern einen für den Nutzer sichtbaren Mehrwert.

Der Zeitplan für die Umsetzung der Userstories konnte in jedem Release-Zyklus eingehalten werden. In den ersten zwei Release-Zyklen stellte sich heraus, dass das Team die Userstories zu pessimistisch eingeschätzt hatte und die ausgewählten Userstories bereits zu Beginn der vierten Iteration implementiert hatte. Ab der dritten System-Version wurden die Schätzungen genauer. Zu optimistische Schätzungen wurden nicht vorgenommen. Es kann somit resümiert werden, dass das Hauptziel des Release-Management von XP in vollem Umfang erreicht wurde.

5.2.3 Steuerungsphase

Der Release-Planungzyklus orientiert sich an der Sicht des Auftraggebers, indem er den Begriff der Userstory in den Vordergrund stellt. Der Iterationsplanungszyklus orientiert sich dagegen an der Sicht der Entwickler. Die vom Auftraggeber mit der höchsten Priorität versehenen Userstories werden zu Beginn einer Iteration auf der Basis technischer Kriterien in *Tasks* genannte Teilaufgaben aufgeteilt, die jeweils in höchstens drei idealen Entwicklertagen abgeschlossen werden können. Anschließend wird die Verantwortung für die Realisierung der einzelnen Tasks unter den Entwicklern aufgeteilt.

Die Länge einer Iteration wird zu Beginn eines Projekts festgelegt und dann nicht mehr verändert. Sie beträgt in der Regel eine bis vier Wochen.[12] Die Anzahl der Iterationen pro Release-Zyklus kann variieren und ergibt sich aus den vom Auftraggeber ausgewählten Userstories für das Release. Übliche Werte liegen zwischen zwei und vier Iterationen pro Release.

Berechnung der Project Velocity

Ähnlich wie bei der Release-Planung darf das Team Tasks nur in einem Umfang übernehmen, der erfahrungsgemäß tatsächlich innerhalb des Iterations-Zyklus implementiert werden kann. Der entsprechende Grenzwert wird initial geschätzt und im Laufe des Projekts aus Erfahrungswerten berechnet, die in vorangegangenen Iterationen ermittelten wurden. Es gibt allerdings unterschiedliche Empfehlungen, wie dabei genau zu verfahren ist.

- Kent Beck schlägt vor, für jeden Entwickler einen individuellen Belastungsfaktor als Verhältnis der gesamten Arbeitszeit zur effektiv für die Programmierung aufgewendeten Zeit zu berechnen. Nach Verteilung der Verantwortlichkeit für die einzelnen Userstories auf die Entwickler multipliziert jeder Entwickler die ihm zugeteilten idealen Entwicklerstunden mit seinem individuellen Belastungsfaktor und vergleicht das Ergebnis mit der verfügbaren Zeit für die Iteration. Ist der Wert zu hoch, so wird zunächst versucht, durch Verschiebung der Userstories zwischen den Entwicklern eine akzeptable Gesamtbelastung zu erreichen. Gelingt dies nicht, so muss das Team in Abstimmung mit dem Auftraggeber Userstories aussortieren, die in der aktuellen Iteration nicht mehr umgesetzt werden können.[13]

- In einem später erschienenen Buch empfiehlt Kent Beck als Co-Autor von Martin Fowler die Berechung einer so genannten *Project Velocity* für das

[12] vgl. [?, S. 91]
[13] vgl. [?, S. 93]

gesamte Entwicklerteam als Summe der im letzten Iterations-Zyklus umgesetzten idealen Entwicklerstunden.[14] Die Project Velocity stellt damit direkt die oben erwähnte Schranke dar.

- Don Wells empfiehlt ebenfalls die Berechung der Project Velocity für das gesamte Entwicklerteam.[15] Bemerkenswert ist sein Vorschlag, zur Berechnung der umgesetzten Entwicklerstunden die ursprünglichen Schätzwerte und nicht die effektiv aufgewendeten Zeiten zu verwenden. Nach Wells' Ansicht werden auf diese Weise zu optimistische oder pessimistische Schätzungen über die Project Velocity korrigiert.

- Auch Ron Jeffries empfiehlt die Berechung der Project Velocity für das gesamte Team.[16] Für die Berechnung eines initialen Werts gibt er einen Belastungsfaktor von drei an.[17] Ob für die Berechnung der Project Velocity des vorangegangenen Iterations-Zyklus effektive oder geschätzte Werte zu verwenden sind, wird von Jeffries nicht näher angegeben.

Alle Berechnungsmodelle passen sich durch die Übernahme von Effektivwerten aus vorangegangenen Iterationen im Laufe des Projekts an die realen Verhältnisse an. Unterschiede zwischen den Modellen und durch initiale Schätzwerte bedingte Abweichungen gleichen sich deshalb innerhalb weniger Iterations-Zyklen aus.

Im Projekt Netlife Finance Suite übernahm das Team das Berechnungsmodell von Beck und Fowler. Für den ersten Iterations-Zyklus wurde entsprechend der Empfehlung von Jeffries ein Belastungsfaktor von drei angenommen. Dieser Faktor sank in den nächsten Iterationen auf zwei und stieg für zwei Iterationen wieder auf drei an, als das Team aufgestockt wurde.

Plankorrektur

In der Mitte eines Iterations-Zyklus werden die bisher erzielten Fortschritte bei der Umsetzung der Tasks von einem Teammitglied erfasst und mit den ursprünglichen Schätzungen verglichen. Ergeben sich nennenswerte Unterschiede, muss das Entwicklerteam in Zusammenarbeit mit dem Auftraggeber entweder eingeplante Tasks streichen oder zusätzliche Tasks übernehmen. Gegebenenfalls ist daraufhin der Release-Plan anzupassen.

Wie bereits an anderer Stelle erwähnt erwiesen sich im Projekt Netlife Finance Suite die ersten Schätzungen der Entwickler als zu pessimistisch. Dies führte dazu, dass das Team gelegentlich in der zweiten Woche einer Iteration zusätzliche Tasks übernahm. Die Anpassung des Release-Plans erfolgte dann jeweils zu Beginn des nächsten Release-Zyklus.

Aufgabenverteilung

Die von Kent Beck und anderen Autoren vorgeschlagene Art der Verteilung der Tasks auf die Entwickler erwies sich im Projekt Netlife Finance Suite als umständlich. Die Ursache hierfür waren die unterschiedlichen Erfahrungen, Programmierfähigkeiten und Belastungen der Entwickler. Es kam regelmäßig vor,

[14] vgl. [?, S.59]
[15] vgl. [?, www.extremeprogramming.org/rules/velocity.html]
[16] vgl. [?, S.179]
[17] vgl. [?, S. 181]

dass einige Entwickler bereits gegen Mitte des Iterations-Zyklus die ihnen zugeteilten Tasks abgearbeitet hatten. Auf die Berechung eines individuellen Belastungsfaktors wurde jedoch aus folgenden Gründen verzichtet:

- Es sollte der Eindruck vermieden werden, dass mit dem neuen Entwicklungsprozess ein System der individuellen Bewertung eingeführt wird.
- Der Einfluss der Kombination von Entwicklern beim Pair Programming lässt einen festen individuellen Belastungsfaktor fragwürdig erscheinen.
- Die Verteilung der Tasks zwischen den Entwicklern auf der Iterationsplanungs-Sitzung wird komplizierter.

Stattdessen wurde ab der vierten Iteration bei der Verteilung der Tasks folgendermaßen verfahren:[18]

- Es wurde wie bisher eine Project Velocity für das gesamte Entwicklerteam berechnet. Sie bildete die Beschränkung für die Tasks, die der Auftraggeber in der Iterationsplanungs-Sitzung auswählen konnte.
- Die Aufteilung von Userstories inkl. einer groben Abschätzung nahmen jeweils zwei Entwickler gemeinsam vor.
- Auf der Iterationsplanungs-Sitzung wählte der Auftraggeber zunächst eine Menge favorisierter Userstories ohne Rücksicht auf den erforderlichen Realisierungsaufwand aus.
- Daraufhin überprüfte und korrigierte das gesamte Team die Einschätzung der zugehörigen Tasks.
- Der Auftraggeber wählte auf Basis der korrigierten Schätzungen eine Menge von Userstories aus, die in der aktuellen Iteration umzusetzen war.
- Die Karten der Tasks für die aktuelle Iteration wurden nach der Iterationsplanungs-Sitzung in einem frei zugänglichen Karteikasten deponiert. Jeweils zwei Entwickler suchten sich gemeinsam eine Karte aus und implementierten die zugehörigen Tasks gemeinsam.
- Nach Erledigung einer Task wurde die tatsächlich benötigte Zeit auf der zugehörigen Karteikarte erfasst und diese in einem eigens dafür vorgesehenen Kasten abgelegt.

Etwa einen Monat später erfolgte nochmals eine leichte Modifikation. Die Karteikarten der ausgewählten Tasks wurden fortan nach der Iterationssitzung in ein webbasiertes Aufgabenverwaltungssystem namens *Request-Tracker* übertragen, das per Intranet auf jedem Arbeitsrechner verfügbar war. Pro Task wurden mit diesem System neben der tatsächlich aufgewendeten Zeit ggf. Abschätzungs-Korrekturen und Kommentare erfasst.

Im Vergleich zur ursprünglichen Iterationsplanung war die Benennung eines Verantwortlichen für jede in einer Iteration geplante Task bei dieser Vorgehensweise nicht mehr gegeben. Dennoch erfolgte die Umsetzung der Tasks zügig und es kam während der gesamten Projektlaufzeit zu keinerlei Stockungen und nennenswerten zeitlichen Verzögerungen.

[18] Die Anregung hierzu kam in einem Gespräch von Dr. Martin Lippert vom Arbeitsbereich Softwaretechnik am Fachbereich Informatik der Universität Hamburg.

Die Release-Planung von XP im Vergleich

Es lässt sich unabhängig vom Modus der Aufgabenverteilung feststellen, dass die Release-Planung von XP sich weniger aufwändig gestaltet als in traditionellen Entwicklungsprozessen, wo zur Aufstellung eines Projektplans gewöhnlich ein Netzplan entworfen werden muss, um über die Berechnung von kritischen Pfaden einen Fertigstellungstermin angeben zu können.[19] Begünstigt durch die Maßnahmen zur Vermeidung von Wissensinseln erlaubt die Release-Planung von XP dagegen eine relativ zuverlässige Projektplanung, die auch beim unvorhergesehenen Ausfall einzelner Entwickler in der Regel gültig bleibt.

Unterschiede zwischen Releaseversion und Iterationsversion

Da eine umfangreiche Userstory unter Umständen nicht im Laufe einer einzigen Iteration implementiert werden kann, ist das System am Ende einer Iteration für den Auftraggeber nicht zwangsläufig von größerem Nutzen als zu Beginn der Iteration – auch wenn dies angestrebt werden sollte. Das Entwicklerteam muss jedoch sicherstellen, dass die aktuelle System-Version am Ende einer Iteration stabil ist und ohne weitere Modifikationen in Betrieb gehen könnte.

Diese Anforderung wurde im Netlife Finance Suite Projekt bis auf zwei Ausnahmen erfüllt. Die Ausnahmen waren jeweils auf unzureichend getestete Komponenten zurückzuführen, lagen also nicht originär in der Iterationsplanung begründet.

5.3 Arbeitsorganisation

Die Aufgabe der Entwickler in einem XP-Projekt besteht zum überwiegenden Teil aus der Implementierung der übernommenen Tasks. Dazu bilden die Entwickler Paare, deren Zusammensetzung je nach Aufgabe wechselt.

Der erste Schritt bei der Umsetzung einer Task ist die Beseitigung von Unklarheiten in der Spezifikation der zugehörigen Userstory. Dies erledigen die Entwickler in Zusammenarbeit mit dem Auftraggeber.

Im nächsten Schritt prüfen die Entwickler, ob die System-Architektur alle erforderlichen Eigenschaften aufweist oder ob sie eventuell vor Implementierung der benötigten Funktionen per Refactoring verändert werden muss. Dabei konsultieren sie gegebenenfalls andere Entwickler mit speziellen Kenntnissen der betroffenen Teilsysteme.

Der dritte Schritt besteht in der Programmierung der erforderlichen Komponenten. Hierbei arbeiten die Entwickler paarweise nach gewissen Regeln an einem Arbeitsrechner.

Ist eine Task fertig implementiert und getestet, so erfolgt die Integration der neuen bzw. veränderten Komponenten in das Quelltext-Versionsverwaltungs-System, das immer eine stabile System-Version enthält. Abschließend erfolgt die Zeiterfassung für die abgeschlossene Task und das Übernehmen einer neuen Task.

[19] s. Balzert

5.3.1 Kooperation

XP lebt davon, dass alle Beteiligten und insbesondere die Entwickler den Willen zur konstruktiven Kooperation aufbringen.

- Der Auftraggeber erarbeitet gemeinsam mit den Entwicklern neue Systemanforderungen.
- Auftraggeber und Entwickler entwerfen gemeinsam nach bestem Wissen einen Release-Plan.
- Die Entwickler implementieren paarweise die Tasks einer Userstory.
- Die Entwickler stehen sich gegenseitig für Rückfragen zur Verfügung und bieten sich gegenseitig Hilfe bei Problemen an.

Es ist Aufgabe des so genannten *Coaches*, allen Projektmitarbeitern die Notwendigkeit dieser konstruktiven Kooperation vor Augen zu führen und Problemsituationen idealerweise bereits im Anfangsstadium zu erkennen und zu vermeiden.

Mangelnde Kooperationsbereitschaft

Die Ursachen für mangelnde Kooperationsbereitschaft sind vielfältig und können hier nicht umfassend erörtert werden. Beispielhaft seien genannt:

- Ein Entwickler fühlt sich beim Pair Programming beobachtet.
- Ein Entwickler ist es nicht gewohnt, seine Programme mündlich zu erläutern und bevorzugt deshalb die Erstellung schriftlicher Dokumentation.
- Ein Entwickler möchte sein Spezialwissen nicht weitergeben, um sich selbst für den Arbeitgeber unentbehrlich zu machen.

In diesen Fällen muss der Coach versuchen, die Betroffenen für die Mitarbeit im Team zu gewinnen. Gelingt dies nicht, so ist ihr Ausschluss aus dem Entwicklerteam zu erwägen.

Im Projekt Netlife Finance Suite wurde das Entwicklerteam nach einem Monat um drei neue Entwickler aufgestockt, von denen zwei aus unterschiedlichen Gründen nicht in das Team integriert werden konnten.

Im ersten Fall handelte es sich um einen ausländischen Studenten mit Teilzeitvertrag. Seine Integration gelang nicht, weil seine Sprachkenntnisse eine effiziente Kommunikation in den Teammeetings nicht zuließen und weil der so erworbene Informationsrückstand durch die Teilzeittätigkeit noch verstärkt wurde.

Im zweiten Fall ergab sich aus Gesprächen mit dem Betroffenen, dass er eine vollständig andere Arbeitsweise gewohnt war als sie bei XP propagiert wird. Bei der Einarbeitung in das Projekt beklagte er den Mangel an schriftlicher Dokumentation. Dennoch vermied er es, sich das System von erfahrenen Entwicklern mündlich erklären zu lassen. Es wurde der Versuch unternommen, ihm kleine und überschaubare Teilaufgaben zu übertragen, die ohne umfangreiches Vorwissen erledigt werden konnten. Da er diese auch mit gelegentlicher Hilfestellung nicht lösen konnte, erfolgte das ultimative Angebot, das Wissen über

die System-Architektur als Pair-Programming-Partner der erfahrenen Entwickler zu erwerben. Als der Betreffende dieses Angebot nicht wahrnahm, sondern stattdessen versuchte die ihm ursprünglich übertragenen Aufgaben im Alleingang zu lösen, erfolgte der Ausschluss aus dem Entwicklerteam.

Einarbeitung neuer Mitarbeiter

Die Einarbeitung neuer Mitarbeiter in das Entwicklerteam der Netlife Finance Suite geschah in drei Schritten.

- Zu Beginn erfolgte in einer jeweils etwa zweistündigen Schulung eine Einführung in den Entwicklungsprozess und eine Vorstellung der System-Architektur.
- Anschließend erhielten die neuen Mitarbeiter ein schriftliches Dokument, das ihnen die System-Architektur detaillierter erläuterte und die Einrichtung der Entwicklungsumgebung beschrieb.
- In der Folge wurden die neuen Mitarbeiter bei der Aufgabenverteilung wie erfahrene Entwickler behandelt. Sie setzten die Tasks grundsätzlich gemeinsam mit einem erfahrenen Entwickler um, der ihnen währenddessen alle erforderlichen Einzelheiten der System-Architektur erklärte.
- Bei der Berechnung der Project Velocity wurden sie zunächst nicht berücksichtigt, da ihre Produktivität nur gering war. Ein Anstieg ihrer Produktivität machte sich schließlich in einer höheren Project Velocity bemerkbar.

Auf diese Weise gelang inzwischen die Integration von vier Mitarbeitern in das Entwickler-Team der Netlife Finance Suite.

Aufgaben des Projektleiters

XP überträgt allen Entwicklern des Teams ein hohes Maß an Verantwortung bei der Abschätzung von Userstories und beim Entwurf der System-Architektur. Damit wird eine Voraussetzung dafür geschaffen, dass die Entwickler ein Interesse am Fortschritt des Projekts haben. Ist dieser Idealzustand erreicht, so stellt sich die Frage, ob das Team sich von nun an selbst verwalten und auf einen Coach beziehungsweise einen Teammanager verzichten kann. Dies hätte unter anderem den Vorteil, dass ein krankheitsbedingter Ausfall des Teammanagers keine zeitlichen Verzögerungen mehr verursachen könnte.

Beobachtungen aus dem Projekt Netlife Finance Suite zeigen jedoch, dass ohne einen Teammanager wesentliche Aufgaben nicht mehr wahrgenommen werden. Das Projekt wurde etwa sechs Monate von einem Teammanager geleitet, der neben seiner Entwicklertätigkeit folgende Aufgaben wahrnahm:

- Vorbereitung der Release- und Iterationsplanungs-Sitzungen
- Kommunikation mit dem Auftraggeber und mit den Nutzern des Systems
- Überwachung des Teams hinsichtlich der Einhaltung des Entwurfsprozess-Prinzipien
- Einberufung der morgendlichen Standup-Meetings

- Einberufung von Diskussionsrunden bei wichtigen Entscheidungen über die System-Architektur.

Der Teammanager verließ das Team nach Fertigstellung des zweiten Release, stand aber noch für weitere drei Monate als Berater zur Verfügung. Relativ bald stellte sich heraus, dass einige Aufgaben nicht mehr wahrgenommen wurden.

- Die morgendlichen Standup-Meetings fanden nur noch sporadisch statt, obwohl alle Teammitglieder die dort ausgetauschen Informationen für wichtig hielten.
- Probleme, für die keine einfache Lösung existierte, blieben unbearbeitet.
- Kritische Entscheidungen wurden vertagt.

Zur Vermeidung dieser Misssände wurde die Rolle eines *Teamcoaches* geschaffen, die von einem Teammitglied im täglichen Wechsel bekleidet wurde. Die Aufgaben des Teamcoaches entsprachen denen des Teammanagers, mit Ausnahme der Vorbereitung der Release- und Iterationsplanungs-Sitzungen.

Nach Einführung des Teamcoaches fanden die Standup-Meetings wieder regelmäßig statt und elementare Entscheidungen wurden wieder getroffen. Insgesamt bewerten die Teammitglieder den Teamcoach-Modus jedoch zwiespältig und bejahen die Frage, ob ein Teammanager erforderlich ist.[20]

Aus diesen Erfahrungen muss geschlossen werden, dass auf die Rolle eines Teammanagers auch in einem XP-Projekt nicht verzichtet werden kann. Dies deckt sich mit den Empfehlungen der meisten Autoren. Kent Becks Empfehlung kann stellvertretend für die anderen Autoren angeführt werden. Er empfiehlt die Unterscheidung der beiden Rollen *Coach* und *Terminmanager*.[21] Der Coach hat folgende Aufgaben:

- Unterstützung von Entwicklern als Partner bei schwierigen technischen Aufgaben
- Erkennung von Refactoring-Zielen und Förderung von Refactorings
- Erklärung des Prozesses gegenüber dem Management
- Permanente Motivierung der Teammitglieder.

Der Terminmanager ist zuständig für die Erfassung des Projektfortschritts und die Anpassung der Projektpläne. Beide Rollen können in kleineren Projekten von einer einzelnen Person übernommen werden.

Charakteristisch für die Rolle des Coaches ist, dass er verantwortlich für den „Rest“ der Probleme ist, die nicht explizit durch XP-Methoden behandelt werden. Diese treten immer wieder auf, lassen sich aber wegen ihrer Vielschichtigkeit nicht näher spezifizieren. Die Abspaltung der Rolle des Terminmanagers, dessen Aufgabenbereich klar umrissen ist und deshalb leicht delegiert werden kann, ist deshalb konsequent und sinnvoll.

[20] s. Kap. 7, S. 99
[21] [?, S. 73 f.]

5.3.2 Arbeitsbedingungen

Aus der Betonung der direkten mündlichen Kommunikation ergeben sich in einem XP-Projekt besondere Anforderungen an die räumlichen und zeitlichen Arbeitsbedingungen.

Arbeitsräume

Alle Entwickler des Teams sollten spontan mit jedem Teamkollegen ein direktes Zwiegespräch führen können. Dazu ist es erforderlich, dass die räumlichen Distanzen zwischen den Arbeitsplätzen nicht zu groß sind. Andererseits muss darauf geachtet werden, dass die Entwickler sich durch die beim Pair Programming üblichen Diskussionen nicht gegenseitig stören.

Ein Mensch benötigt erfahrungsgemäß etwa 15 Minuten, bevor er bei der Bewältigung einer anspruchsvollen Aufgabe aktiv wird. Dieser Umstand macht eine gewisse Rücksichtnahme bei der Kommunikation der Entwickler-Paare untereinander erforderlich. Es ist deshalb vorteilhaft, wenn die Entwickler per Sichtkontakt feststellen können, ob ein bestimmter Ansprechpartner momentan in eine Aufgabe vertieft ist. In diesem Fall sollte abgewogen werden, ob eine sofortige Antwort benötigt wird.

Kent Beck und Ron Jeffries begründen ihre Empfehlung für Großraumbüros damit, dass sie die oben skizzierten Anforderungen am besten erfüllen. Es muss jedoch angenommen werden, dass diese nicht für alle Software-Projekte verfügbar sind. Desweiteren darf bezweifelt werden, dass der Lärmpegel bei größeren Entwicklerteams nicht als störend empfunden wird, auch wenn er durch den Lärmpegel der Arbeitsplatzrechner relativiert wird.

Im Netlife Finance Suite-Projekt standen den Entwicklern vier benachbarte Büroräume mit jeweils zwei Arbeitsplätzen zur Verfügung. Jeder Arbeitsplatz besaß eine große Arbeitsplatte und war mit einem Arbeitsrechner ausgestattet. Außerdem befand sich in jedem Raum ein Whiteboard.

Die Entwickler empfanden diese Arbeitsumgebung als zweckmäßig, sie erfüllte im Wesentlichen die obigen Anforderungen. Aufgrund der akustischen Verhältnisse entschieden sich die Entwickler in der Regel dagegen, mit zwei Paaren in einem Raum zu arbeiten. Die Kontaktaufnahme zwischen zwei Entwicklerpaaren machte es deshalb in der Regel erforderlich, den eigenen Arbeitsplatz zu verlassen und in ein benachbartes Büro hinüber zu gehen. Dies wurde jedoch von keinem der Entwickler als unangenehm empfunden.

Arbeitszeiten

Da die Entwickler den überwiegenden Teil ihrer Arbeitszeit mit der Programmierung in Paaren und mit Diskussionen untereinander verbringen ist die Einhaltung einer Kernarbeitszeit unverzichtbar. Dabei muss auch beachtet werden, dass Überstunden über einen längeren Zeitraum kontraproduktiv sind. Dies gilt insbesondere, wenn Entwickler auf Grund von Nachtschichten nicht an den morgendlichen Standup-Meetings teilnehmen. Der Teammanager sollte deshalb darauf achten, dass die Entwickler nur in seltenen Ausnahmefällen Überstunden leisten.

Im Netlife Finance Suite Projekt gelang es dem Team auf Grund der relativ präzisen Release- und Iterationsplanung, die Notwendigkeit für Überstunden zu vermeiden.

5.3.3 Pair Programming

Die Umsetzung einer Task erfolgt jeweils durch zwei Entwickler. Sie prüfen gemeinsam, ob ein Umbau der System-Architektur erforderlich ist und diskutieren, wie sie grundsätzlich vorgehen wollen. Anschließend setzen sie sich zusammen an einen Arbeitsrechner und besprechen fortlaufend die nächsten Arbeitschritte. Während der an der Rechnertastatur sitzende Entwickler (der „Fahrer“) sich auf das Schreiben des Programmtextes konzentriert, achtet der neben ihm Sitzende (der „Beifahrer“) auf Programmierfehler und stellt strategische Überlegungen an.

Diese unter dem Namen Pair Programming bekannte Arbeitsweise versucht einige Probleme der klassischen Form der Programmierung zu vermeiden, bei der die Entwickler einzeln programmieren:

- Beim Entwurf von Schnittstellen wird technischen Aspekten mehr Beachtung geschenkt als Verständlichkeit, Einfachheit und ausreichender Dokumentation.
- Programmier-Richtlinien und -Methoden werden nicht konsequent beachtet, insbesondere wenn die Entwicklung einer Komponente nicht so geradlinig verläuft wie ursprünglich angenommen.
- In einer Erforschungsphase erstellte provisorische Lösungen werden aus Versehen oder Bequemlichkeit in das Quelltext-Versionsverwaltungs-System übernommen.
- Die Entwickler erweitern ihre Kenntnisse von Programmiersprachen und Programmbibliotheken nicht mehr während der täglichen Arbeit, sondern bestenfalls in Schulungen oder in ihrer Freizeit.

Kritiker bemängeln an Pair Programming vor allem, dass die Zahl der erzeugten Codezeilen pro Zeiteinheit trotz der doppelten Personalkosten nicht höher ist als bei der klassischen Programmierung. Untersuchungen belegen jedoch, dass die höhere Qualität der erstellten Programme geringere Wartungskosten zur Folge hat und damit die höheren Kosten für die Implementierung im Laufe der System-Lebenszeit ausgleicht und sogar überkompensiert.[22]

Durch Befragungen und empirische Untersuchungen wurde festgestellt, dass durch Pair Programming die oben angeführten Probleme in der Regel vermieden bzw. entschärft werden. Darüber hinaus können folgende Vorteile konstatiert werden:

- Die Teambildung und die Erhaltung des Teamgeists lässt sich durch Pair Programming fördern.[23]
- Durch die bessere Verteilung des Wissens über die System-Architektur lässt sich das Risiko signifikant senken, das mit dem Ausfall einzelner Mitarbeiter verbunden ist.[24]

[22] vgl. [?, S. 3 f.]
[23] vgl. [?, S. 8]
[24] vgl. [?, S. 8 f.]

Die im Projekt Netlife Finance Suite gemachten Beobachtungen und die Aussagen der Entwickler im Interview bestätigen diese Angaben[25]. Auf eine detaillierte Untersuchung des Einflusses auf die Produktivität musste mangels Vergleichsdaten allerdings verzichtet werden. Im Einzelnen wurden folgende Erfahrungen gemacht:

- Die Gewöhnung der Entwickler an die Arbeitsweise beim Pair Programming dauert etwa eine Woche. In dieser Zeit lernen sie ihre Kritik konstruktiv zu formulieren und die Arbeit zwischen Fahrer und Beifahrer sinnvoll aufzuteilen.
- Vielen Entwicklern vermittelt Pair Programming ein größeres Gefühl der Zufriedenheit mit ihrer Arbeit und den erzielten Ergebnissen. Frustrationen im Fall von Misserfolgen treten seltener auf.
- Einige Entwickler haben Vorbehalte gegenüber Pair Programming, weil sie sich durch einen Partner beobachtet und kontrolliert fühlen. Diese Aussagen sind jedoch bisher nur dokumentiert von Entwicklern, die bisher noch keine Erfahrung mit Pair Programming gesammelt haben.
- Sehr gute Entwickler haben die Meinung geäußert, dass sie sich durch den Partner beim Pair Programming behindert fühlen. Ein homogenes Entwicklerteam kann deshalb von Vorteil sein.
- Der Lerneffekt betrifft vor allem das jeweilige System und weniger die allgemeinen Programmierkenntnisse.
- Das Weiterreichen der Tastatur an den Programmierpartner („Fahrerwechsel“) setzt voraus, dass alle Entwickler den gleichen Texteditor verwenden. Dies war im Netlife Finance Suite Projekt nicht der Fall, weshalb der Fahrerwechsel praktisch nie stattfand. Dies wurde allerdings von den Entwicklern nicht als Manko empfunden.

Die Entwickler sind jedoch der Ansicht, dass Pair Programming nicht bei allen Programmieraufgaben sinnvoll ist. Als Ausnahmen wurden im Interview kleinere, klar spezifizierte Reparaturarbeiten sowie Routineaufgaben genannt. Für sehr empfehlenswert bzw. unerlässlich wird Pair Programming dagegen bei allen Aufgaben angesehen, die eine Änderung bzw. Erweiterung der System-Architektur bedingen.

5.3.4 Programmierstandards

Die gemeinsame Verantwortung *aller* Entwickler des Teams für *alle* Teile des Systems wird durch die Einhaltung von Richtlinien bei der Programmierung erleichtert bzw. ermöglicht. Diese Richtlinien betreffen

- die Formatierung des Quelltextes
- die Dokumentation von Klassen und Methoden
- die Gestaltung von Schnittstellen

[25] s. Abschnitt 7.5.1, S. 98

- die Verwendung von Entwurfsmustern
- Art und Umfang von Unittests
- Arbeitsmethoden wie z.B. den Test-First-Ansatz.

Es ist Aufgabe des Teammanagers, die Entwickler zu Beginn des Projekts auf die zu beachtenden Richtlinien hinzuweisen und fortlaufend stichprobenartig zu kontrollieren, ob die Richtlinien tatsächlich eingehalten werden. Werden dabei große Abweichungen festgestellt, so sollte dies als alarmierendes Signal verstanden und sofortige Gegenmaßnahmen eingeleitet werden. Der Grund hierfür ist, dass die Arbeitsmoral der meisten Entwickler von der Einschätzung des Systems abhängt, das sie konstruieren bzw. weiterentwickeln sollen. Wird dieses als minderwertig eingestuft, so wird auch bei der Erweiterung weniger sorgfältig vorgegangen, was wiederum einen negativen Einfluss auf die Gesamtqualität des Systems hat.

Zur Vermeidung dieser Abwärtsspirale hat sich die Einführung und Erhaltung eines *Qualitätskults* bewährt. Wichtige Bedingungen für die Ausbildung eines Qualitätskults sind die ständige Motivation der Entwickler und der Hinweis, dass im Zweifel die Qualität einer Komponente den Vorrang vor einer Erweiterung ihres Funktionsumfangs oder einer pünktlichen Fertigstellung hat.

Wird bei einer Stichprobe die grobe Verletzung der Programmier-Richtlinien festgestellt, sollte zunächst Ursachenforschung betrieben werden, statt die Entwickler nur an ihre Pflichten zu erinnern. Gegebenenfalls ist eine Anpassung der Richtlinien sinnvoller, als die Entwickler zur stupiden Durchführung von Maßnahmen zu verpflichten, die im aktuellen Projekt nur von geringem Nutzen sind.

Die Programmier-Richtlinien für das Projekt Netlife Finance Suite sind bereits in Kapitel 4 aufgeführt worden. Als hilfreich erwies sich insbesondere ein Werkzeug zur Code-Formatierung, weil durch seine Verwendung unnötige Diskussionen über Konventionen der Klammersetzung und dergleichen vermieden werden konnten.

Es gibt eine Reihe von sehr unterschiedlichen Schreibweisen für Quelltexte in imperativen Programmiersprachen. Untersuchungen haben ergeben, dass mehrere davon mit etwas Gewöhnung gut lesbar sind, dass aber der unmittelbare Wechsel von einer Schreibweise zur nächsten sehr verwirrend ist. Dies bestätigte sich im Projekt Netlife Finance Suite. Der zu Beginn des Projekts definierte einheitliche Schreibstil war für einige Entwickler zunächst ungewohnt, wurde aber bereits nach kurzer Zeit allgemein akzeptiert.

Als hilfreich zur Etablierung eines Qualitätskults wurde von den Entwicklern auch die Konvention angesehen, Collective Code Ownership anzustreben. Hierdurch wurde die gemeinsame Verantwortung der Entwickler für das Gesamtsystem betont und die Vereinheitlichung der Quelltexte gefördert.

Der Test-First-Ansatz

Es wird vielfach empfohlen, die Tests für eine Komponente vor deren Implementierung zu spezifizieren und codieren. Die Implementierung der Komponente ist genau dann abgeschlossen, wenn sie die zugehörigen Tests erfolgreich absolviert. Diese *Test-First* genannte Arbeitsmethode weist folgende Vorteile auf:

- Die Entwickler überlegen sich vor der Implementierung einer Komponente ein sinnvolles Interface.
- Die Komponente wird nicht mit Eigenschaften und Methoden überfrachtet, die aktuell nicht benötigt werden.
- Die Entwickler erhalten zuverlässig einen Hinweis, wenn eine komplexe Komponente noch nicht vollständig implementiert ist.
- Das Schreiben der Tests für die Komponente wird nicht aus Nachlässigkeit unterlassen.

Die Entwickler im Projekt Netlife Finance Suite haben unterschiedliche Erfahrungen mit dem Test-First-Ansatz gemacht. Die überwiegende Mehrheit findet den Ansatz in der Theorie gut, wendet ihn in der Praxis aber nicht konsequent an. Als Gründe wurden angegeben:

- Ein geeignetes Interface für eine Komponente ist selten offensichtlich. Muss ein Interface aber nachträglich angepasst werden, so fällt beim Test-First-Ansatz zusätzliche Arbeit an.
- Eine komplexe Komponente kann oft nicht innerhalb eines Arbeitstages vollständig implementiert werden. Da aber täglich mindestens einmal eine Integration durchzuführen ist, müssten Tests deaktiviert werden, die noch nicht implementierte Eigenschaften betreffen.

Ein Entwickler hat allerdings sehr positive Erfahrungen mit dem Test-First-Ansatz gemacht und wendet ihn konsequent an. Ob die restlichen Entwickler noch einmal aufgefordert werden, ebenfalls nach dem Test-First-Ansatz zu verfahren, ist noch nicht entschieden.

5.3.5 Fortlaufende Integration

Die von den Entwicklern vorgenommenen Änderungen an den Quelltexten werden nach Abschluss einer Arbeitseinheit unverzüglich in das Quelltext-Versionsverwaltungs-System übernommen. Dies hat den Vorteil, dass Probleme mit parallel vorgenommen Änderungen an einer bestimmten Komponente unverzüglich erkannt und behoben werden können.

Kent Beck schlägt vor, einen bestimmten Arbeitsrechner für das Integrieren zu verwenden. Auf diese Weise kann vermieden werden, dass zwei oder mehr Entwicklerpaare das Hauptsystem in einen instabilen Zustand versetzen, weil sie gleichzeitig ihre Änderungen vornehmen. Nachteilig ist jedoch, dass dieser Rechner nicht mehr sinnvoll für andere Zwecke genutzt werden kann und dass gegebenenfalls Wartezeiten entstehen.

Im Projekt Netlife Finance Suite wurde den Entwicklern erlaubt, die im Versionsverwaltungssystem CVS vorhandenen Schutzmechanismen zu benutzen, um beim Integrieren instabile Zustände zu vermeiden. Dabei waren folgende Regeln zu beachten:

1. Beim Start der Integration muss die lokale Kopie des Systems alle Tests fehlerfrei absolvieren.

2. Alle lokalen Änderungen gegenüber der Masterversion im Versionsverwaltungssystem müssen in einem atomaren Arbeitsschritt integriert werden.

3. Schlägt die Integration fehl, so müssen zunächst alle Änderungen, die sich zwischenzeitlich an der Masterversion ergeben haben, in die lokale Kopie übernommen werden. Anschließend ist wieder bei Schritt 1 fortzufahren.

Auf diese Weise konnten instabile Zustände der Masterversion weitgehend vermieden werden. Der Einsatz von Werkzeugen, die die oben beschriebenen Schritte inklusive der Testdurchläufe automatisiert durchführen, wurde erwogen aber mangels Dringlichkeit bisher nicht in die Praxis umgesetzt.

5.4 Qualitätssicherung

Bei klassischen Prozessen sind Qualitätssicherungs-Maßnahmen in der Regel erst für eine relativ späte Projektphase bzw. Entwicklungsphase einer Komponente vorgesehen. Dies hat zur Folge, dass viele Fehler nicht von den Entwicklern sondern von Mitarbeitern der so genannten Qualitätssicherungs-Abteilung gefunden werden und dann an die Entwickler zurück übermittelt werden müssen.

Um diese zeitaufreibende Art der Fehlerbehebung möglichst selten erforderlich zu machen, schreiben die Entwickler in einem XP-Projekt für jede System-Komponente automatisierte Tests, bevor sie implementiert wird. Diese Tests werden in die System-Testsuite aufgenommen und regelmäßig aufgerufen. Auf diese Weise erhalten die Entwickler sofort einen Hinweis, wenn sie versehentlich bei der Implementierung einer neuen Komponente das Verhalten einer bestehenden Komponente in unzulässiger Weise verändert haben. Voraussetzung für diese Arbeitsweise ist, dass die Kompilierung der aktuellen System-Version und das Durchlaufen der System-Testsuite nur wenige Minuten in Anspruch nimmt.

Neben so genannten *Unittests*, die die korrekte Funktionsweise von einzelnen System-Komponenten abfragen, enthält eine System-Testsuite auch *Akzeptanztests*, die das Zusammenspiel von System-Komponenten testen. Die Aktzeptanztests werden vom Auftraggeber als Black-Box-Tests[26] entworfen und bilden für die Entwickler die exakte Spezifikation einer Userstory. Eine Userstory ist genau dann im Sinne des Auftraggebers umgesetzt, wenn alle zugehörigen Akzeptanztests fehlerfrei absolviert werden. Durch Einsatz eines geeigneten Werkzeuges ist es deshalb ohne besonderen Aufwand möglich, den aktuellen Projektstatus anhand der bereits absolvierten Akzeptanztests zu ermitteln und gegebenenfalls firmenintern zu publizieren.

5.4.1 Rolle der Qualitätssicherungs-Abteilung

Es wird kontrovers diskutiert, ob die klassische Rolle der so genannten Qualitätssicherungs-Abteilung in einem XP-Projekt verändert werden sollte. Die ursprüngliche Aufgabe der Qualitätssicherung besteht darin, für ein System anhand der Systemspezifikation Tests zu entwerfen, während die Entwickler das System auf Basis derselben Spezifikation konstruieren. In einer späteren Projektphase wird das System von Mitarbeitern der Qualitätssicherungs-Abteilung mit den vorbereiteten Tests geprüft und die aufgetretenen Fehler werden an die Entwickler gemeldet.

[26] Ein *Black-Box-Test* für ein System wird ohne Kenntnis der Systeminterna entworfen.

Da sie die von der Qualitätssicherungs-Abteilung entworfenen Tests nicht kennen, soll mit dieser Form der Arbeitsteilung der Eifer und die Sorgfalt der Entwickler gesteigert werden. In der Praxis weist diese Vorgehensweise jedoch einige Nachteile auf.

- Eine nicht vollständig formalisierte System-Spezifikation wird in der Regel von der Qualitätssicherungs-Abteilung anders interpretiert als von den Entwicklern. Die Unterschiede in der Interpretation werden bei der oben skizzierten klassischen Arbeitsteilung erst relativ spät erkannt. Die erforderlichen Korrekturen sind deshalb relativ teuer.

- Bei Änderungen der Spezifikation ist sowohl auf seiten der Qualitätssicherungs-Abteilung wie auch auf Entwicklerseite eine Anpassung in der jeweiligen formalisierten Fassung der Spezifikation vorzunehmen, das Fehlerrisiko bei diesem Schritt wird verdoppelt.

In einem XP-Projekt bilden die Akzeptanztests die einzige gültige Spezifikation. Damit werden die mit der doppelten Formalisierung der Spezifikation verbundenen Kosten vermieden. Die Aufgabe der Qualitätssicherungs-Abteilung besteht darin, die Akzeptanztests zusammen mit dem Auftraggeber zu spezifizieren und gegebenenfalls zu codieren. Die Durchführung der Akzeptanztests obliegt den Entwicklern.

Im Projekt Netlife Finance Suite war eine derartige Einbindung der Qualitätssicherungs-Abteilung nicht vorgesehen, da es sich um ein firmeninternes Pilotprojekt handelte. Die Akzeptanztests wurden von den Entwicklern nach Rücksprache mit dem Auftraggeber entworfen und codiert.

5.4.2 Erfahrungen mit automatisierten Tests

Im Interview erklärten alle Entwickler automatisierte Tests im Allgemeinen und Unittests im Besonderen als sehr hilfreich. Einschränkend fügten manche allerdings hinzu, dass die richtige Verwendung von automatisierten Tests einen Lernprozess erfordert.

- Ein positives Ergebnis beim Durchlaufen der System-Testsuite bedeutet nicht, dass das System fehlerfrei ist, sondern dass die vorhandenen Tests keinen Fehler aufgedeckt haben. Der subtile Unterschied zwischen beiden Aussagen kann zwar theoretisch vermittelt werden, wird von den Entwicklern jedoch erst verinnerlicht, wenn die Testsuite fehlerfrei absolviert wird und das System dennoch nachweislich einen Fehler enthält. Es ist deshalb empfehlenswert, gelegentlich stichprobenartige Plausibilitätskontrollen durchzuführen.

- Die Wertschätzung für automatisierte Tests steigt, je öfter sie fehlschlagen. Um diese Erfahrung zu machen, müssen die Entwickler das System jedoch zunächst verändern.

- Ein Systemfehler, der nicht durch die Unittests sondern auf andere Weise entdeckt worden ist, lässt zunächst das Vertrauen der Entwickler in die System-Testsuite sinken. Erst wenn konsequent jeder derartige Fehler mit einem Unittest abgesichert worden ist, wird deren Wert erkannt.

- Bei manchen Komponenten ist ein automatisierter Test scheinbar nicht möglich, beispielsweise weil die Infrastruktur der Komponente zustandsbehaftet ist und keine beliebige Wiederholung von Testläufen erlaubt. Die Praxis zeigt jedoch, dass sich die Infrastruktur in diesem Fall häufig mit geringem Aufwand als Mock-Up mit den für automatisierte Tests erforderlichen Eigenschaften implementieren lässt.

Gelegentlich wird gegen die ausschließliche Spezifikation eines System über automatisierte Tests das Argument vorgebracht, die Entwickler würden dann „nur“ noch darauf bedacht sein, ein System zu konstruieren, das diese Akzeptanztests erfolgreich absolviert. Hinter dieser Kritik verbirgt sich offensichtlich die Vorstellung, mit nicht automatisierten Tests mehr oder andere Fehler zu entdecken als mit automatisierten Tests. Es ist jedoch nicht nachvollziehbar, warum die entsprechenden Tests nicht ebenfalls automatisiert werden können.

Zusammenfassend lässt sich sagen, dass automatisierte Tests heute bereits in großem Umfang eingesetzt werden und vielerorts als unverzichtbar angesehen werden. Dennoch werden sie noch nicht so konsequent verwendet, wie es eigentlich wünschenswert wäre. Der Grund hierfür ist, dass die mit automatisierten Tests verbundenen Vorteile erst für denjenigen offensichtlich werden, der sie in der Praxis verwendet hat.

5.5 Dokumentation

Kent Beck ist der Ansicht, dass der Quelltext die wichtigste Dokumentation eines Systems darstellt und allen anderen Dokumentationsformen überlegen ist, weil er immer den aktuellen Status des Systems widerspiegelt.[27] Zur Verbesserung des Dokumentationscharakters von Quelltexten sind vor allem Refactorings durchzuführen, um die System-Architektur zu vereinfachen und die Benennung von Klassen und Methoden sprechender zu machen. Bei Verständnisproblemen sind gegebenenfalls die ursprünglichen Autoren zu befragen.

Auch wenn nur wenige Entwickler diesen radikalen Standpunkt teilen, so kann er doch als Anregung verstanden werden, den Wert von Dokumentation neu zu überdenken.

5.5.1 Quelltextkommentare

Im Projekt Netlife Finance Suite waren die Entwickler gehalten, das Konzept einer Klasse, den Zweck jeder Methode und ggf. die Bedeutung ihrer Parameter formalisiert so zu dokumentieren, dass aus diesen Angaben mit Hilfe eines Tools eine vollständige Schnittstellendokumentation der Klasse erstellt werden konnte. Diese Aufgabe wurde erleichtert durch ein Tool zur standardisierten Quelltext-Formatierung. Es erzeugte leere, aber formal korrekte Kommentar-Rümpfe vor Klassen- und Methoden-Deklarationen.

Ausnahmen von der Pflicht zur Dokumentation waren nur erlaubt bei Klassen, Methoden und Parametern, die nach Einschätzung der Entwickler so sprechend benannt waren, dass sich ein weiterer Kommentar erübrigte. Durch diese Regel sollte die Wahl von ausdrucksstarken Bezeichnern und die Konstruktion von kurzen Methoden mit klar spezifiziertem Zweck gefördert werden.

[27] vgl. [?, S. 44 f. sowie S. 156]

Nach Ansicht der Entwickler ist die Dokumentation der Netlife Finance Suite in relativ gutem Zustand und hilfreich bei der Weiterentwicklung des Systems. Die meisten Methoden sind mit einem Kommentar versehen, allerdings ist dieser an manchen Stellen bei nachträglichen Änderungen nicht aktualisiert worden.

Als vermuteter Grund für nicht vorhandene oder veraltete Quelltextkommentare wurde von den Entwicklern im Netlife Finance Suite Projekt vor allem Nachlässigkeit genannt. Dass ein Entwickler keine Kommentare schreibt, weil er sie in der Vergangenheit als nicht hilfreich kennengelernt hat, wurde dagegen als Grund nahezu ausgeschlossen. Andererseits werden Quelltextkommentare nur in Ausnahmefällen tatsächlich als hilfreich empfunden.

Es lässt sich also feststellen, dass

1. Kent Becks Empfehlung der gängigen Praxis entspricht.
2. Die meisten Entwickler diesen Zustand jedoch beklagenswert finden.
3. Nur die wenigsten Entwickler konsequent ihren eigenen Quelltext kommentieren.

Die von Kent Beck propagierte Vorgehensweise erscheint angesichts dieser Widersprüche zumindest als der Weg des geringsten Übels. Denn er spart die Zeit ein, die Entwickler für das Schreiben von nicht verwendeten Kommentaren aufbringen. Wird dagegen Wert auf tatsächlich nutzbare Quelltextkommentare gelegt, so erscheint eine akribische Kontrolle der Entwickler unvermeidlich.

5.5.2 Dokumentation der System-Architektur

Entwickler, die mit den Kernkomponenten der System-Architektur vertraut sind, brauchen für die Durchführung von Refactorings

- einen gut strukturierten oder gut dokumentierten Quelltext
- gegebenenfalls die Auskunft der ursprünglichen Autoren.

Neue Teammitglieder können sich jedoch nur unter großer Mühe mit diesen Informationsquellen einen Überblick über die Architektur eines Systems verschaffen. Daneben wird gewöhnlich eine vereinfachte Darstellung der System-Architektur zu Präsentations- und Marketingzwecken benötigt.

In traditionellen Entwicklungsprozessen stehen als Basis für derartige Dokumente nach Abschluss der Entwurfsphase der Grob- bzw. Feinentwurf zur Verfügung. In einem XP-Projekt wird jedoch auf die Erstellung derartiger Dokumente gewöhnlich verzichtet. Erfahrungsgemäß ist ihr Wert nach dem Start der Implementierungsphase gering, weil sie die Architektur des realen Systems nicht adäquat darstellen.

Alistair Cockburn schlägt für Projekte mit leichtgewichtigen Entwicklungsprozessen vor, Dokumente erst bei Bedarf zu erstellen und ihre Erzeugung sowie ihre Aktualisierung als Userstory in die Release-Planung einzubeziehen. Der Informationsfluss erfolgt bei dieser Vorgehensweise also grundsätzlich vom realen System zum Dokument und nicht wie bei traditionellen Entwürfen in der umgekehrten Richtung. Damit wird unnötiger Arbeitsaufwand vermieden und andererseits eine Aktualität der Dokumente zum Zeitpunkt ihrer Nutzung sichergestellt. Zudem kann die Art der Systemdarstellung an den jeweiligen Adressaten angepasst werden.

Im Projekt Netlife Finance Suite wurde nach diesem Muster verfahren. Es entstand dabei ein Dokument, das sich in der ersten Version an neue Teammitglieder wendete und die grundlegende Architektur als Text mit Grafiken darstellte. Die grafischen Darstellungen konnten später für die Erstellung von Präsentationen vor dem Management verwendet werden. Eine Aktualisierung des etwa 30-seitigen Dokuments erfolgte immer dann, wenn die Einarbeitung neuer Teammitglieder anstand. Als hilfreich erwies sich außerdem ein Dokument, das die Antworten auf häufig gestellte Fragen sowie die technische Beschreibung von Arbeitsabläufen enthält.

5.6 Wissenstransfer

Collective Code Ownership bietet in einem XP-Projekt unter anderem Vorteile bei der Release-Planung, die sich sehr einfach gestaltet. Er setzt allerdings voraus, dass alle Entwickler mit den Kernkomponenten des Systems vertraut sind und dass ihr Kenntnisstand laufend aktualisiert wird. Dies wird durch verschiedene Maßnahmen erreicht.

- Allmorgendlich findet ein so genanntes *Standup-Meeting* statt, auf dem alle Entwickler kurz berichten, mit welcher Aufgabe sie gerade befasst sind und welche kürzlich durchgeführten Änderungen für alle anderen Entwickler von Bedeutung sind.
- Durch den regelmäßigen Wechsel der Partner beim Pair Programming wird der Austausch aktueller Neuerungen zusätzlich gefördert.
- Ein einfaches Design in Verbindung mit ständigen Refactorings stellt sicher, dass sich in die System-Architektur keine Notlösungen einschleichen, die ohne ausführliche Dokumentation nicht verständlich sind.
- Bei Unklarheiten geben die Entwickler sich gegenseitig Auskunft und leisten gegebenenfalls Hilfe. Durch die Teilnahme an den Standup-Meetings sind die Entwickler informiert darüber, welcher Entwickler an welcher Komponente die letzten Änderungen vorgenommen hat. Im Zweifelsfall ist diese Information auch über das Quelltext-Versionsverwaltungs-System abrufbar.

Die klassische Form des Wissenstransfers – die Kommunikation über schriftliche Dokumente wie Quelltextkommentare und Entwurfsdokumente – spielt dagegen in einem XP-Projekt nur eine untergeordnete Rolle. Gegenüber den obigen Maßnahmen hätte sie in der Theorie den Vorteil, dass die Weiterentwicklung des Systems von einem neuen Entwicklerteam übernommen werden könnte. Die Praxis zeigt jedoch, dass in der Regel die von den Entwicklern geschriebenen Quelltextkommentare und die Entwurfsdokumente alleine nicht ausreichen, um einem neuen Entwicklerteam die „Philosophie“ eines Systems zu vermitteln. Ohne diese kann jedoch ein komplexes System nicht verstanden oder gar weiterentwickelt werden.

Im Projekt Netlife Finance Suite gelang es mit den beschriebenen Maßnahmen die Kenntnisse der Entwickler auf aktuellem Stand zu halten. Bei der Durchführung der Standup-Meetings entwickelten sich des öfteren spontan Diskussionen über Lösungsansätze für aktuelle Probleme. In den Interviews wurde

dies bemängelt, weil diese Diskussionen oft nicht von allgemeinem Interesse waren, sondern nur einzelne Entwickler betrafen. Es wurde deshalb angeregt, dass ein Moderator derartige Diskussionen abbricht und den interessierten Entwicklern vorschlägt, die Diskussion unmittelbar nach dem Standup-Meeting fortzusetzen.

5.7 Mitarbeitermotivation

Neben den Kenntnissen und Fähigkeiten der Entwickler hängt die Güte einer System-Architektur von der Sorgfalt ab, mit der sie entworfen und weiterentwickelt wurde. Die Sorgfalt der Entwickler wird aber bewusst und unbewusst von ihrer Motivation beeinflusst. Es ist deshalb ratsam, die Motivation der Entwickler gezielt zu fördern und so genannte „Motivationskiller“ zu vermeiden. Als klassische Motivationskiller gelten

- die Pflicht zur Durchführung von Tätigkeiten, deren Nutzen anzweifelt wird
- die unnötige Bevormundung in technischen und organisatorischen Dingen
- die regelmäßige Ableistung von Überstunden
- unzureichende Arbeitsmittel bzw. Arbeitsplatzausstattung.

Die Motivation der Entwickler kann dagegen beispielsweise durch die Übertragung von Verantwortung und durch Etablierung eines so genannten Qualitätskults gefördert werden.

Bei XP wird die Motivation der Entwickler explizit durch die folgenden Regeln bzw. Vorgehensweisen berücksichtigt:

- Den Entwicklern wird die Zeit gegeben, Software mit hohen Qualitätsansprüchen zu entwickeln.
- Es wird auf die Erstellung von Zwischenprodukten verzichtet, deren Nutzen sich in der Praxis nicht erwiesen hat.
- Zur Vermeidung von Terminstress übernehmen die Entwickler bei der Release-Planung die Aufwandsabschätzung für Userstories und Tasks.
- Alle Entwickler übernehmen gemeinsam die Verantwortung für das Gelingen des Projekts.
- Die Entwickler werden ausdrücklich aufgefordert, nur in Ausnahmefällen Überstunden zu leisten.

Darüber hinaus hat der Teammanager die Aufgabe, die Motivation der Entwickler zu fördern bzw. zu erhalten. Zu diesem Zweck steht er als Ansprechpartner bei organisatorischen Problemen zur Verfügung. Am Ende einer Iteration gibt er den Entwicklern bei einer Besprechung die Gelegenheit, auf Mißstände hinzuweisen. Anschließend erörtert er gemeinsam mit ihnen Lösungen für die aufgetretenen Probleme.

Kapitel 6

Fazit und Ausblick

6.1 Fazit

Eine dieser Arbeit zu Grunde liegende Frage lautete: „Welche Anforderungen werden aus heutiger Sicht an einen Software-Entwicklungsprozess gestellt?“ Im Laufe der Untersuchungen stellte sich heraus, dass die Antwort auf diese Frage vielschichtiger ausfallen muss als ursprünglich angenommen. Wie namhafte Autoren seit einiger Zeit betonen, hängen die Anforderungen an einen Software-Entwicklungsprozess stark vom Projekt ab, in dem er verwendet werden soll.

6.1.1 Neue Trends bei Software-Entwicklungsprozessen

Es können jedoch in einzelnen Disziplinen wie System-Entwurf, Release-Management, Arbeitsorganisation usw. Trends identifiziert werden, die sich in neuen Entwicklungsprozessen niedergeschlagen haben und auch Auswirkungen auf etablierte Prozesse bzw. deren Inkarnationen haben. Beispielhaft sei in diesem Zusammenhang der Rational Unified Process genannt, für den in jüngster Zeit Vorschläge unterbreitet wurden, die auf eine Verschmelzung mit Extreme Programming abzielen.

Diese Trends sind zum einen auf veränderte Rahmenbedingungen durch technologische Neuerungen und wirtschaftliche Entwicklungen sowie auf die Defizite einzelner Methoden oder ganzer Entwicklungsprozesse zurückzuführen. Maßgeblichen Einfluss hatte auch die neue Einschätzung diverser Risiken, die bei der Software-Entwicklung einzugehen sind.

System-Entwurf

Wegen der Bedeutung der System-Architektur wurde ihrem detaillierten Entwurf vor der System-Implementierung traditionell eine wichtige Rolle eingeräumt. Bei der Implementierung erwies sich eine auf diese Weise erstellte System-Architektur allerdings oft als nicht praxistauglich.

Es hat sich deshalb der Trend etabliert, die Entwurfsphase abzukürzen und die System-Implementierung relativ früh mit einer flexiblen Basis-Architektur zu beginnen, die im Laufe des Projekts Schritt für Schritt erweitert wird.

Release-Management

Beim Release-Management galt lange eine detaillierte und vollständige Zeitplanung als erstrebenswert. Die zu erreichenden Ziele wurden von den Entwicklern definiert und waren deshalb für den Auftraggeber relativ abstrakt. In der Praxis führte ein derartiges Release-Management oft dazu, dass das konstruierte System nicht innerhalb des geplanten Zeitrahmens fertig gestellt wurde oder nicht den eigentlichen Anforderungen des Auftraggebers entsprach.

Beim Release-Management werden heute kurze Release-Zyklen von wenigen Monaten bevorzugt. Der Auftraggeber bestimmt, welche für ihn nutzbringenden Funktionen innerhalb dieses Zeitraums in das System integriert werden sollen. Die Zeitplanung ist realistisch, weil die Entwickler die Aufwands-Abschätzung für die erforderlichen Arbeitsschritte vornehmen. Vor dem Beginn des nächsten Release-Zyklus wird die aktuelle System-Version vom Auftraggeber eingehend begutachtet.

Arbeitsorganisation

Klassische Entwicklungsprozesse wurden in der Regel für Projekte aller Größenordnungen konzipiert. Sie sehen für ein Entwicklerteam mehrere Hierarchie-Ebenen und eine starre Rollenverteilung vor. Die Kommunikation in einer solchen Projektorganisation basiert auf schriftlichen Dokumenten und verläuft deshalb schleppend. Der Releaseplan ergibt sich durch die Berechnung von kritischen Pfaden in Netzplänen und wird obsolet, wenn einzelne Mitarbeiter mit Schlüsselqualifikationen nicht wie geplant zur Verfügung stehen.

In den letzten Jahren wurden für kleine bis mittlere Projekte angemessenere Organisationsformen gefunden. Dazu wurden gezielt die Vorteile von überschaubaren Entwicklerteams ohne feste Rollenverteilung genutzt. Mit der Betonung direkter mündlicher Kommunikation und der Vermeidung von Spezialisierung der Entwickler konnten die obigen Nachteile vermieden werden. Unterstützend wirkten dabei Methoden wie Pair Programming und Collective Code Ownership.

Qualitätssicherung

Organisierte Qualitätssicherung wurde lange als eine Art von Endkontrolle aufgefasst, die erst unmittelbar vor der Auslieferung eines Systems stattfand. Diese Vorgehensweise erwies sich als unzureichend, weil generelle Fehlentwicklungen zu spät aufgedeckt wurden und damit das Risiko von gravierenden Planabweichungen unzulässig anstieg.

Inzwischen wird die Qualitätssicherung als Bestandteil des Entwurfs- und Implementierungsprozesses verstanden. Sie bezieht sich nicht mehr ausschließlich auf das gesamte System, sondern verifiziert zunächst dessen Komponenten und erst anschließend deren Interaktion. Auf diese Weise werden Fehler früher erkannt und lassen sich schneller lokalisieren und beheben.

Eine wichtige Rolle kommt dabei automatisierten Tests zu. Da die Interpretation ihrer Resultate keinen zeitlichen Aufwand erfordert, können sie von Entwicklern durchgeführt werden und bilden zudem die Voraussetzung für Refactorings. Die Aufgabe der Qualitätssicherungs-Abteilung besteht nunmehr darin, die Kriterien für die automatisierten Tests zu definieren.

Dokumentation

Es galt lange Zeit als unverzichtbar, dass Entwickler die Architektur eines Systems und seiner Komponenten schriftlich dokumentieren. Bemerkenswerterweise geschah dies jedoch in vielen erfolgreich verlaufenen Projekten nur in rudimentärer Form. Die Rolle und der Nutzen schriftlicher Dokumentation, die während des Projekts von den Entwicklern anzufertigen ist, wird deshalb zur Zeit kontrovers diskutiert.

Ein relativ neuer Vorschlag zu diesem Thema stammt von Kent Beck. Er empfiehlt den weitgehenden Verzicht auf Dokumentation zugunsten von aussagekräftigen, überschaubaren Klassen-, Variablen- und Methoden-Bezeichnungen.

Auch wenn diese radikale Sicht nur von wenigen geteilt wird, ist dennoch der Trend zu verzeichnen, Dokumentation nur noch bei konkretem Bedarf zu erstellen. Dies ist einerseits okönomisch und bietet zum anderen den Vorteil einer größeren Aktualität der Dokumente.

Wissenstransfer

Die Übermittlung von elementarem Wissen über die System-Architektur erfolgte lange vornehmlich über schriftliche Dokumente, die jedoch nur sporadisch aktualisiert wurden. Als Nachteil erwies sich dabei auch, dass das Vorwissen der jeweiligen Leser nur schwer berücksichtigt werden konnte.

In manchen Projekten ist man inzwischen dazu übergegangen, das Wissen über die zentralen Komponenten eines Systems mündlich zu vermitteln und permanent auf Besprechungen zu aktualisieren. Diese Form des Wissenstransfers ist nur für kleinere Projekte bzw. Teams praktikabel, hat sich aber gerade hier bewährt.

Mitarbeitermotivation

In der Vergangenheit ist immer wieder versucht worden, durch Bürokratisierungsmaßnahmen die Arbeitsabläufe in Software-Firmen von Individuen unabhängig zu machen. Inzwischen wurde erkannt, dass viele dieser Maßnahmen die Motivation der Entwickler untergraben und darüber hinaus nur von geringem Nutzen sind.

In jüngster Zeit sind verstärkt Bestrebungen zu verzeichnen, die Motivation von Software-Entwicklern durch die Übertragung von Verantwortung, die Etablierung eines Qualitätskults und angemessene Arbeitszeit-Regelungen gezielt zu fördern.

6.1.2 Software-Firmen und Entwicklungsprozesse

Vielerorts wird das Ziel verfolgt, die einzelnen Aspekte der Software-Entwicklung innerhalb einer Firma oder Organisation so weit wie möglich durch die Vorgabe von Werkzeugen, Methoden und Prozessen zu vereinheitlichen. Diese Auffassung bildete die Basis für Prozess-Verbesserungsmodelle wie ISO 9000 und das Capability Maturity Model. Inzwischen wurde erkannt, dass die Kosten einer solchen Standardisierung ihren Nutzen oft bei weitem übersteigen.

Stattdessen erscheint es heute ratsamer, vor der Entwicklung eines Systems festzustellen, mit welchem Prozess es kostengünstig entwickelt werden kann. Zu

diesem Zweck sind eine Reihe von Faktoren wie die Größe und Zusammensetzung des Entwicklerteams, die Kritikalität des Systems, die Firmenkultur usw. zu berücksichtigen.

Der Gedanke eines firmenweit eingesetzten Entwicklungsprozesses ist damit jedoch keineswegs diskreditiert. Die Kosten für die Einführung von neuen Entwicklungsprozessen und die Ähnlichkeit vieler Projekte innerhalb kleinerer Software-Häuser lassen ihn nach wie vor sinnvoll erscheinen. Bei der Auswahl des Prozesses sollten jedoch die oben erwähnten Kriterien berücksichtigt werden.

6.1.3 Bewertung von Extreme Programming

Die zweite Ausgangsfrage für diese Arbeit lautete: „Inwiefern wird der leichtgewichtige Prozess Extreme Programming den aktuellen Anforderungen an Software-Entwicklungsprozesse gerecht?“ Wie bei der ersten Ausgangsfrage fällt die Antwort komplexer aus als ursprünglich angenommen. Leichtgewichtige Prozesse und Extreme Programming im Besonderen haben die oben aufgeführten Trends weitestgehend begründet und tragen damit zwangsläufig dem aktuellsten Kenntnisstand Rechnung. Wie sich aus der Antwort auf die erste Ausgangsfrage ergibt, bedeutet dies jedoch nicht, dass XP für alle aktuellen Software-Projekte geeignet wäre.

Zunächst lässt sich jedoch feststellen, dass XP sich im untersuchten Pilotprojekt Netlife Finance Suite bewährt hat:

- Der unorthodoxe System-Entwurf von XP lieferte eine stabile und flexible System-Architektur.
- Das XP-Release-Management erwies sich als einfach in der Durchführung und erlaubte eine zuverlässige Projektplanung.
- Die Arbeitsorganisation beruhte im Wesentlichen auf den XP-Methoden Pair Programming, Collective Code Ownership, Standup-Meeting und Fortlaufende Integration. Es traten dabei keine nennenswerten Probleme auf. Der Lernprozess für diese Methoden konnte in etwa einer Woche absolviert werden.
- Die Qualitätssicherung stützte sich auf Unit- und Akzeptanz-Tests. Diese wurden von allen Entwicklern als hilfreich empfunden und unterstützten die Refactorings wie vorgesehen.
- Die Erstellung von Dokumentation erfolgte nur bei Bedarf. Dieser bestand insbesondere bei der Einarbeitung von neuen Teammitgliedern.
- Der Wissenstransfer erfolgte im Wesentlichen auf der Basis mündlicher Kommunikation. Probleme ergaben sich bei Mitarbeitern, die auf Grund von Teilzeit-Arbeitsverträgen nicht an allen Teambesprechungen teilnehmen konnten.
- Die Mitarbeitermotivation konnte über einen Qualitätskult gefördert und erhalten werden.

Die im Laufe des Projekts aufgetretenen Probleme waren nicht gravierend und konnten in der Regel unverzüglich gelöst werden. Das Hauptziel von XP – die

fortlaufende Erstellung von stabilen Systemversionen mit jeweils gesteigertem Nutzwert – wurde in vollem Umfang erreicht.

Die Tatsache, dass ein einzelnes Projekt unter Verwendung von XP erfolgreich abgeschlossen wurde, ist statistisch nicht signifikant. Aber auch eine größere Stichprobenmenge ließe keine wirklich wertvolle Aussage über die allgemeine Eignung von XP zu. Zur Beurteilung von XP – und jedes anderen Entwicklungsprozesses – ist es vielmehr wichtig, die Bedingungen für die erfolgreiche Anwendung und die Gründe für gescheiterte Projekte zu untersuchen. Auf der Basis dieser Erkenntnisse lassen sich schließlich die Rahmenbedingungen für einen Prozess definieren. Diese lassen sich für XP derzeit wie folgt skizzieren:

- Das Entwicklerteam sollte aus höchstens zwölf Mitgliedern bestehen.
- Alle Entwickler arbeiten gleichzeitig in einem großen Raum.
- Die Entwickler müssen harmonisch kommunizieren und interagieren.
- Die Entwickler müssen persönlich Verantwortung für das gesamte Projekt übernehmen.
- Die Firmenleitung muss ein selbstbewusstes Entwicklerteam akzeptieren und unterstützen.

6.2 Ausblick

Angesichts der breiten Palette von Entwicklungsprozessen, die es inzwischen gibt, sind für die nächste Zukunft weniger revolutionäre Entwicklungen als vielmehr Neuerungen in einzelnen Teilbereichen des Projektmanagements zu erwarten.

6.2.1 Schriftliche Dokumentation

Jahrzehntelang galt als ehernes Gesetz, dass sämtliche Quelltexte sorgfältig zu dokumentieren sind. Die Frage nach dem Kosten-Nutzen-Verhältnis führte inzwischen dazu, dass die Entscheidung über die Notwendigkeit von Quelltextkommentaren in das Belieben der Entwickler gestellt wurde. Ähnliches gilt für die Dokumentation der System-Architektur. Dieser Ansatz weist jedoch zwei gewichtige Nachteile auf.

- Bei der Konstruktion eines komplexen Systems reicht das Gedächtnis der Entwickler unter Umständen nicht mehr für alle relevanten Informationen aus. Eine nachträgliche Dokumentation gestaltet sich in der Regel schwierig.
- Es wird eventuell die Chance vertan, die dokumentationswürdigen Schnittstellen eines Systems frühzeitig zu identifizieren und damit kostengünstig eine übersichtlichere Architektur zu erhalten.

Es wäre deshalb wünschenswert, einen breit akzeptierten Kanon von einfachen Richtlinien für die Dokumentation von Systemen zu etablieren, der einerseits flexibel ist und andererseits die obigen Nachteile vermeidet. In diesem Zusammenhang müsste auch untersucht werden, mit welchen Maßnahmen der Nutzen von Dokumenten erhöht werden kann, die von Entwicklern erstellt werden.

6.2.2 Leichtgewichtige Prozesse und große Projekte

Wegen der Fixierung auf homogene, intensiv kommunizierende Teams sind viele neuere Prozesse scheinbar nicht für größere Projekte geeignet, bei denen der Entwicklerstab deutlich mehr als zehn Mitglieder umfasst. Es ist jedoch ein naheliegender Gedanke, auch bei solchen Projekten die Vorteile leichtgewichtiger Entwicklungsprozesse zu nutzen, indem ein großer Entwicklerstab in mehrere kleinere Entwicklerteams aufgeteilt wird, die für die Entwicklung von Subsystemen zuständig sind.

Dabei tauchen allerdings Fragen auf, zu deren Beantwortung bisher nur wenig Erfahrungswerte vorliegen:

- Sollte die Aufteilung in Subteams temporär erfolgen oder ist sie besser fest für die gesamte Projektlaufzeit vorzunehmen?
- Wie verläuft die Integration der Subsysteme?
- Wer fungiert als Ansprechpartner des Auftraggebers?
- Wer ist verantwortlich für die Abschätzung von Aufgaben?
- Wie erfolgt die Release-Planung für das Gesamtprojekt?
- In welcher Form interagieren die Teams?

Es gibt bereits erprobte Lösungsansätze wie die Holistic Diversity Strategy oder Scrum. Sie sind zunächst auf ihre Allgemeingültigkeit bzw. ihre Grenzen zu überprüfen. Angesichts der Komplexität der Thematik darf überdies angenommen werden, dass weitaus mehr praktikable Lösungsansätze existieren. Diese gilt es ebenfalls experimentell zu untersuchen und zu dokumentieren.

Kapitel 7

Anhang A - Interview

Bei der Durchführung des halbstandardisierten Interviews mit den Teammitgliedern des Netlife Finance Suite Projekts wurde vom Interviewer jeweils der einleitende Abschnitt zu einem Themenkomplex sowie die erste Frage vorgelesen. Danach ergab sich in der Regel ein Dialog, in dessen Verlauf die restlichen Fragen sinngemäß angesprochen wurden. Der Verlauf des Interviews wurde handschriftlich protokolliert und unmittelbar anschließend in einen anonymisierten Fragebogen übertragen.

7.1 Release-Management

Das *Release-Management* legt die Zeitplanung bei der Umsetzung von neuen System-Funktionen und -Eigenschaften fest.

7.1.1 Fragen

Das Release-Management von XP sieht die Beteiligung der Entwickler bei der Abschätzung von Userstories vor. Die Alternative besteht in der Abschätzung von Arbeitsaufwänden durch den Projektleiter.

1. Welche Vorgehensweise finden Sie sinnvoller? Begründen Sie Ihre Ansicht.
   (a) Die Abschätzung durch die Entwickler ist sinnvoller, weil genauer (6)
2. Wird durch das XP-Release-Management Ihrer Meinung nach der Terminstress reduziert?
   (a) Ja, mit geringen Einschränkungen
   (b) Nur bedingt (5)
3. Wird durch das XP-Release-Management Ihrer Meinung nach die Planungssicherheit in IT-Projekten erhöht?
   (a) Ja (2)
   (b) Nein (4)

7.2 System-Entwurf

Beim *System-Entwurf* wird auf abstrakte Weise festgelegt, wie ein System realisiert werden soll. Dazu werden einzelne System-Komponenten identifiziert und ihre Verantwortlichkeiten festgelegt. Zwecks Vereinfachung der internen Kommunikation orientieren sich die Entwicklern dabei an bewährten Entwurfsmustern.

7.2.1 Fragen

XP empfiehlt, die meisten Entwurfsentscheidungen beim Pair Programming am Arbeitsrechner zu treffen.

1. Welche Erfahrungen haben Sie mit diesem Ansatz gemacht?
   (a) Funktioniert gut (3)
   (b) Zufriedenstellend; Nachteil: Grundlegende Alternativen werden nicht mehr erwogen
   (c) Funktioniert nicht gut
2. Was würden Sie am XP-Entwurfsprozess verbessern wollen?
   (a) Es sollte ein initialer Entwurf auf dem Papier stattfinden (4)
3. Wie hilfreich sind folgende Entwurfstechniken aus Ihrer Sicht:
   (a) CASE-Tools:
      i. Wertlos (2)
      ii. Kaum hilfreich
      iii. Wertvoll (2)
   (b) UML-Diagramme:
      i. Hilfreich (2)
      ii. Bedingt hilfreich (4)
   (c) Freie Skizzen auf Whiteboards:
      i. Sehr hilfreich (6)
   (d) Class Responsibility Cards:
      i. Hilfreich (3)
      ii. Bedingt Hilfreich (2)
   (e) System-Metaphern:
      i. Sehr hilfreich (6)
4. Wie schätzten Sie den Wert von Refactorings ein?
   (a) Sehr hoch (5)
   (b) Hoch

7.3 Qualitätssicherung

Eine geeignete *Qualitätssicherung* gewährleistet, dass jederzeit verlässlich überprüft werden kann, inwieweit ein System seiner Spezifikation entspricht.

7.3.1 Fragen

XP empfiehlt die ausgiebige Verwendung von Unittests und Akzeptanztests.

1. Welche Erfahrungen haben Sie mit automatisierten Tests gemacht?
   (a) Sehr gut (5)
   (b) Gut
2. Wie beurteilen Sie den Test-First-Ansatz? Wenden Sie ihn konsequent an?
   (a) Positive Beurteilung, wird konsequent angewandt.
   (b) Positive Beurteilung, wird häufig angewandt.
   (c) Ansatz ist in der Theorie sehr vernünftig. In der Praxis wird oft anders verfahren, weil noch eine Explorationsphase vorangeht (3)
   (d) Ansatz nur theoretisch interessant. Nicht in die Praxis umsetzbar.
3. Haben Unittests und Akzeptanztests Ihrer Erfahrung nach Dokumentationscharakter?
   (a) Ja, aber nur wenn sie kommentiert sind (4)
   (b) Eingeschränkt, Kommentare sind unverzichtbar
   (c) Nein
4. Spiegeln automatisierte Tests Ihrer Erfahrung nach ein falsches Bild wieder, weil sie lückenhaft sind?
   (a) Ja (3)
   (b) Nein (2)

7.4 Wissenstransfer

Wissenstransfer bezeichnet die Vermittlung von relevanten Wissensinhalten innerhalb des Entwicklerteams.

7.4.1 Fragen

XP veranschlagt den Wert schriftlicher Dokumentation eher gering.

1. Wie groß war für Sie der Wert der in Ihrem letzten Projekt erstellten Dokumentation?
   (a) Relativ hoch (aber nicht sehr umfangreich)
   (b) Gering (2)

(c) Sehr gering (3)

2. Was ist nach Ihrer Ansicht der Grund für fehlende, lückenhafte oder fehlerhafte Dokumentation?

   (a) Zeitdruck
      i. Nein (6)
   (b) Nachlässigkeit
      i. Ja (6)
   (c) Vermeidung von Redundanz bei „sprechenden“ Signaturen
      i. Ja (6)
   (d) Zu niedriges Kosten-Nutzen-Verhältnis
      i. Ja
      ii. Nein (5)
   (e) Überforderung der Programmierer wg. fehlender Schulung
      i. Ja (3)
      ii. Nein (3)

3. Wie könnte Ihrer Meinung nach der Wert der Dokumentation gesteigert werden?

   (a) Schulungen
      i. Ja (5)
      ii. Nein
   (b) Richtlinien und Beispiele
      i. Ja (5)
      ii. Nein
   (c) Unterstützung durch Tools
      i. Ja (5)
      ii. Nein (2)
   (d) weniger starre Regeln zur Vermeidung von Bürokratie
      i. Ja
      ii. Nein (5)
   (e) mehr mündliche Kommunikation
      i. Ja, mit Einschränkungen
      ii. Nein (5)

4. Welche Formen der Vermittlung von Wissen über den Aufbau eines Systems haben sich als hilfreich erwiesen?

   (a) Quelltexte
      i. Ja (2)

ii. Nein (4)

(b) Quelltextkommentare

i. Ja
ii. Nur bedingt
iii. Nur sehr bedingt (3)
iv. Nein

(c) Handbücher

i. Ja (5)
ii. Nein

(d) Schulungen

i. Ja (4)
ii. Nein (2)

(e) Persönliche Kommunikation

i. Ja (6)

5. Was sind Ihrer Erfahrung nach Beispiele für gute Dokumentation?

(a) JDK von Sun Microsystems

i. Ja (5)
ii. Nein

(b) Win32 von Microsoft

i. Ja
ii. Nein

(c) Weblogic-Doku von BEA

i. Ja (2)
ii. Nein (4)

(d) KDE

i. Ja

(e) QT

i. Ja

6. Wie können Ihrer Meinung nach neue Mitarbeiter am besten eingearbeitet werden?

(a) Schulung (4)
(b) Handbücher (6)
(c) Pair Programming (6)

7.5 Arbeitsorganisation

Arbeitsorganisation regelt die bei der Software-Entwicklung anfallenden Tätigkeiten wie Aufgabenverteilung, Koordinierung der Projektbeteiligten, Konfigurationsmanagement etc.

7.5.1 Fragen

XP empfiehlt, Programmcode grundsätzlich in Zweierteams an einem gemeinsamen Arbeitsrechner zu entwickeln (Pair Programming).

1. Welche Erfahrungen haben Sie mit Pair Programming gemacht?
   (a) Sehr gut
   (b) Gut (3)
   (c) Gut, aber ist anstrengend
   (d) Schlechte, weil zu einseitig
2. Sind Sie der Meinung, dass sich die Qualität der erstellten Software mit Pair Programming steigern lässt? Begründen Sie Ihre Ansicht.
   (a) Ja, wegen des besseren System-Entwurfs. (2)
   (b) Ja, weil weniger Flüchtigkeitsfehler gemacht werden.
   (c) Ja, weil die Programmierkonventionen besser eingehalten werden. (2)
3. Steigert Pair Programming die Produktivität? Begründen Sie ihre Ansicht.
   (a) Ja, wegen der höheren Qualität
   (b) Ja, weil weniger Flüchtigkeitsfehler unterlaufen (2)
   (c) Ja, weil wichtiges Programmier-Know-how transportiert wird
   (d) Ja, wenn das Umfeld stimmt
   (e) Nein
   (f) Nicht einschätzbar.
4. Haben Sie den Eindruck, dass die Qualität der erstellten Software durch die bei Meinungsverschiedenheiten erforderlichen Kompromisslösungen gemindert wird?
   (a) Ja (2)
   (b) Nein
   (c) Nein, im Gegenteil (2)
5. Bei welcher Art von Aufgabe würden Sie Pair Programming empfehlen, bei welcher nicht?
   (a) Empfehlenswert bei: Architekturentwurf und -änderung, Komplexe Aufgaben, Spike Solutions
   (b) Nicht empfehlenswert bei: Fehlersuche, klar spezifizierten kleinen Aufgaben, Trivialaufgaben, Routineaufgaben
6. Ist Pair Programming zur Einarbeitung neuer Mitarbeiter geeignet?
   (a) Ja (6)
7. Verläuft der Entwicklungsprozess bei Pair Programming zielgerichteter?

(a) Ja (4)

(b) Nein (2)

8. Geraten Sie beim Pair Programming seltener in gedankliche Sackgassen?

   (a) Ja (5)

   (b) Nein

XP empfiehlt bei der Entdeckung von Mängeln in der System-Architektur ein Redesign (Refactoring). Dies hat zur Folge, dass bei einer relativ kleinen Änderung u.U. umfangreiche Vorarbeiten notwendig werden. Dabei wird ggf. auch Code von anderen als den ursprünglichen Verfassern abgeändert (Collective Code Ownership).

1. Welche Erfahrungen haben Sie mit der konsequenten Durchführung von Refactorings gemacht?

   (a) Sehr gute (6)

2. Wie beurteilen Sie Collective Code Ownership?

   (a) Sehr empehlenswert (2)

   (b) Sehr empfehlenswert, erhöht die Disziplin (3)

   (c) Grundsätzlich empfehlenswert, sollte aber in sensiblen Bereichen eingeschränkt werden.

3. Beeinflusst Ihre Beurteilung des Gesamtsystems Ihre Einstellung, mit der sie die aktuelle Aufgabe bearbeiten?

   (a) Ja, sehr (3)

   (b) Ja (2)

   (c) Nein

XP betont die Eigenverantwortung der Entwickler bei der Abschätzung und beim System-Entwurf.

1. Haben Sie den Eindruck, dass dies den Bedürfnissen der Entwickler entspricht?

   (a) Ja, im überwiegenden Maße (5)

   (b) Nein

2. Gibt es Ihrer Meinung nach Bedarf für einen Projektleiter, der beispielsweise in kritischen Fragen Entscheidungen trifft?

   (a) Ja (6)

3. Wie funktioniert Ihrer Erfahrung nach die Übertragung eines Teils der Verantwortung auf einen Teamcoach, dessen Rolle täglich von einem anderen Entwickler wahrgenommen wird?

(a) Gut (2)

(b) Schlecht (4)

4. Welche Art der Aufgabenverteilung hat sich in Ihren Augen bewährt?

   (a) Dynamische Verteilung per Karteikartenpool bzw. Request-Tracker (3)

   (b) Dynamische Verteilung per Request-Tracker (2)

XP betont den Wert von direkter verbaler Kommunikation in Standup-Meetings.

1. Wie beurteilen Sie den Wert von Standup-Meetings?

   (a) Es werden relevante Informationen ausgetauscht (6)

2. Welche Regeln sollten Ihrer Meinung nach für Standup-Meetings gelten?

   (a) Ausufernde Diskussionen sollten schnell unterbrochen und im Anschluss an das Meeting fortgeführt werden. (3)

   (b) Alle Teilnehmer sollten tatsächlich aufstehen und sich auf den augenblicklichen Redner konzentrieren. (2)

   (c) Keine (2)

   (d) Ein Moderator sollte ausufernde Diskussionen unterbinden.

Die Aufgabenverteilung erfolgt bei XP durch Committen der einzelnen Entwickler zu bestimmten Tasks. Im Laufe des Projekts wurde dazu übergegangen, die Aufgaben nicht mehr auf Iterationsmeetings fest zu verteilen, sondern dynamisch über einen Pool mit Karteikarten.

1. Wie funktioniert diese Art der Aufgabenverteilung?

   (a) Sehr gut (4)

   (b) Nicht gut

2. Wie beurteilen Sie die Aufgabenverteilung über den Request-Tracker?

   (a) Sehr gut (5)

   (b) Gut

Zum Schluss noch ein paar Fragen zu XP-Komponenten, die bisher noch nicht zur Sprache gekommen sind.

1. Welche Erfahrungen haben Sie mit dem in XP-Projekten üblichen häufigen Integrieren gemacht?

   (a) Unbedingt empfehlenswert (4)

   (b) Nicht notwendig, eher störend (2)

2. Wie ist Ihr Gesamturteil zu XP?

   (a) Positiv (4)

(b) Neutral

3. Welche Dinge würden Sie an XP verbessern wollen?

   (a) Es sollte eine System-Entwurfsphase integriert werden. (2)

   (b) Die Verpflichtung zum Pair Programming sollte aufgehoben werden.

   (c) Die Einhaltung der Regeln sollte strenger kontrolliert werden.

   (d) Es sollte weniger mit verschiedenen Techniken (z.B. Aufgabenverteilung) experimentiert werden.

Kapitel 8

Anhang B - Abkürzungsverzeichnis

CASE Computer Aided Software Engineering

CMM Capability Maturity Model

CRC Class Responsibility Card

FDD Feature Driven Development

RUP Rational Unified Process

UML Unified Modeling Language

XP Extreme Programming

Kapitel 9

Anhang C - Glossar

Abwärts-Tailoring: das Streichen von *Aktivitäten* bei einem vollständigen *Software-Entwicklungsprozess*.

Aktivität: legt im Rahmen eines *Software-Entwicklungsprozesses* fest, *was* in welcher *zeitlichen Reihenfolge* zu tun ist.

Artefakt: ein im Laufe eines Software-Projekts erstelltes Produkt oder Zwischenprodukt. Artefakte umfassen Texte, Grafiken und Programme.

Aufwärts-Tailoring: das Hinzufügen von *Aktivitäten* zu einem unvollständigen *Software-Entwicklungsprozess*.

Computer Aided Software Engineering: der rechnergestützte Entwurf und die rechnergestützte Verwaltung der System-Architektur.

Design: der Akt des Entwurfs der *System-Architektur*.

Kritikalität: ein Maß für den maximalen Schaden, der durch einen Systemfehler verursacht werden kann.

Kriterium: ein Maß zur Definition der Beschaffenheit eines *Artefakts*.

Methode: legt im Rahmen eines *Software-Entwicklungsprozesses* fest, *wie* ein Arbeitsschritt durchzuführen ist. Auch: Bestandteil einer Klasse bei objektorientierter Programmierung.

Mock-Up: eine Attrappe, die das Verhalten eines bestimmten Teilsystems simuliert.

Pair Programming: eine Methode zur Implementierung von Programmen: die Entwickler arbeiten paarweise an einem Arbeitsrechner.

Produktivität: die mit einer Software erzielten Erlöse im Verhältnis zu den Produktionskosten.

Programm: Formulierung eines Algorithmus und der zugehörigen Datenbereiche in einer Programmiersprache.

Refactoring: die fortlaufende Anpassung der System-Architektur an die aktuellen Anforderungen.

Rolle: definiert im Rahmen eines *Software-Entwicklungsprozesses* die Verantwortlichkeiten einzelner Projektteilnehmer.

Scopus: die jeweilige Bandbreite der in einem Software-Entwicklungsprozess definierten *Rollen*, *Aktivitäten* und *Methoden*.

Software: die Gesamtheit aller *Programme*, die auf einer Rechenanlage eingesetzt werden können[1]

Software-Entwicklungsprozess: musterhafte Vorgehensweise zur Entwicklung von Software-Produkten. Ein Software-Entwicklungsprozess definiert *Rollen*, *Aktivitäten, Methoden* und *Kriterien*.

System: eine Einheit, die durch die Zusammenfassung mehrerer Komponenten entsteht.[2]

System-Architektur: legt die Beziehungen zwischen den Komponenten eines Systems fest und ordnet ihnen Verantwortungsbereiche zu.

System-Metapher: liefert eine plastische Vorstellung vom Zweck und der Struktur eines Systems.

Tailoring: die Anpassung eines *Software-Entwicklungsprozesses* für den Einsatz in einem bestimmten Projekt.

Task: eine klar umrissene Teilaufgabe, die innerhalb eines kurzen Zeitraums gelöst werden kann.

Validation: die Prüfung der Eignung eines Systems für einen bestimmten Einsatzzweck.

Verifikation: die Prüfung eines Systems auf Einhaltung einer bestimmten Spezifikation.

[1] [?, S. 655]
[2] vgl. [?, S. 717]

Index

Literaturverzeichnis

[V-Modell 1997] Allgemeiner Umdruck Nr. 250: *Entwicklungsstandard für IT-Systeme des Bundes, Vorgehensmodell*, BWB IT 15 1997

[Balzert 2000] Balzert, H: *Lehrbuch der Software-Technik. Bd. 1. Software-Entwicklung, 2. Auflage,* Spektrum-Verlag 2000; ISBN 3-8274-0480-0

[Balzert 1998] Balzert, H: *Lehrbuch der Software-Technik. Bd. 2. Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*, Spektrum-Verlag 1998; ISBN 3-8274-0065-1

[Beck 2000] Beck, K: *Extreme Programming. Die revolutionäre Methode für Softwareentwicklung in kleinen Teams*, Addison-Wesley 2000; ISBN 3-8273-1709-6

[Beck 2001] Beck, K; Fowler, M: *Planning Extreme Programming*, Addison-Wesley 2001; ISBN 0-201-71091-9

[Benington 1956] Benington, H: *Production of Large Computer Programs*, in: Proc. ONR Symposium on Advanced Programming Methods for Digital Computers 1956, pp. 15-27

[Boehm 1981] Boehm, B: *Software Engineering Economics*, Prentice Hall 1981; ISBN 0-1382-2122-7

[Boehm 1986] Boehm, B: *A Spiral Model of Software Development and Enhancement*, in: ACM SIGSOFT 1986, S. 14-24

[Brooks 1975] Brooks, F: *The Mythical Man-Month*, Addison-Wesley 1975; ISBN 0-201-00650-2

[Coad 1999] Coad, P et al.: *Feature Driven Development*, Object International, Inc. 1999; http://www.oi.com

[Cockburn 1998] Cockburn, A: *Surviving Object-Oriented Projects*, Addison-Wesley 1998; ISBN 0-201-49834-0

[Cockburn 2001] Cockburn, A: *Agile Software Development*, Addison-Wesley 2001; ISBN 0-201-69969-9

[DeMarco et al. 1999] DeMarco, T; Lister, T: *Wien wartet auf Dich! Der Faktor Mensch im DV-Management*, Hanser-Verlag 1999; ISBN 3-446-21277-9

[Engesser 1993] Engesser, H: *Duden Informatik*, Dudenverlag 1993; ISBN 3-411-05232-5

[Fowler 1999] Fowler, M: *Refactoring. Improving the design of existing code*, Addison-Wesley 2000; ISBN 0-201-48567-2

[Fowler 2000] Fowler, M; Scott, K: *UML konzentriert. Eine strukturierte Einführung in die Standard-Objektmodellierungssprache. 2. Auflage*, Addison-Wesley 2000; ISBN 3-8273-1617-0

[Gamma 1995] Gamma, E et al.: *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley 1995; ISBN 0-201-63361-2

[Highsmith 2000] Highsmith, J: *Extreme Programming*, e-Business Application Delivery newsletter, Cutter Consortium 2000

[Jacobson 1999] Jacobson, I et al.: *The Unified Software Development Process*, Addison-Wesley 1999; ISBN 0-201-57169-2

[Jeffries 2001] Jeffries, R et al.: *Extreme Programming Installed*, Addison-Wesley 2001; ISBN 201-70842-6

[Meyer 1988] Meyer, B: *Object Oriented Software Construction*, Prentice Hall 1988; ISBN 0-13-629049-3

[Paulk 1993] Paulk, M C et al.: *Capability Maturity Model for Software, Version 1.1*, Research Access, Inc. 1993; http://www.elliance.com

[Rational 1998] Rational Software Corporation: *Rational Unified Process. Best Practices for Software Development Teams*, Rational Software Corporation 1998; http://www.rational.com

[Rising 2000] Rising, L; Janoff, N S: *The Scrum Software Development Process for Small Teams*, IEEE Software 2000; http://www.ieee.org

[Royce 1970] Royce, W: *Managing the development of large software systems*, in: Proceedings of WESCON 1970, pp. 1-9

[Wells 1999] Wells, D: *Extreme Programming. A gentle introduction*, extremeprogramming.org, 1999; http://www.extremeprogramming.org

[Williams 2000] Williams, L; Cockburn, A: *The Costs and Benefits of Pair Programming*, pairprogramming.com, 2000; http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt.
Mit einer Veröffentlichung der Arbeit in der Bibliothek des Fachbereichs Informatik der Universität Hamburg erkläre ich mich einverstanden.

Hamburg, 15. November 2002

Raimund Heid