2

Agile und UML-basierte Methodik

Die wertvollsten Einsichten sind die Methoden.
Friedrich Wilhelm Nietzsche

Der zielgerichtete, in eine Methodik eingebettete Einsatz ist für die erfolgreiche Verwendung einer Modellierungssprache unverzichtbar. In diesem Kapitel werden Charakteristika agiler Methoden, insbesondere des *Extreme Programming (XP)* Prozesses [Bec04, Rum01] herausgearbeitet. Gemeinsam mit weiteren Elementen wird daraus ein Vorschlag für eine agile, auf der UML basierende Methodik eingeführt.

B.Rumpe, *Agile Modellierung mit UML*, 2. Aufl., Xpert.press
DOI 10.1007/978-3-642-22430-0_2, © Springer-Verlag Berlin Heidelberg 2012

Die Verbesserungen in der Softwaretechnik sind seit Jahren verantwortlich für kontinuierliche Effizienzsteigerungen in Softwareentwicklungsprojekten. Software mit hoher Qualität ist mit immer weniger personellen Ressourcen in immer kürzeren Zeiten zu erstellen. Prozesse wie der Rational Unified Process (RUP) [Kru03] oder das V-Modell XT [HH08] sind eher für große Projekte mit vielen Teammitgliedern geeignet. Sie sind aber speziell für kleinere Projektgrößen überfrachtet mit Tätigkeiten, die für das Endergebnis nicht unbedingt essentiell sind, und deshalb auch zu schwerfällig, um auf die sich schnell ändernde Umgebung (Technik, Anforderungen, Konkurrenz) eines Softwareentwicklungsprojekts reagieren zu können.

Flexibilität, schnelles Feedback, Konzentration auf die wesentlichen Arbeitsergebnisse und die Fokussierung auf die im Projekt beteiligten Personen und insbesondere Kunden sind wesentliche Charakteristika der neu entstandenden Prozesse, denen unter dem Begriff der „Agilen Prozesse“ ein gemeinsames Label gegeben wurde.

Der Standish Report [Gro09b] beschreibt, dass wesentliche Ursachen des Scheiterns von Projekten im schlechten Projektmanagement zu suchen sind: Es wird nicht adäquat kommuniziert, zu viel, zu wenig oder das Falsche dokumentiert, Risiken nicht rechtzeitig entgegengesteuert oder zu spät Rückkopplung von den Anwendern eingefordert.

Speziell die menschliche Komponente, also die projektinterne Kommunikation und Zusammenarbeit zwischen Entwicklern untereinander und der Entwickler mit den Anwendern, wird als wesentliche Ursache für das Scheitern von Softwareentwicklungsprojekten erkannt. Auch laut [Coc06] scheitern Projekte selten aus technischen Gründen. Das spricht einerseits für die gute Beherrschung auch von innovativen Techniken, ist andererseits aber zumindest teilweise zu hinterfragen. Denn wenn die Technik Schwierigkeiten bereitet, dann verursachen diese Probleme oft zwischenmenschliche Kommunikationsstörungen, die dann im weiteren Projektverlauf aus emotionalen Gründen in den Vordergrund rücken und letztlich als „gefühlte“ Gründe für das Scheitern des Projekts in Erinnerung bleiben.

Die richtige Technik ist in Form von Sprachen und Werkzeugen die wesentliche Grundlage für die „Leichtgewichtigkeit“ der neuen Prozessgeneration. Je kompakter die Sprache und je besser die Analyse- und Generierungswerkzeuge sind, um so effizienter sind die Entwickler und um so weniger Zusatzaufwand für Management oder Dokumentationsleistungen fällt an.

Softwaretechnik bietet mittlerweile ein großes *Portfolio* an Vorgehensweisen, Prinzipien, Entwicklungspraktiken, Werkzeugen und Notationen, die zur Entwicklung von Softwaresystemen unterschiedlichster Form, Größe und Qualität verwendet werden können. Diese Elemente des Portfolios ergänzen sich teilweise, können aber auch alternativ eingesetzt werden, so dass in einem Projekt eine Bandbreite an Auswahlmöglichkeiten zur Verfügung steht, die das Management, die Kontrolle und die Durchführung des Projekts betrifft.

Dieses Kapitel untersucht zunächst den aktuellen Stand des Portfolios der Softwaretechnik. In Abschnitt 2.2 wird „Extreme Programming“ (XP) diskutiert und in Abschnitt 2.3 drei essentielle Praktiken daraus vorgestellt. Abschnitt 2.4 beinhaltet einen Vorschlag für eine einfache Methode, die sich als Referenz für die detailliert diskutierten Notationen der UML/P und Techniken dieses Buchs eignet.

In Kapitel 3 folgt eine kompakte Übersicht über das für die vorgeschlagene Methode verwendbare Sprachprofil UML/P der Unified Modeling Language, die die wesentlichsten Eigenschaften des Sprachprofils erläutert und in [Sch12] mit einem geeigneten Werkzeugt unterstützt wird. Das Sprachprofil UML/P ist wie die UML selbst weitgehend *methodenunabhängig*. Das bedeutet, es ist möglich und sinnvoll, UML/P als Notation in Kombination mit anderen Methoden zu verwenden. Allerdings ist durch den Fokus auf Generierbarkeit von Code und Tests die UML/P besonders gut geeignet für den Einsatz in agilen Methoden.

2.1 Das Portfolio der Softwaretechnik

In [AMB+04, BDA+99] wurde ein Anlauf unternommen, das mittlerweile seit über 40 Jahren gewachsene Wissen der Softwaretechnik in einem „Software Engineering Body of Knowledge“ (SWEBOK) zusammenzufassen. Dabei werden Begriffsbildungen vereinheitlicht, die wesentlichen Kernelemente der Softwaretechnik als Ingenieursdisziplin dargestellt und insbesondere versucht, einen allgemein akzeptierten Konsens über die Inhalte und Konzepte der Softwaretechnik herzustellen.

Ein für unsere Überlegungen wesentlicher Teil der für Softwareentwicklungsprozesse verwendeten Terminologie ist in Abbildung 2.1 dargestellt.

Aus der in den letzten Jahren gesammelten Erfahrung bei der Durchführung von Softwareentwicklungsprojekten kristallisiert sich heraus, dass es *den* Prozess für Softwareentwicklung nicht geben kann. Es scheint heute noch nicht einmal möglich, ein einigermaßen aussagekräftiges Prozessframework angeben zu können, das für die nach Wichtigkeit, Größe, Anwendungsbereich und Projektumfeld sehr unterschiedlichen Softwareentwicklungsprojekte Allgemeingültigkeit besitzt. Stattdessen ist man dabei, eine Sammlung an Konzepten, Best Practices und Werkzeugen aufzustellen, die es erlaubt, projektspezifischen Erfordernissen in einem individuellen Prozess Rechnung zu tragen. Dabei werden Detaillierungsgrade und Präzision der Dokumente, Meilensteine und die abzuliefernden Ergebnisse in Abhängigkeit von der Größe des Projekts und der gewünschten Qualität des Ergebnisses festgelegt. Vorhandene, als Muster zu verstehende Prozessbeschreibungen können dabei Hilfestellung geben. Jedoch werden projektspezifische Anpassungen als grundsätzlich notwendig erachtet. Für Projektbeteiligte ist

Vorgehensmethode. Eine Vorgehensmethode (syn. *Vorgehensmodell*) beschreibt das Vorgehen „für die Abwicklung der Softwareerstellung in einem Projekt“ [Pae00]. Es kann zwischen technischem, sozialem und organisatorischem Vorgehensmodell unterschieden werden.

Softwareentwicklungsprozess. Der Begriff Softwareentwicklungsprozess wird gelegentlich als Synonym zu „Vorgehensmethode“ verwendet, oft aber als detaillierte Form verstanden. So definiert [Som10] einen Prozess als Menge von Aktivitäten und Ergebnissen, mit denen ein Softwareprodukt erstellt wird. Meist werden auch die zeitliche Reihenfolge oder die Abhängigkeiten der Aktivitäten herausgestellt.

Entwicklungsaufgabe. Ein Prozess wird in eine Reihe von Entwicklungsaufgaben unterteilt, die jeweils bestimmte Ergebnisse in Form von *Artefakten* erbringen. Die Projektbeteiligten führen *Aktivitäten* aus, um diese Aufgaben zu erledigen.

Prinzip. „Prinzipien sind Grundsätze, die man seinem Handeln zugrunde legt. Prinzipien sind allgemein gültig, abstrakt, allgemeinster Art. Sie bilden eine theoretische Grundlage. Prinzipien werden aus der Erfahrung und der Erkenntnis hergeleitet [...].“ [Bal00]

Best Practices beschreiben erfolgreich erprobte *Entwicklungspraktiken* in Entwicklungsprozessen. Eine Entwicklungspraktik kann als konkretes, operationalisiertes Prozessmuster aufgefasst werden, das ein allgemeines Prinzip umsetzt (vgl. RUP [Kru03] oder XP [Bec04]).

Artefakt. Entwicklungsergebnisse werden in einer konkreten Notation dargestellt. Dafür werden zum Beispiel die natürliche Sprache, UML oder eine Programmiersprache verwendet. Die Dokumente dieser Sprachen werden *Artefakte* genannt. Dies sind beispielsweise Anforderungsanalysen, Modelle, Code, Reviewergebnisse oder ein Glossar. Ein Artefakt kann hierarchisch gegliedert sein.

Transformation. Als Transformation kann die Entwicklung eines neuen Artefakts oder einer verbesserten Version eines gegebenen Artefakts verstanden werden, die automatisiert oder manuell erfolgen kann. Letztendlich können fast alle Aktivitäten als Transformationen der im Projekt gegebenen Menge von Artefakten verstanden werden.

Abbildung 2.1. Begriffsdefinitionen im Softwareentwicklungsprozess

es daher sinnvoll, möglichst viele Vorgehensweisen aus dem derzeit vorhandenen Portfolio zu kennen.

War in den 90’er Jahren ein starker Trend hin zu vollständigen und dadurch eher bürokratischen Softwareentwicklungsprozessen zu beobachten, so haben sich die agilen Methoden in den 2000’er Jahren von diesem Trend abgekoppelt. Ermöglicht haben diese Umkehr das deutlich gewachsene Verständnis der Aufgaben bei der Entwicklung komplexer Softwaresysteme sowie die Verfügbarkeit verbesserter Programmiersprachen, Compiler und einer Reihe von weiteren Entwicklungswerkzeugen. So ist es heute nahezu genauso effizient eine GUI direkt zu implementieren, wie sie zu spezifizieren. Entsprechend kann die Spezifikation durch einen für den Anwender ausprobierbaren und in der Realisierung weiterverwendbaren Prototyp ersetzt werden. Der Trend zur Reduzierbarkeit der notwendigen

Entwicklerkapazitäten wird dadurch verstärkt, dass bei weniger Projektbeteiligten auch weniger organisatorischer Overhead notwendig ist und damit die Personalaufwände weiter reduziert werden können.

Auch die mit agilen Methoden verstärkte Betonung der individuellen Fähigkeiten und Bedürfnisse der am Projekt beteiligten Entwickler und Kunden erlaubt die weitere Reduktion von Projektbürokratie zugunsten verstärkter Eigenverantwortung. Diese Teamorientierung kann auch in anderen Bereichen des Wirtschaftslebens, wie zum Beispiel bei flachen Management-Hierarchien, beobachtet werden und basiert auf der Annahme, dass mündige und motivierte Projektbeteiligte von sich aus verantwortungsvoll und couragiert handeln werden, wenn durch das Projektumfeld die Möglichkeiten dafür geschaffen werden.

2.2 Extreme Programming (XP)

Extreme Programming (in Kurzform: XP) ist eine „agile“ Softwareentwicklungsmethode, deren wesentlicher Kern in [Bec04] beschrieben ist. Obwohl XP als Vorgehensweise unter anderem in Softwareentwicklungsprojekten einer Schweizer Bank definiert und verfeinert wurde, zeigt schon die Namensgebung eine starke Beeinflussung durch die nordamerikanische, von Pragmatik geprägte Softwareentwicklungs-Kultur. Trotz des gewählten Namens ist XP allerdings keine Hacker-Technik, sondern besitzt einige sehr detailliert ausgearbeitete und rigoros zu verwendende methodische Aspekte. Diese erlauben es ihren Befürwortern zu postulieren, dass durch XP mit verhältnismäßig wenig Aufwand qualitativ hochwertige Software unter Einhaltung der Budgets zur Zufriedenstellung der Kunden erstellt werden kann. Statistisch aussagekräftige, über Anekdoten hinausgehende Untersuchungen von XP-Projekten gibt es mittlerweile einige [DD08, RS02, RS01].

XP erfreut sich in der Praxis einer hohen Popularität. Mittlerweile sind viele von Bücher über XP erschienen, von denen auch die ersten [Bec04, JAH00, BF00, LRW02] unterschiedliche Aspekte der Thematik sowie jeweils aktuelle Wissensstände der sich noch in Weiterentwicklung befindlichen Methodik detailliert betrachten oder Fallbeispiele durchgeführter Projekte illustrieren [NM01]. [Wak02, AM01] beinhalten vor allem praktische Hilfestellungen zur Umsetzung von XP, [Woy08] betrachtet kulturelle Aspekte und [BF00] diskutiert die Planung in XP-Projekten.

Eine kritische Beschreibung mit expliziter Diskussion von Schwachstellen von XP bieten [EH00a] und [EH01]. Darin wird unter anderem der fehlende Einsatz von Modellierungstechniken wie etwa der UML bemängelt und ein kritischer Vergleich zu Catalysis [DW98] gezogen. Eine dialektische Diskussion der Vor- und Nachteile von Extreme Programming beinhaltet [KW02]. Darin werden unter anderem die Notwendigkeit zum disziplinierten Vorgehen, die starken und gegenüber klassischen Ansätzen deutlich

geänderten Anforderungen insbesondere an den Teamleiter und den Coach hervorgehoben.

Nachfolgend werden wesentliche Elemente von XP gemäß den Einführungen aus [Bec04, Rum01] dargestellt und diskutiert. Weiterführende Themen in der Literatur behandeln mittlerweile XP parallel mit anderen agilen Methoden [Han10, Leh07, Ste10, HRS09] und erhalten so ein Portfolio agiler Techniken, oder sie adaptieren agile Methoden für verteilte Teams [Eck09] oder behandeln die Migration von Unternehmen hin zu agilen Methoden [Eck11].

Übersicht über XP

XP ist eine leichtgewichtige Softwareentwicklungsmethode. Darin wird auf eine Reihe von Elementen der klassischen Softwareentwicklung verzichtet, um so eine schnellere und effizientere Codierung zu erlauben. Die dabei für die Qualitätssicherung möglicherweise entstehenden Defizite werden durch stärkere Gewichtung anderer Konzepte (insbesondere der Testverfahren) kompensiert. XP besteht aus einer Reihe von Konzepten, von denen im Rahmen dieses Überblicks nur die wesentlichsten behandelt werden.

XP versucht explizit nicht auf neue oder nur wenig erprobte methodische Konzepte zu setzen, sondern integriert bewährte Techniken zu einem Vorgehensmodell, das auf Wesentliches fokussiert und auf organisatorischen Ballast soweit wie möglich verzichtet. Weil der Programmcode das ultimative Ziel einer Softwareentwicklung ist, fokussiert XP von Anfang an genau auf diesen Code. Alles an zusätzlicher Dokumentation wird als zu vermeidender Ballast betrachtet. Eine Dokumentation ist aufwändig zu erstellen und oft sehr viel fehlerhafter als der Code, weil sie normalerweise nicht ausreichend automatisiert analysierbar und testbar ist. Zusätzlich reduziert sie die Flexibilität bei der Weiterentwicklung und Anpassung des Systems als schnelle Reaktion auf neue oder veränderte Anforderungen der Kunden, wie es in der Praxis häufig der Fall ist. Folgerichtig wird in XP-Projekten (außer dem Code und den Tests) nahezu keine Dokumentation erstellt. Zum Ausgleich wird dafür Wert auf eine gute Kommentierung des Quellcodes durch Codierungsstandards und eine umfangreiche Testsammlung gelegt.

Die primären Ziele von XP sind die effiziente Entwicklung qualitativ hochwertiger Software unter Einhaltung von Zeit- und Kostenbudgets. Welche Mittel dazu eingesetzt werden, wird anhand der *Werte*, der *Prinzipien*, der grundlegenden *Aktivitäten* und der darin umgesetzten *Entwicklungspraktiken* in Pyramide in Abbildung 2.2 veranschaulicht.

Erfolgsfaktoren von XP

Nach mittlerweile einigen Jahren des Einsatzes von XP kristallisieren sich einige der für den Erfolg von XP-Projekten wesentlichen Faktoren heraus:

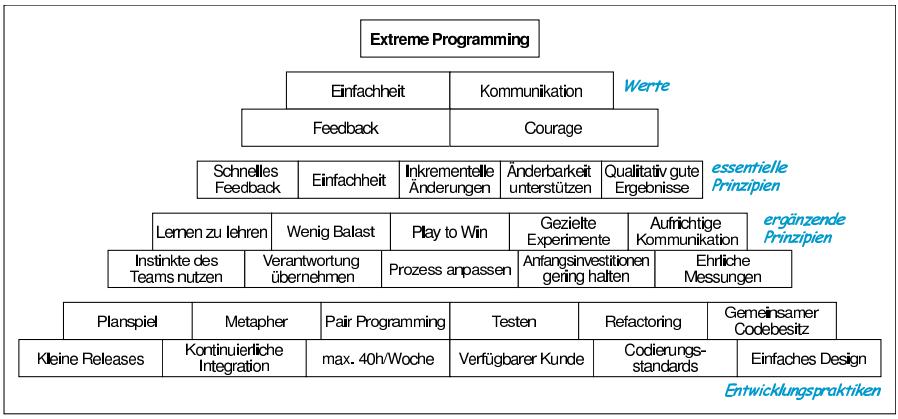

Abbildung 2.2. Aufbau des Extreme Programming

- Das Team ist motiviert und das Arbeitsumfeld für XP geeignet. Das bedeutet zum Beispiel, dass Arbeitsplätze für Pair Programming eingerichtet sind und die Entwickler in räumlicher Nähe untereinander und zum Kunden arbeiten.
- Der Kunde arbeitet in seiner Rolle aktiv am Projekt mit und steht für Fragen zur Verfügung. Die Studie [RS02] hat gezeigt, dass dies als einer der kritischen Erfolgsfaktoren zu werten ist.
- Die Wichtigkeit der Tests auf allen Ebenen erschließt sich, sobald Änderungen stattfinden, neue Entwickler hinzukommen oder das System eine gewisse Größe erhält und damit manuelle Tests nicht mehr durchführbar sind.
- Der in allen Bereichen diskutierte Zwang nach Einfachheit führt zum Weglassen von Dokumentation genauso wie zu einem möglichst einfachen Design und erlaubt daher eine signifikante Reduktion der Arbeitslast.
- Das Fehlen eines Pflichtenhefts und die Anwesenheit eines Kunden, mit dem auch während des Projekts über Funktionalität verhandelt werden kann, führt zu einer intensiveren Einbindung des Kunden in den Projektverlauf. Dies hat zwei Auswirkungen: Zum einen erlaubt es eine schnelle Reaktion auf sich verändernde Kundenwünsche. Zum anderen können so auch die Kundenwünsche durch das Projekt beeinflusst werden. Der Projekterfolg wird dadurch auch eine soziale Übereinkunft zwischen Kunden und Entwicklerteam und nicht nur eine objektiv überprüfte, auf Dokumenten basierende Zielerreichung.

Gerade der letzte Aspekt entspricht der XP-Philosophie, weniger zu kontrollieren und stattdessen mehr Eigenverantwortung und Engagement zu fordern. So könnte ein für alle Projektbeteiligten zufriedenstellenderes Ergebnis

entstehen, als es mit festgeschriebenen Pflichtenheften in einer Welt der sich wandelnden Anforderungen möglich ist.

Grenzen der Anwendbarkeit von XP

XP ist in Bezug auf die Projektdokumentation und auf die Einbindung von Kunden eine durchaus revolutionäre Vorgehensweise. Entsprechend gibt es eine Reihe von Einschränkungen und Anforderungen an Projektgröße und Projektumfeld, die für XP gelten. Diese werden unter anderem in [Bec04, TFR02, Boe02] erörtert. XP ist vor allem für Projekte bis zu zehn Personen gut geeignet [Bec04], aber es ist eine offensichtliche Problematik, XP für große Projekte zu skalieren, wie dies zum Beispiel in [JR01] diskutiert wird. XP ist eben nur eine weitere Vorgehensweise im Portfolio der Softwaretechnik, die wie viele andere Techniken auch nur unter den gegebenen Prämissen eingesetzt werden kann.

Die Grundannahmen, Techniken und Konzepte von XP führen zu einer relativ starken Polarisierung der Meinungen. Auf der einen Seite glauben Programmierer manchmal, dass XP das Hacking zur Vorgehensweise erhebt. Auf der anderen Seite wird XP nicht ganz ernst genommen, weil es vieles irgnoriert, was in den letzten Jahrzehnten an Entwicklungsprozessen erarbeitet worden ist. Tatsächlich ist beides nur sehr bedingt richtig. Zum einen können Hacker tatsächlich leichter zu einer XP-artigen Vorgehensweise als zu einem Vorgehen nach dem RUP motiviert werden. Zum anderen erkennen viele Softwareentwickler bei genauerer Betrachtung bereits bisher gelebte Entwicklungspraktiken wieder. Außerdem ist XP in seiner Vorgehensweise sehr rigoros und erfordert Disziplin in der Umsetzung.

Sicherlich richtig ist, dass XP eine leichtgewichtige Softwareentwicklungsmethode ist, die explizit als Gegengewicht zu schwergewichtigen Methoden wie dem RUP [Kru03] oder dem V-Modell XT [HH08] positioniert ist. Wesentliche Unterschiede stellen die Konzentration alleine auf den Code als Ergebnis und die Einbeziehung der Bedürfnisse der Projektbeteiligten dar. Der interessanteste Unterschied ist aber die gesteigerte Fähigkeit von XP auf Änderungen des Projektumfelds oder der Anforderungen der Anwender flexibel zu reagieren. Aus diesem Grund gehört XP zur Gruppe der „agilen Methoden“.

Die Fehlerbehebungskosten in XP-Projekten

Eine der grundlegenden Annahmen in XP stellt wesentliche Erkenntnisse der Softwaretechnik infrage. Während man bisher davon ausgegangen ist, dass die Kosten zur Behebung von Fehlern oder zur Durchführung von Änderungen wie in [Boe81] beschrieben exponentiell mit der Zeit steigen, geht XP davon aus, dass diese im Projektverlauf abflachen. Abbildung 2.3 zeigt diese beiden Kostenkurven schematisch.

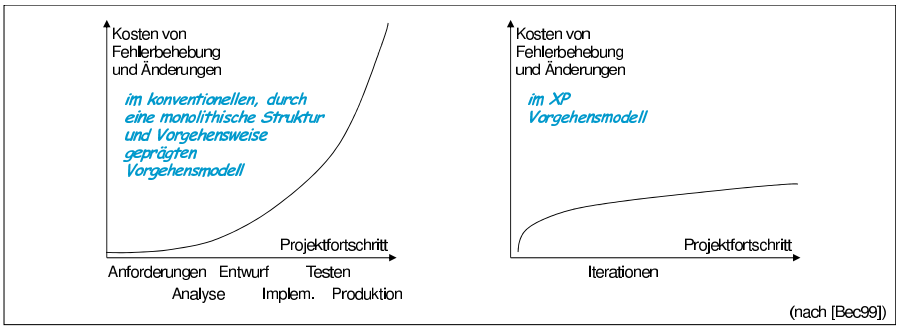

Abbildung 2.3. Fehlerbehebungskosten im Projektverlauf

In XP wird angenommen, dass die Änderungskosten sich mit der Zeit nicht mehr dramatisch erhöhen [Bec04, Kap. 5]. Diese Annahme von XP ist nicht empirisch belegt, birgt aber wesentliche Implikationen für die Anwendbarkeit von XP. Wird davon ausgegangen, dass mit XP die Kostenkurve abgeflacht werden kann, dann ist die initiale Entwicklung einer weitgehend korrekten und für alle noch kommenden Anforderungen ausbaufähigen Architektur tatsächlich nicht mehr essentiell. Die gesamte Wirtschaftlichkeit des XP-Ansatzes basiert daher auf dieser Annahme.

Es gibt allerdings Anhaltspunkte, die zumindest für eine gewisse Abflachung der Kostenkurve in XP sprechen. Durch die Nutzung besserer Sprachen wie Java, besserer Web- und Datenbanktechnologie und die Verbesserung der Entwicklungspraktiken, die sich auch in Codierungsstandards widerspiegeln, sowie bessere Werkzeuge und Entwicklungsumgebungen können Mängel schneller und umfassender behoben werden. Auch Mängel, die nicht an einer Stelle lokalisiert sind, können aufgrund des gemeinsamen Codebesitzes ohne größere Planung und Diskussionsrunden behoben werden. Der Verzicht auf Dokumentation verhindert die Notwendigkeit erstellte Dokumente konsistent zu halten. Demgegenüber steht der Aufwand, die automatisierten Tests nachzuziehen. Allerdings bietet die Automatisierung den wesentlichen Vorteil einer effizienten Erkennung von nicht mehr korrekten Tests gegenüber dem aufwändigen Korrekturlesen von schriftlicher Dokumentation.

Einer der wesentlichsten Indizien für eine reduzierte Kostenkurve ist aber die Vorgehensweise in kleinen Iterationen. Fehler und Mängel, die sich in einer Iteration lokalisieren lassen, haben nur innerhalb dieser Iteration eine mit der Zeit exponentiell ansteigende Wirkung. In darauf folgenden Iterationen bleibt die Wirkung allerdings begrenzt, so dass dann nur noch ein langsamer Anstieg der Fehlerbehebungskosten zu erwarten ist. Die iterative Vorgehensweise, eventuell gekoppelt mit der Dekomposition des Systems in Subsysteme führt daher möglicherweise zu der in Abbildung 2.4

dargestellten Kostenkurve, die zum Beispiel auch in dem Auktionsprojekt zu finden war.[1]

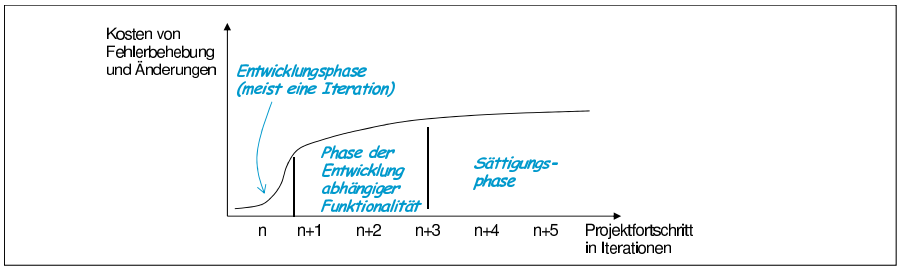

Abbildung 2.4. Fehlerbehebungskosten in iterativen Projekten

Abhängig von der Lokalisierbarkeit des Fehlers innerhalb eines Teils des Systems kann in der verursachenden und den unmittelbar darauf aufbauenden Iterationen ein gewisser Anstieg der Fehlerbehebungskosten entstehen. In späteren Iterationen sollte sich dann nur noch der Aufwand zur Identifikation des Mangels und seiner Quelle erhöhen. Für Mängel in der Architektur, die damit viele Teile des Systems betreffen, tritt die Sättigung allerdings erst sehr spät und auf hohem Niveau ein, so dass es sich im Allgemeinen lohnen dürfte einen gewissen initialen Aufwand in die Architekturmodellierung zu investieren.

Obwohl also gewisse Argumente zumindest partiell für die erreichbare Kostenreduktion sprechen, muss ein Beleg dieser Aussage durch Zahlenmaterial erst durch die Untersuchung einer ausreichenden Menge von XP-Projekten erfolgen.

Es kann jedoch als Vorteil von XP angesehen werden, dass durch die Entwicklung automatisierter Tests, die kontinuierliche Integration, das Pair Programming, die kurzen Iterationszyklen und das permanente Feedback mit den Kunden die Wahrscheinlichkeit, Fehler frühzeitig zu finden, erhöht wurde.

Beziehung zwischen XP und CMM

In dem Artikel [Pau01] wurde durch einen der Autoren des Capability Maturity Models (CMM) festgestellt, dass XP und das Software-CMM [PWC+95] keine unvereinbaren Widersprüche darstellen, sondern sich ergänzen. XP besitzt demzufolge gute Entwicklungspraktiken, die Kernanforderungen des CMM erfüllen. XP bietet für viele vom CMM geforderte „key process areas“ (KPA) der fünf CMM-Levels teilweise eigenwillige, jedoch für den von der XP-Vorgehensweise abgedeckten Projektbereich gut geeignete Praktiken,

[1] Statistisch aussagekräftiges Zahlenmaterial existiert dazu allerdings nicht.

die die Ziele des CMM adressieren. Der Artikel [Pau01] schlüsselt die Unterstützung der KPA's durch XP in einer Tabelle einzeln auf, in der von 18 KPA's sieben negativ und elf positiv bis sehr positiv bewertet wurden. Von den negativ bewerteten können allerdings das Trainingsprogramm aufgrund des Pair Programming von Experten mit Neulingen ebenfalls als positiver bewertet und das Management von Unterverträgen vernachlässigt werden. [Gla01] beschreibt übereinstimmend dazu, dass der XP-Ansatz ohne Mühe CMM-Level 2 einhält und der projektbezogene Anteil von CMM-Level 3 ebenfalls ohne viel Zusatzaufwand erreichbar ist. CMM-Level 3 erfordert allerdings projektübergreifende, unternehmensweite Maßnahmen, die durch XP nicht adressiert werden.

Zusammenfassend kommt [Pau01] zu dem nachvollziehbaren Schluss, dass XP gute Techniken beinhaltet, die Unternehmen durchaus in Betracht ziehen könnten, aber sehr kritische Systeme nicht ausschließlich mit XP-Techniken realisiert werden sollten.

Erkenntnisse aus den Erfahrungen mit XP

XP stellt einen leichtgewichtigen Prozess aus kohärenten Techniken zur Verfügung, der durch die Fokussierung auf den Code wesentlich effizienter gestaltet werden kann als zum Beispiel der RUP. Durch die Effizienz werden gleichzeitig weniger Ressourcen benötigt und der Prozess flexibler. Mit der gestiegenen Flexibilität wird eines der Grundprobleme der Softwareentwicklung, der Behandlung von sich über die Projektlaufzeit verändernden Anwenderanforderungen, gemildert.

Um die Qualität des entwickelten Produkts zu sichern, werden Pair Programming und rigorose automatisierte Tests gefordert, ohne aber auf eines der seit langem bekannten Testverfahren zurückzugreifen. Eine weitere Steigerung der Qualität und der Effizienz wird durch die permanente Forderung nach Einfachheit erreicht.

Aufgrund der vorliegenden Analyse von XP, die teilweise aus der Literatur und teilweise aus eigenen Projekterfahrungen unter anderem bei dem in Anhang D, Band 1 beschriebenen Auktionsprojekt beruht, lässt sich XP als eine für kleine Projekte im Bereich von ungefähr 3-15 Personen geeignete Vorgehensweise einstufen. Durch geeignete Maßnahmen, wie zum Beispiel der hierarchischen Dekomposition von größeren Projekten entlang von Komponenten [JR01, Hes01] und die damit einhergehenden zusätzlichen Aktivitäten, lässt sich XP begrenzt auf größere Aufgaben skalieren.

Umgekehrt kann jedoch auch versucht werden, die Größe der Aufgabe durch Verwendung innovativer Technik zu verringern. Dazu gehört die Wiederverwendung und Adaption eines vorhandenen Systems, soweit dies möglich ist.

Insbesondere gehört dazu aber die Weiterentwicklung von Sprachen, Werkzeugen und Klassenbibliotheken. Viele mit technischem Code behaftete

Programmteile, zum Beispiel zur Ausgabe, Speicherung oder Kommunikation von Daten, sind sich strukturell ähnlich. Wenn diese Programmteile generiert und mit einer abstrakten Darstellung der Applikation zu lauffähigen Code komponiert werden können, dann ergeben sich weitere Steigerungen der Effizienz von Entwicklern. Dies gilt für den Produktionscode, aber vielmehr noch für die Entwicklung von Tests, die bei abstrakter Modellierung durch Diagramme außerdem besser verständlich sind.

Eine Verwendung einer ausführbaren Teilsprache der UML als High-Level-Programmiersprache muss dementsprechend zum Ziel haben, Modelle noch effizienter zu entwickeln und in Code umzusetzen und damit eine weitere Beschleunigung der Softwareentwicklung zu bewirken. Idealerweise reduziert sich der Entwurfs- und Implementierungsanteil eines Projekts soweit, dass ein Projekt vor allem aus der Erhebung von Anforderungen besteht, die effizient und direkt umgesetzt werden können.

2.3 Ausgesuchte Entwicklungspraktiken

Pair Programming, der dem XP zugeordnete Test-First-Ansatz und die Weiterentwicklung von Code werden als drei der technik-orientierten Elemente von XP weiter untersucht, da sie auch bei agiler Modellierung mit UML eine bedeutende Rolle spielen.

2.3.1 Pair Programming

Pair Programming war bereits vor XP bekannt und wurde zum Beispiel in [Con95] beschrieben. War es zunächst eine eigenständige Technik, so ist es heute unter anderem in XP integriert, weil es eine gute Basis für gemeinsamen Codebesitz darstellt und es deshalb grundsätzlich für alle Teile des System mindestens zwei Personen gibt, die damit vertraut sind.

Die wesentliche Idee des Pair Programming ist auch als „Vier-Augen-Prinzip“ bekannt: Zwei Entwickler sitzen gemeinsam an einer Aufgabe, die sie kooperativ lösen. Sie benötigen dafür nur einen Rechner und während einer das gemeinsam Erarbeitete eintippt, findet durch den Partner gleichzeitig ein konstruktiver Review statt. Die Tastatur und damit die Kontrolle über das konstruktiv Eingearbeitete wird allerdings sehr schnell zwischen den Beteiligten gewechselt. Dieses Prinzip war zunächst zur Anwendung unter Gleichen gedacht, eignet sich aber auch zur Kombination eines Kenners des Systems mit einem Projektneuling. Auf diese Weise kann die Einarbeitung des Neulings in vorhandene Softwarestrukturen sowie neue Techniken effizient erfolgen. Rein rechnerisch bedeutet Pair Programming aber zunächst doppelten Personalaufwand.

Pair Programming wurde in einigen an Universitäten angesiedelten Tests bereits eingehender untersucht beziehungsweise Untersuchungsschemata erstellt [SSSH01]. Die Untersuchungen [WKCJ00, CW01] zeigen, dass nach

einer relativ kurzen Einarbeitungszeit in den neuen Programmierstil zwar der Gesamtaufwand gegenüber der Einzelprogrammierung erhöht war, sich aber eine deutliche Reduktion der Projektdauer und insbesondere eine signifikante Erhöhung der Softwarequalität ergeben hat.[2] Allerdings zeigt sich auch, dass die Technik des paarweisen Programmierens erst erlernt werden muss und dass paarweises Programmieren nicht jedem liegt.

In der Praxis dürfte deshalb ein rigoros erzwungenes Pair Programming nicht zielführend sein, sondern vielmehr eine kooperative Förderung desselben zu optimalen Ergebnissen führen. Dies ist vor dem Hintergrund zu sehen, dass es im Projekt durchaus neben dem Programmieren Tätigkeiten gibt, die eine paarweise Ausübung nicht erfordern. Dazu gehören zum Beispiel Planungstätigkeiten, Werkzeuginstallation und -wartung sowie unter Umständen Diskussionen mit Kunden zur Erhebung von Anforderungen. Leider stehen auch flexible Arbeitszeiten der Entwickler und ungünstige Räumlichkeiten einer konsequenten Anwendung des Pair Programming im Weg.

2.3.2 Test-First-Ansatz

In verschiedenen Methodiken werden Tests an unterschiedlicher Stelle und unterschiedlich intensiv diskutiert und eingesetzt. Während zum Beispiel das V-Modell XT eine explizite Trennung zwischen den Tests von Methoden, Klassen, Subsystemen und des Gesamtsystems trifft, ist im Rational Unified Process nach [Kru03] weder eine Unterscheidung der Testebenen, noch eine Diskussion von Metriken zur Testüberdeckung zu finden. In XP spielt Testen als eine der vier Kernaktivitäten eine wesentliche Rolle und wird deshalb genauer diskutiert.

In [Bec01] ist auf anschauliche Weise beschrieben, welche Vorteile der in XP propagierte Test-First-Ansatz bei der Softwareentwicklung haben kann. In [LF02] wird dieser Ansatz zur Beschreibung von Unit-Tests elaboriert und Vor- und Nachteile sowie der methodische Einsatz in pragmatischer Form diskutiert. [PP02] vergleicht den Test-First-Ansatz mit traditioneller Entstehung des Tests nach der Implementierung in einem allgemeinen Kontext. [Wak02, S. 8] beinhaltet eine Beschreibung für einen Mikro-Entwicklungszyklus auf Basis von Tests und Codierung.

Die wesentliche Idee des Test-First-Ansatzes besteht darin, sich vor der Entwicklung der eigentlichen Funktionalität (insbesondere einzelner Methoden) Testfälle zu überlegen, die zur Überprüfung der Korrektheit der zu realisierenden Funktionalität geeignet sind, um damit diese Funktionalität

[2] Als statistisch aussagekräftiges Zahlenbeispiel wird in [WKCJ00, CW01] angegeben, dass beim Pair Programming die Entwicklungskosten um 15% gestiegen sind, aber der resultierende Code um 15% weniger Mängel hatte. Dies erlaubt eine Reduktion der Fehlerbehebung. Es wird geschätzt, dass dadurch die Gesamtkosten um 15% bis 60% reduziert werden können.

exemplarisch zu beschreiben. Diese Testfälle werden in automatisiert ablaufenden Tests festgehalten. Laut [Bec01, LF02] hat dies eine Reihe von Vorteilen:

- Die Definition der Testfälle vor der eigentlichen Implementierung erlaubt es, eine explizite Abgrenzung der zu realisierenden Funktionalität vorzunehmen, so dass der Testentwurf de facto einer Spezifikation gleichkommt.
- Der Test begründet die Notwendigkeit des Codes und beschreibt unter anderem, welche Parameter für die zu realisierende Funktion notwendig sind. Der Code wird dadurch so entworfen, dass er *testbar* ist.
- Nach der Realisierung der Funktionalität können die bereits vorhandenen Testfälle zur sofortigen Überprüfung verwendet werden, wodurch das Zutrauen in den entwickelten Code enorm steigt. Obwohl natürlich eine Fehlerfreiheit nicht gesichert ist, zeigt doch die praktische Anwendung, auch im Auktionsprojekt, dass dieses Zutrauen gerechtfertigt ist.
- Der logische Entwurf wird von der Implementierung besser getrennt. Während der Definition der Testfälle, also noch vor der Implementierung, wird zunächst die Signatur der neuen Funktionalität bestimmt. Dies beinhaltet Namen, Parameter und die Klassen, in der Methoden angesiedelt sind. Erst im nächsten Schritt, nach der Definition der Tests, wird die Implementierung der Methoden vorgenommen.
- Der Aufwand zur Formulierung von Testfällen ist im Allgemeinen nicht zu vernachlässigen, insbesondere wenn komplexe Testdaten zu erstellen sind. Das führt laut [Bec01] dazu, dass Funktionen so definiert werden, dass ihnen nur die jeweils benötigten Daten zur Verfügung gestellt werden. Dies resultiert in einer Entkopplung der Klassen und damit in besseren und einfacheren Entwürfen.
  Dieses Argument mag in Einzelfällen zutreffen, ist möglicherweise aber nicht immer richtig. Vielmehr erfolgt Entkopplung von Klassen eher durch frühzeitiges Erkennen der Möglichkeit zur Entkopplung oder durch nachträgliches Refactoring. Dies demonstrieren auch typische Beispiele des Test-First-Ansatzes [LF02, Bec01].
- Die frühzeitige Definition von Testdatensätzen und Signaturen ist auch beim Pair Programming hilfreich. Entwickler können dann expliziter über die gewünschte Funktionalität diskutieren.

Ein Vorteil von vorhandenen Testfällen ist, dass andere Entwickler anhand der Testfallbeschreibungen die gewünschte Funktionalität erkennen können. Dies ist oft notwendig, wenn der Code unübersichtlich und nicht ausreichend kommentiert ist und eine explizite Spezifikation der Funktionalität nicht besteht. Die Testfälle stellen daher selbst ein Modell für das System dar.

Die Erfahrung zeigt aber, dass die Möglichkeit, sich anhand der Tests die Funktionalität zu erarbeiten, beschränkt ist. Dies liegt daran, dass Tests normalerweise weniger sorgfältig definiert werden als der eigentliche Code und

die in einer Programmiersprache geschriebenen Tests die eigentlichen Testdaten nicht sehr kompakt und übersichtlich darstellen.

Obwohl der Test-First-Ansatz eine Reihe von Vorteilen bietet, ist dennoch in der Praxis davon auszugehen, dass die Vorab-Definition von Tests keine ausreichende Überdeckung der Implementierung beinhaltet. Die aus der Testtheorie bekannten Überdeckungsmetriken werden aber in XP nicht eingesetzt. Man gibt sich weitgehend mit einem sehr informellen Begriff zur Testüberdeckung zufrieden, der vor allem auf der Intuition der Entwickler beruht. Diese Situation ist einerseits für das Controlling eines Projekts wenig zufriedenstellend, zeigt aber andererseits dennoch praktische Erfolge. Zur Zeit werden trotzdem Werkzeuge entwickelt beziehungsweise für den XP-Ansatz adaptiert [Moo01], die durch automatische Verfahren versuchen, herauszufinden, inwieweit die Tests lokale Änderungen des Codes entdecken und damit bestimmte Überdeckungskriterien erfüllen. Dazu gehören zum Beispiel Mutationstests [Voa95, KCM00, Moo01], die durch einfache Mutation des Testlings prüfen, ob diese Veränderungen durch einen Test entdeckt werden.

Ist die Implementierung der gewünschten Funktionalität gegeben, so sollte nach Erfüllung der initialen Testsammlung durch die Entwicklung weiterer Tests eine bessere Überdeckung erreicht werden. Dazu gehören zum Beispiel die Behandlung von Grenzwertfällen und die Untersuchung von Fallunterscheidungen und Schleifen, die teilweise durch die Technik oder ein genutztes Framework bedingt sein können und deshalb in den vorab entwickelten Testfällen nicht antizipiert wurden.

Generell wird der Test-First-Ansatz als eine Tätigkeit angesehen, die Analyse, Entwurf und Implementierung in „Mikro-Zyklen“ vereint. In [Bec01] wird jedoch auch erwähnt, dass Testverfahren, die zur systematischen Definition von Testfällen dienen und nach verschiedenen Kriterien die Überdeckung des Codes messen, im Wesentlichen ignoriert werden. Das wird absichtlich in Kauf genommen mit dem Argument, dass dadurch sehr viel mehr Aufwand entsteht, aber die Ergebnisse nicht wesentlich (wenn überhaupt) verbessert werden. Immerhin wird in [LF02, S. 59] darauf hingewiesen, dass Tests nicht nur vor der Implementierung, sondern auch nach Fertigstellung einer Aufgabe erstellt werden, um dann eine „ausreichende“ Überdeckung zu erzielen, ohne allerdings präzise zu klären, wann eine Überdeckung ausreichend ist.

In Abhängigkeit von der Art und Größe des durchgeführten Projekts kann der strikte Test-First-Ansatz ein interessantes Element des Softwareentwicklungsprozesses sein, der typischerweise nach einer zumindest initialen Architekturmodellierung und einer Partitionierung in Subsysteme eingesetzt werden kann. Unter Verwendung der ausführbaren Teilsprache der UML/P kann dieser Ansatz mehr oder weniger unverändert auf die Modellierungsebene gehoben werden. Dazu können zunächst exemplarische Daten mittels Objektdiagrammen und exemplarische Abläufe mit Sequenzdiagrammen als Testfälle modelliert werden, auf deren Basis die zu realisierende

Funktionalität mit Statecharts und Klassendiagrammen modelliert werden kann.

2.3.3 Refactoring

Der Wunsch nach Techniken zur inkrementellen Verbesserung und Modifikation von Programmcode durch Regelsysteme ist nicht wesentlich jünger als die Entstehung der ersten Programmiersprachen [BBB+85, Dij76]. Ziel einer transformationellen Softwareentwicklung ist die Zerlegung des Softwareentwicklungsprozesses in kleine, systematisch durchführbare Schritte, die aufgrund lokal begrenzter Auswirkungen beherrschbar werden. Refactoring wurde bereits in [Opd92] für Klassendiagramme erstmals diskutiert. Das empfehlenswerte Buch [Fow99] beschreibt eine ausführliche Sammlung von Transformationstechniken für die Programmiersprache Java. Die Refactoring-Techniken erlauben die Migration von Code entlang einer Klassenhierarchie, die Zusammenlegung oder Teilung von Klassen, die Verschiebung von Attributen, das Expandieren oder Auslagern von Code-Teilen in eigene Methoden und ähnliches mehr. Die Stärke der Refactoring-Techniken basiert auf der Überschaubarkeit der einzelnen als „Mechanik" bezeichneten Transformationsschritte und auf der flexiblen Kombinierbarkeit, die zu großen, zielorientierten Verbesserungen der Softwarestruktur führt.

Das Ziel des Refactorings sind semantikerhaltende Transformationen eines bereits existenten Programms. Refactoring dient nicht zur Erweiterung der Funktionalität, sondern zur Verbesserung der Qualität des Entwurfs unter Beibehaltung der Funktionalität. Es ergänzt damit die normale Programmiertätigkeit.

Refactoring und die Weiterentwicklung von Funktionalität sind sich ergänzende Tätigkeiten, die sich im Entwicklungsprozess in schneller Folge abwechseln können. Der Verlauf eines Projekts kann damit wie in Abbildung 2.5 skizziert werden. Allerdings existiert für die „Qualität des Designs" kein objektives Messkriterium. Erste Ansätze dafür messen zum Beispiel die Konformität zu Codierungsstandards, sind aber für eine Evaluation der Qualität von Architektur und Implementierung in Bezug auf Wartbarkeit und Testbarkeit unzureichend.

Zur Sicherung der Korrektheit semantikerhaltender Transformationen werden keine Verifikationstechniken eingesetzt, sondern die vorhandene Testsammlung genutzt. Unter der Annahme der Existenz einer qualitativ hochwertigen Testsammlung ist die Wahrscheinlichkeit hoch, dass fehlerhafte, also das Verhalten des Systems verändernde Modifikationen erkannt werden. Durch Refactoring werden oft interne Signaturen eines Teilsystems modifiziert, wenn zum Beispiel eine Methode einen zusätzlichen Parameter erhält. Deshalb müssen bestimmte Tests gemeinsam mit dem Code angepasst werden. In [Pip02] wird sogar vorgeschlagen, im Sinne des Test-First-Ansatzes zunächst die Tests und dann erst den Code einer Refaktorisierung zu unterziehen.

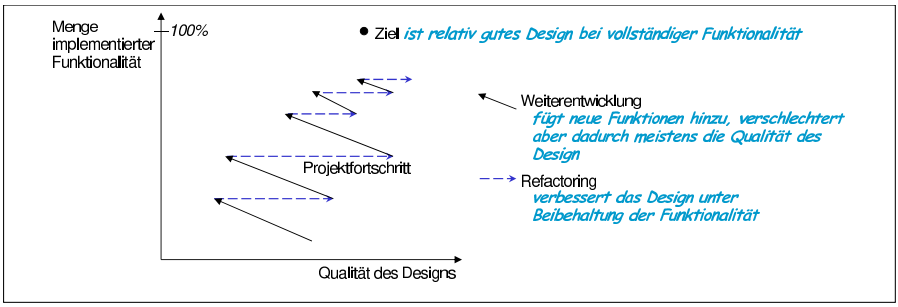

Abbildung 2.5. Refactoring und Weiterentwicklung ergänzen sich

Refactoring-Techniken zielen auf verschiedene Ebenen des Systems. Manche Refactoring-Regeln wirken im Kleinen, andere sind für Modifikation einer Systemarchitektur geeignet. Durch die Möglichkeit, eine bereits im Code realisierte Systemarchitektur zu verbessern, verliert die Notwendigkeit a priori eine korrekte und stabile Systemarchitektur zu definieren an Brisanz. Natürlich sind Modifikationen an der Systemarchitektur kostenintensiv. In XP wird jedoch angenommen, dass die Wartung ungenutzter Systemfunktionalität auf Dauer kostenintensiver ist. Entsprechend dem Prinzip der *Einfachheit* wird es im XP-Ansatz bevorzugt, die Systemarchitektur einfach zu halten und nur bei Bedarf durch geeignete Refactoring-Schritte zu modifizieren oder zu erweitern.

Während die Definition von Refactoring-Techniken für Java auf Basis von [Fow99] bereits stark im Ausbau ist, existieren nur wenig ähnliche Techniken für UML-Diagramme [SPTJ01].

2.4 Agile UML-basierte Vorgehensweise

Aufgrund der Diversität der Softwareentwicklungsprojekte in Größe, Fachdomäne, Kritikalität, Kontext etc. lässt sich folgern, dass eine einheitliche Vorgehensweise in der diversifizierten Projektlandschaft nicht sinnvoll und auch nicht möglich sein wird.

Stattdessen ist, wie in Crystal [Coc06] vorgeschlagen, aus einer Sammlung von Vorgehensweisen nach Kriterien wie Größe und Art des Projekts, Kritikalität der Anwendung sowie Erfahrung und Kenntnissen der Projektbeteiligten eine geeignete Vorgehensweise zu wählen und bei Bedarf anzupassen.

Entsprechend dieser Diversität wird hier ein Vorschlag einer leichtgewichtigen, agilen Methode unter Nutzung der UML skizziert. Dieser Vorschlag konzentriert sich vor allem auf den technischen Teil einer Vorgehensweise und stellt keinen Anspruch für alle Arten von Projekten geeignet zu sein.

Definition des Begriffs „Agilität“

Die Charakterisierung der Agilität einer Methode ist in Abbildung 2.6 skizziert.

Eine Vorgehensmethode wird als „agil“ bezeichnet, wenn sie verstärkt Wert auf folgende Kriterien legt:

- Die*Effizienz* des Gesamtprozesses und der einzelnen Schritte ist möglichst optimal.
- Die *Reaktivität*, also die Geschwindigkeit, mit der auf sich ändernde Anforderungen oder ein anderes Projektumfeld reagiert wird, ist hoch. Die Planungen sind daher eher *kurzfristig und adaptierbar*.
- Auch die Vorgehensmethode selbst ist *flexibel adaptierbar* um sich inneren und durch das Umfeld bestimmten äußeren und daher nur teilweise kontrollierbaren Projektgegebenheiten dynamisch anzupassen.
- *Einfachheit* und pragmatische Umsetzung der Entwicklungsmethode und ihrer Techniken führen zu einfachem Entwurf und Implementierung.
- Die *Kundenorientierung* der Methode erlaubt und erfordert die aktive Einbindung der Kunden während des Projekts.
- Den *Fähigkeiten, Kenntnissen und Bedürfnissen der Projektbeteiligten* wird im Projekt Rechnung getragen.

Abbildung 2.6. Begriffsdefinition: Agilität einer Methode

Effizienz kann durch geschicktes Weglassen unnötiger Arbeit (Dokumentation, nicht benötigte Funktionalität, etc.) verbessert werden, aber auch durch die effizientere Erstellung der Implementierung.

Die *Qualität* des entwickelten Produkts ist nicht notwendigerweise im Fokus aller Techniken einer agilen Methode. Bestimmte Elemente einer agilen Methode, wie zum Beispiel die Konzentration auf die Einfachheit des Entwurfs und die umfassende Entwicklung automatisierter Tests, unterstützen die Verbesserung der Qualität. Andere Vorgaben, die von manchen agilen Methoden getroffen werden, wie etwa das Fehlen detaillierter Spezifikationen und Reviews, stehen einer Verwendung für hochkritische Systeme eher entgegen, so dass dafür ein alternativer Prozess oder eine geeignete Erweiterung zu wählen ist.

Unterstützung der Agilität durch verbesserte Techniken

Die Verbesserung der Agilität eines Projekts kann neben einer Neugestaltung der Aktivitäten insbesondere durch Erhöhung der *Effizienz* der einzelnen Entwickler erreicht werden. Dabei sind Effizienzsteigerungen bei der Erstellung von Modellen, der Implementierung und von Tests von Interesse. Besonders wirksam ist es, wenn die Implementierung und die Tests möglichst

effizient oder sogar vollständig automatisch aus Modellen hergeleitet werden können. Eine solche automatische Generierung ist dann von Interesse, wenn die benutzte Quellsprache für die Modelle eine kompaktere Darstellung erlaubt, als es die Implementierung selbst könnte.

Prämisse für diese Einsatzform von Modellen ist die schnellere Erstellung aufgrund größerer Kompaktheit, die aber dennoch eine vollständige Beschreibung des Systems erlaubt. Die UML ist in der im UML-Standard [OMG10] angebotenen Form dafür nicht ausreichend. Deshalb ist die UML/P so erweitert worden, dass sie eine vollständige Programmiersprache beinhaltet.

Einsatzformen für Modelle

Modelle, aus denen Code generiert wird, erfordern einen hohen Detaillierungsgrad. Dafür entfällt die aufwendige Aktivität der sonst üblichen Codierung. Die detaillierte Modellierung und die Implementierung verschmelzen zu einer einzigen Tätigkeit. Codegenerierung ist also ein wichtiges Hilfsmittel für den erfolgreichen Einsatz von Modellen. Dadurch wird ein ähnliches Ziel wie in XP erreicht, bei dem Entwurfs- und Modellierungsvorgänge grundsätzlich im Code repräsentiert werden.

Es gibt jedoch neben der Codegenerierung noch weitere Ziele für die Modelle eingesetzt werden können. Für die Kommunikation zwischen Entwicklern reichen *abstrakte* und relativ *informelle* Modelle aus. *Abstraktion* bedeutet, dass Details ausgelassen werden können, die zur Darstellung des kommunizierten Sachverhalts nicht von Interesse sind. *Informalität* bedeutet, dass die sprachliche Korrektheit des dargestellten Modells nicht präzise eingehalten werden muss. Beispielsweise können Diagramme auf Papier durchaus intuitive, aber nicht notwendigerweise zur Sprache gehörige notationelle Elemente nutzen, wenn die Entwickler sich über deren Bedeutung soweit wie notwendig einig sind. Außerdem muss bei *informellen* Diagrammen die Konsistenz des Diagramms mit dem modellierten Sachverhalt sowie anderen Diagrammen nicht vollständig gegeben sein.

Modelle können darüber hinaus zur Dokumentation des Systems eingesetzt werden. Dokumentation ist eine in Schriftform gebrachte und damit für längeren Gebrauch vorgesehene Form der Kommunikation. Damit sind je nach Ziel der Dokumentation ein höherer Detaillierungsgrad, größere Formalität und Konsistenz sinnvoll. Für ein knappes Dokument zur Übersicht der Architektur eines Systems und eine Einführung in wesentliche Entscheidungen, die zu dieser Architektur führten, ist die Detaillierung niedrig, aber Formalität und Konsistenz innerhalb des Modells und mit der Implementierung wichtig. Für eine vollständige Dokumentation ist es ideal, die Übereinstimmung der Dokumentation mit dem implementierten System dadurch zu erreichen, dass die Implementierung und soweit möglich auch Tests aus den Modellen der Dokumentation generiert werden.

In einem Projekt, das sich auf die UML/P zur Modellierung und zur Implementierung stützt, ist also zwischen

- *Implementierungs-Modellen*,
- *Testmodellen*,
- *Modellen zur Dokumentation* und
- *Modellen zur Kommunikation*

zu unterscheiden. Für alle Zwecke lässt sich die UML/P einsetzen, obwohl diese Modelle unterschiedliche Charakteristika besitzen. Ein wesentlicher Vorteil besteht aber darin, dass zwischen Spezifikation, Implementierung und Testfällen keine Notationsbrüche stattfinden. Die UML/P bietet die Möglichkeit, sowohl für detaillierte, als auch für abstrakte und unvollständige Modellbildungen verwendet zu werden.

Bearbeitung von Modellen

Wenn ein Modell ausschließlich zur Kommunikation dient, liegt es typischerweise nur als informelle Zeichnung vor. Ein Modell, das durch ein Werkzeug erstellt wurde, besitzt eine formale Repräsentation der Syntax[3] und kann daher zur werkzeuggestützten Weiterbearbeitung dienen. Die Syntax der UML/P ist deshalb in den Anhängen A, Band 1, B, Band 1 und C, Band 1 definiert.

Als eine der wesentlichen Techniken für den Einsatz von Modellen wurde bereits die *Codegenerierung* identifiziert. Sie besitzt zwei Varianten. Zum einen kann der Produktionscode generiert werden. Zum anderen kann Testcode generiert werden.

Die Notwendigkeit zur Adaption von Software aufgrund sich ändernder Bedingungen erfordert es außerdem, dass auch Techniken zur *Transformation* beziehungsweise für das *Refactoring* dieser Modelle zur Verfügung stehen. Die systematische, bei jedem Schritt durch automatisierte Tests und Invarianten überprüfte Adaption von Modellen auf neue und geänderte Funktionalitäten erlaubt diese bereits aus XP bekannte dynamische Anpassung. Da ein Modell nach den aufgestellten Anforderungen kompakter und damit leichter verständlich ist als der Code, wird die Anpassung von Modellen ebenfalls erleichtert. Zum Beispiel können strukturelle Anpassungen, die im Code über viele Dateien verstreut waren, innerhalb eines Klassendiagramms bearbeitet werden. Die Notwendigkeit der frühen Entwicklung und Fixierung einer adäquaten Architektur nimmt, wie bereits im XP-Ansatz beobachtet, weiter ab. Umgekehrt steigt die Flexibilität zur Einarbeitung neuer Funktionalität und damit die Agilität des Projekts weiter an.

[3] Zum Beispiel ist das *XML Metadata Interchange Format* (XMI) ein Standard zur Darstellung von UML-Modellen und damit eine solche formale Repräsentation der Syntax.

Als wesentliche Techniken im Umgang mit Modellen können daher

- die Generierung des Produktionscodes,
- die Generierung des Testcodes und
- das Refactoring von Modellen

identifiziert werden [Rum02]. Darüber hinaus sind verschiedene Techniken zur Analyse von Modellen, wie zum Beispiel die Erreichbarkeit von Zuständen in Statecharts, von Interesse. Hilfreich zur effizienten Erstellung von Tests ist auch die Ableitung von Testfällen aus OCL-Bedingungen oder Statecharts.

Spezielle Techniken, wie zum Beispiel zur Generierung von Datenbanktabellen, einer einfachen graphischen Oberfläche aus Datenmodellen oder zur Migration von zum alten Modell konformen Datenbeständen in ein neues Modell, sind ebenfalls wichtig und von einem geeignet parametrisierten Codegenerator zu unterstützen. Solche Techniken werden allerdings in diesem Buch nicht weiter vertieft.

Weitere sinnvolle Prinzipien und Praktiken

Für die Komplettierung einer Vorgehensweise sind projektspezifisch weitere Prinzipien und Praktiken zu wählen. Für kleine Projekte lässt sich zum Beispiel die folgende, vor allem an XP angelehnte Liste identifizieren:

- Viele, kleine Iterationen und Releases,
- Einfachheit,
- schnelles Feedback,
- permanenter Dialog mit dem ständig verfügbaren Kunden,
- tägliche kurze interne Treffen zur Abstimmung,
- Review nach jeder Iteration zur Prozessoptimierung,
- Tests vor der Implementierung entwickeln („Test-First“),
- Pair Programming als Lern- und Review-Technik,
- gemeinsamer Besitz der Modelle,
- kontinuierliche Integration und
- Modellierungsstandards als Vorgabe für Form und Kommentierung der Modelle ähnlich den Codierungsstandards.

Die Entwicklung von Tests vor einer Implementierung nach dem in Abschnitt 2.3.2 diskutierten Test-First-Ansatz lässt sich hervorragend auf UML/P übertragen. Sequenzdiagramme, die eine wesentliche Komponente von Tests sind, werden zunächst als Beschreibungen für Anwendungsfälle modelliert und können dann mit vertretbarem Aufwand in Testfälle transformiert werden. Ähnliches gilt für Objektdiagramme, die als Beispiele für Datensätze bei der Anforderungserfassung erstellt werden können.

Ein projektweiter Modellierungsstandard ist notwendig, um die Lesbarkeit eines erstellten Modells, das von allen Projektbeteiligten bei Bedarf angepasst und erweitert werden kann, sicherzustellen.

Analog zur Verwendung einer Programmiersprache ist es notwendig, dass die mit UML/P modellierenden Entwickler die benutzte Sprache beherrschen. Eine eventuell notwendige Lernphase kann durch Pair Programming, das jetzt besser als „paarweise Modellierung" zu bezeichnen ist, unterstützt werden.

Qualität, Ressourcen und Projektziele

Die Effekte der in diesem Abschnitt skizzierten Vorgehensweise können anhand des in Abschnitt 2.2 diskutierten Wertesystems und der unter anderem ebenfalls in XP verwendeten vier Kriterien (Zeitverbrauch, Kosten, Qualität und Zielfokussierung) erörtert werden.

Kommunikation ist anhand von UML/P-Modellen besser zu bewerkstelligen als anhand von Implementierungscode, wenn davon ausgegangen wird, dass die Kommunikationspartner UML beherrschen.

Einfachheit der Software wird aus zwei Gründen noch besser unterstützt. Zum einen ist es aufgrund der leichten Änderbarkeit noch unwichtiger, Strukturen für potentiell notwendige, zukünftige Funktionalität vorzubereiten und dadurch frühzeitig eventuell ungenutzte Komplexität einzubauen. Zum anderen können nicht mehr notwendige Elemente durch Refactoring-Schritte aus dem Modell noch leichter entfernt werden. Es bleibt jedoch notwendig, dass die Entwickler selbst unnötige Komplexität und überflüssige Funktionalität erkennen.[4]

Feedback ist durch die größere Effizienz der Entwicklung und die daraus resultierenden noch kürzeren Iterationen gesichert.

Eigenverantwortung und Courage müssen und können den Entwicklern wie in XP erhalten bleiben.

Die vier wesentlichen Variablen des Projektmanagements werden ebenfalls adressiert:

Zeitverbrauch. Zeitliche Einsparungen und insbesondere die frühzeitige Verfügbarkeit erster Fassungen („Time-To-Market") ist nicht nur für Internet-Applikationen wesentlich. Die gesteigerte Effizienz der Entwicklung und die höhere Wiederverwendbarkeit technischer Anteile führen diesbezüglich zu einer Optimierung.

Kosten sinken aufgrund gesteigerter Effizienz und des daraus resultierenden reduzierten Zeit- und Personalaufwands. Aufgrund gesunkenen Personalaufwands können außerdem Managementelemente weggelassen werden, so dass der verwendete Prozess leichtgewichtiger und damit weitere Einsparungen möglich werden.

[4] Analysewerkzeuge können hier begrenzt Unterstützung liefern, indem sie erkennen, welche Methoden, Parameter oder Klassen nicht benötigt werden.

Qualität. Die Qualität des Produkts wird durch die kompakte und damit übersichtlichere Darstellung des Systems, durch die leichtere Modellierung von Testfällen und durch den nicht mehr vorhandenen Bruch zwischen Modellierungs- und Implementierungssprache positiv beeinflusst. Ob die Qualität des resultierenden Systems den Ansprüchen genügt, kann durch Konzentration auf weitere Aspekte, wie dem Test-Überdeckungsgrad, zusätzlichen Modell-Reviews und der aktiven Einbindung des Kunden gesteuert werden.

Zielfokussierung. Um sicherzustellen, dass das System in der vom Anwender gewünschten Form implementiert wird, ist die aktive Einbindung des Anwenders wichtig. Die Verwendung der UML/P hat darauf keinen Einfluss, denn es ist davon auszugehen, dass einem Anwender im Normalfall kaum UML-Diagramme zur Diskussion vorgelegt werden können.

Probleme der agilen, UML-basierten Softwareentwicklung

Neben den diskutierten Vorteilen hat die skizzierte Vorgehensweise auch einige Nachteile, die beachtet werden sollten:

- Der Vorteil, dass nahezu dieselbe Notation für die abstrakte Modellierung und die Implementierung verwendet werden kann, kann sich als Bumerang erweisen. Frühere Ansätze zur Modellierung essentieller Eigenschaften, wie etwa SDL [IT07b, IT07a] oder Algebraische Spezifikationen [EM85, BFG+93], wurden aufgrund ihrer Ausführbarkeit in der Praxis vor allem als High-Level-Programmiersprachen eingesetzt. Dasselbe wird nun für eine Teilsprache der UML vorgeschlagen.[5] Beim Einsatz der UML/P zur Spezifikation kann dies dazu führen, dass für die Spezifikation unnötige Details ausgefüllt werden, weil ein späterer Einsatz als Implementierung bereits vorweg genommen wird. Dieses Phänomen ist auch als „Überspezifikation“ oder „Overengineering“ bekannt.
- Der Lernaufwand für die Verwendung der UML/P ist als groß einzuschätzen. UML/P ist syntaktisch deutlich reichhaltiger als Java, so dass sich hier ein schrittweises Vorgehen empfiehlt. Zunächst ist der Einsatz zur Strukturbeschreibung mit Klassen- und Objektdiagrammen interessant. Als nächstes können Sequenzdiagramme zur Testfallmodellierung und darauf basierend OCL zur Definition von Bedingungen, Invarianten und Methodenspezifikationen erlernt werden. Die Verwendung von Statecharts zur Verhaltensmodellierung erfordert am meisten Übung.

[5] Im Fall der algebraischen Spezifikationen hat dies zumindest teilweise dazu geführt, dass der Fokus weniger auf der abstrakten Modellierung von Eigenschaften und mehr auf der Umsetzbarkeit in effizienten Code lag.

Ähnlich wie in Java ist jedoch nicht nur die Beherrschung der Syntax, sondern insbesondere die Verwendung von Modellierungsstandards und Entwurfsmustern sinnvoll, die ebenfalls erlernt werden müssen.

- Derzeit existiert noch wenig Werkzeugunterstützung, die alle Aspekte der gewünschten Codegenerierung vollständig abdeckt. Insbesondere ist die Effizienz der Codegenerierung wesentlich zur schnellen Erstellung des Systems, die wiederum effiziente Durchführung von Tests und das daraus resultierende Feedback ermöglicht.
  Jedoch arbeiten eine Reihe von Werkzeugherstellern an der Realisierung einer solchen Vision und können bereits gute Erfolge vorzeigen.

2.5 Zusammenfassung

Agile Methoden bilden einen neuen, sich durch mehrere Charakteristika explizit von bisher in der Softwareentwicklung eingesetzten Methoden abhebenden Ansatz. Die Werkzeuge, Techniken und das Verständnis für die Probleme der Softwareentwicklung haben sich signifikant verbessert. Deshalb können diese Methoden unter anderem durch eine Verstärkung des Fokus auf das Primärergebnis, das lauffähige System und einer Reduktion sekundärer Arbeitsleistung für einen Teilbereich der Softwareentwicklungsprojekte effizientere und flexiblere Vorgehensweisen anbieten. Gleichzeitig wird die Motivation und das Engagement der Projektbeteiligten gegenüber detailliert ausgearbeiteten, aber starren Aktivitätsbeschreibungen in den Vordergrund gerückt und durch kurze Iterationen die flexible, situationsabhängige Steuerung der Softwareentwicklung ermöglicht.

Als wesentliche Faktoren zur Bestimmung der Größe eines Projekts wurden die *Problemgröße* und die *Effizienz der Entwickler* genannt. Aus diesen beiden Größen leitet sich die notwendige *Methodengröße* ab, die im Wesentlichen aus den auszuführenden methodischen Elementen, die für die Dokumente notwendige Formalität und dem Zusatzaufwand für Kommunikation und Management besteht. Wegen zusätzlicher Kommunikations- und Managementsoverheads geht in die Bestimmung der *Projektgröße*, also der Anzahl der Entwickler und der Laufzeit, die *Effizienz der Entwickler* überproportional ein. Deshalb ist eine Verbesserung der Entwicklereffizienz ein wesentlicher Hebel für die Reduktion der Entwicklungskosten.

Die UML/P gibt als Modellierungssprache für die *Architekturmodellierung*, *Entwurf* und *Implementierung* einen einheitlichen Sprachrahmen vor, der die vollständige Modellierung des ausführbaren Systems sowie der Tests erlaubt. Die Kompaktheit der Darstellung führt zu größerer Effizienz und den daraus resultierenden Skalierungseffekten für die Projektgröße. Der einheitliche Rahmen verhindert einen sonst oft zu beobachtenden Notationsbruch zwischen Modellierungs-, Implementierungs- und Testsprache.