Universität Potsdam
Institut für Informatik
Didaktik der Informatik
Prof. Dr. A. Schwill

# Didaktische Grundfragen der Informatik
# Partizipative Methoden der Softwareentwicklung

---

Autor: Harald Meyer
E-Mail: hmeyer@cs.uni-potsdam.de
Semester: Wintersemester 02/03

Potsdam, den 3. März 2003

**Zusammenfassung**

Im Folgenden soll das Gebiet der Partizipation bei der Softwareentwicklung näher beleuchtet werden. Hierbei handelt es sich um Methoden, die eine bessere Beteiligung der Benutzer an möglichst vielen Phasen der Softwareentwicklung ermöglichen sollen. Hierzu werden zuerst die Probleme traditioneller Softwareentwicklungsprozesse dargestellt. Dann wird ein von Matthias Rauterberg beschriebenes Modell für einen alternativen Softwareentwicklungsprozess vorgestellt, den dieser als „Globalen Optimierungszyklus“([Rau95]) bezeichnet. Nach der Einführung dieses Prozesses werden die Methoden beschrieben, mit denen eine höhere Benutzerbeteiligung erreicht werden kann. In diesem Zusammenhang werden dann auch die Probleme, die sich aus einer erhöhten Benutzerbeteiligung ergeben, erläutert.

Neben dem globalen Optimierungszyklus, der, wie später noch ersichtlich wird, eher ein theoretisches Modell ist, werden zwei moderne Softwareentwicklungsprozesse, die auch praktische Anwendung finden, vorgestellt. Hierbei handelt es sich um Extreme Programming ([Bec00]) und um den Rational Unified Process ([Kru98]). Beide werden insbesondere im Hinblick auf die Realisierung der Benutzerbeteiligung untersucht.

# Inhaltsverzeichnis

# 1 Einleitung

Im Gegensatz zu anderen Ingenieursdisziplinen sind noch immer unzumutbar viele IT-Projekte nicht erfolgreich. Ungefähr 30% der Projekte scheitern vollständig und werden vor Projektende abgebrochen ([Sta95]). Bei Projekten, die zwar nicht vollständig scheitern, aber mehr kosten und länger dauern als geplant und nicht die gewünschten Einsparungen zur Folge haben, liegt die Quote, je nach Studie und Land, zwischen 60% und 90% (siehe z. B. [Sta95], [OAS96], [KPM97]). Da die Studien aus den letzten 10 Jahren stammen, ist davon auszugehen, dass moderne Techniken wie die objektorientierte Programmierung oder die Definition von Prozessmodellen hier kaum Vorteile gebracht haben.

Das hat in erster Linie zwei Ursachen. Erstens werden bekannte und erprobte Techniken im Alltag kaum eingesetzt. Ein Beispiel hierfür ist das Testen der Software. Entgegen der allgemeinen Erkenntnis, dass Software wie jedes andere Produkt auch getestet werden muss, geschieht dies, wenn überhaupt, nur unsystematisch und ungeplant. Dass es zumindest in diesem Bereich auch anders geht, zeigt die NASA. Die Software, die der Steuerung der Space Shuttles dient, ist praktisch fehlerfrei[1] ([Fis96]).

Die zweite Ursache für das Scheitern vieler IT-Projekte ist, dass die Softwareentwickler oftmals die Erwartungen der zukünftigen Benutzer nicht kennen. Diese Unkenntnis existiert auf vielen Ebenen: Angefangen bei den Anforderungen die gestellt werden, damit sich das Programmsystem in den vorhandenen Arbeitsablauf der gesamten Organisation integriert, bis hin zur Integration in individuelle Arbeitsabläufe einzelner Benutzer (Anforderungen an die Benutzerschnittstelle etc.). Traditionelle Softwareentwicklungsprozesse erkennen zwar die Wichtigkeit der Anforderungsanalyse, doch können sie diese nur unzureichend umsetzen, da sie oftmals von starren, am Anfang der Entwicklung festgelegten, Anforderungen ausgehen. Die Einbindung in den Arbeitsalltag der Benutzer können sie nicht modellieren, da diese nicht in die Entwicklung eingebunden sind. Durch die Nutzung partizipativer Methoden lässt sich das Problem der unzureichenden Anforderungsanalyse abmildern. Eine Möglichkeit eines iterativen, zyklischen Softwarentwicklungsprozesses, der die Benutzer beteiligt, wird im Folgenden an Hand des „Globalen Optimierungszyklus“ von Matthias Rauterberg vorgestellt.

# 2 Traditionelle Softwareentwicklungsprozesse

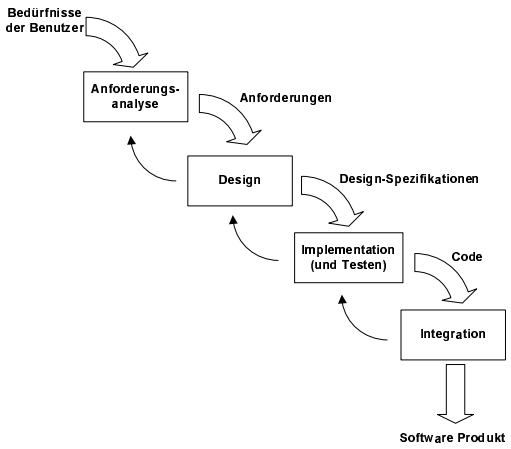


Abbildung 1: Wasserfallmodell

Der wohl bekannteste traditionelle Softwareentwicklungsprozess folgt dem Wasserfallmodell(Abbildung 1). Obwohl er schon relativ alt ist[2], wird er immer noch sehr häufig eingesetzt. Trotz berechtigter Kritikpunkte hat er auch einige Vorteile. So teilt er die Softwarentwicklung in mehrere Phasen ein (vier ([Rau95]) bis sieben([HS93])) und ermöglicht so eine Arbeitsteilung zum Beispiel zwischen Softwarearchitekten, die die Architektur entwerfen, und

[1] Die letzten drei Version der Software hatten angeblich jeweils nur einen einzigen Fehler
[2] Er wurde schon in den 70er Jahren durch Barry Boehm([Boe76]) beschrieben

Programmierern, die für die Implementation des Systems zuständig sind. Neuere Varianten des Wasserfallmodells ermöglichen außerdem die Rückkehr in die vorherige Phase, so dass Fehler, die in dieser aufgetreten sind, korrigiert werden können.

Geht man von einem vier-phasigen Modell aus so sind die Phasen: Anforderungsanalyse, Design, Implementation und Integration. Während der Anforderungsanalyse werden die Anforderungen des Auftraggebers an das zu entwickelnde Programmsystem gesammelt und in eine geeignete Form gebracht. In der folgenden Phase wird aus ihnen dann die Architektur des Systems entworfen. Die Implementations-Phase dient dann der eigentlichen programmiersprachlichen Umsetzung und dem Testen. Die Entwicklung schließt mit der Integrations-Phase ab, die dazu dient das System in den Arbeitsablauf der Organisation einzubinden. In ihrem Rahmen wird dann auch überprüft, ob das System den Anforderungen genügt.

Ein Nachteil ist der Aufwand des Wasserfallmodells, da jede Phase mit einem greifbaren Ergebnis abschließt. So existiert am Ende der Anforderungsanalyse als Ergebnis eine Dokumentation, in der Anforderungen der Anwender in Form von Benutzungsszenarien beschrieben sind. Dieses Dokument, muss nun durch den gesamten Prozess „mitgeschleppt“ werden. Ändern sich die Anforderungen, so muss das Dokument entweder im Nachhinein angepasst werden, was mit Zeitaufwand verbunden ist, oder es veraltet. Praktisch wird meistens der zweite Fall eintreten, weil niemand mehr Lust oder Zeit hat das Dokument zu ändern oder weil es schlicht vergessen wird. So oder so trägt man Ballast mit sich herum, der den gesamten Prozess lähmen kann.

Problematisch beim Wasserfallmodell ist weiterhin, dass trotz der Möglichkeit des Rückschritts in eine vorherige Phase das Modell selbst sequentiell ist. Es wird also davon ausgegangen, dass jede Phase durch einmaliges Abarbeiten (gegebenfalls mit Rückschritten zur Fehlerkorrektur) absolviert werden kann. Moderne Ansätze widersprechen dieser Annahme. Sie gehen davon aus, dass man die einzelnen Phasen beim ersten Durchlaufen weder korrekt noch vollständig abschließen kann. Rauterberg führt zur Untermauerung dieser These mehrere Barrieren an, die im Folgenden erläutert werden.

## 2.1 Barrieren der Softwarentwicklung

Insgesamt handelt es sich um drei Barrieren, die Rauterberg beim Einsatz traditioneller Softwareentwicklungsprozesse ausmacht. Sie können die Entwicklung eines Softwareprodukts verlangsamen oder vollständig zum Stillstand bringen. Selbstverständlich treten bei der Softwareentwicklung noch weitere Probleme auf, die man ebenfalls als Barrieren, die es zu überwinden gilt, charakterisieren kann. Ein Beispiel wäre die Qualifikationsbarriere, die auftreten kann, wenn bei der Entwicklung neue Technologien eingesetzt werden sollen, die von den Entwicklern nur unzureichend beherrscht werden. Solche Art von Barrieren wird aber im Folgenden weder ausführlich beschrieben, noch werden Strategien zu ihrer Überwindung erläutert.

**Spezifikationsbarriere** Wie ihr Name nahelegt, verhindert die Spezifikationsbarriere, dass der Auftraggeber seine Anforderungen an das zu entwickelnde Programmsystem korrekt und vollständig angeben kann.

Die Ursache hierfür sind zwei Probleme, die sich zu dieser Barriere ergänzen. Das erste Problem ist die Form der Spezifikationen. Softwareentwickler wünschen sich, dass sie möglichst formal und detailliert sind. Nur so lassen sich Widersprüche leicht erkennen und Fehlentwicklungen können vermieden werden. Dem Auftraggeber fehlt dagegen oftmals die Qualifikation formale Spezifikation korrekt angeben zu

können. Da er meist auch keine Zeit oder Motivation hat sich diese anzueignen, muss man auf weniger formale Beschreibungstechniken umsteigen, die dann natürlich wieder die Gefahr von Fehlentwicklungen und Missverständnissen erhöhen.

Das zweite Problem ist, dass es auf Seiten des Auftraggebers unterschiedliche Benutzergruppen gibt, die differierende Wünsche und Vorstellung an das System haben. Der Auftraggeber selbst ist in den seltensten Fällen jemand der das System auch benutzen wird. Er wird bestenfalls wissen, wie das System in den vorhandenen Arbeitsablauf der Organisation eingebunden werden soll und verwaltet die Mittel, die für die Entwicklung des Systems zur Verfügung gestellt werden sollen.

Rauterberg schlägt daher eine Einteilung der Benutzer in drei Gruppen vor: Anwender, Benutzer und End-Benutzer. Der Anwender entspricht dem Auftraggeber und wird sich daher meist im Management finden. Er hat weder direkt noch indirekt mit dem System zu tun. Auch die Benutzer arbeiten nicht direkt mit dem System, brauchen aber für ihre Arbeit die Ergebisse, die mit ihm erstellt wurden. Sie können Auskunft über die Einbindung in den vorhandenen Arbeitsablauf geben. Bei einer Software zur Erstellung von Rechnungen können sie zum Beispiel angeben über welche Export-Möglichkeiten die Reporting[3]-Funktionalität der Software verfügen soll.

Die End-Benutzer schließlich benutzen das Programmsystem direkt. Neben der einfachen Beschreibung ihrer Anforderungen an die Benutzeroberfläche können sie oftmals auch nützliche Informationen über Anforderungen aus Teilbereichen, die eigentlich den Anwendern und Benutzern vorbehalten sind, geben. Im Gegensatz zu den Anwendern kennen sie den realen Arbeitsablauf in der Organisation, der möglicherweise wenig mit den Modellen und Vorstellungen der Personen des höheren Managements (also den Anwendern) zu tun hat. Die Benutzer nehmen möglicherweise bestimmte Zustände als gegeben hin ohne darüber nachzudenken, welchen Arbeitsaufwand die End-Benutzer haben, um diesen Zustand zu erreichen.

Um auf das oben genannte Beispiel zurückzukommen: Ein Benutzer erwartet die Quartalsberichte möglicherweise immer als Excel-Datei. Bisher haben die End-Benutzer hierfür alle von Hand geschriebenen Rechnungen in eine solche Datei eingetragen. Wird nun ein neues Programmsystem eingeführt, das die Rechnungsstellung erleichtern soll, dem aber der Excel-Export fehlt, so ist den End-Benutzern wahrscheinlich wenig geholfen. Denn sie müssen die Daten immer noch doppelt eingeben. Ist das neue Programmsystem auch noch schwer zu bedienen, so werden sie möglicherweise die Rechnungen weiterhin von Hand schreiben und der Auftraggeber wird sich wundern, warum es trotz der teuren Neuentwicklung zu keinerlei Effizienzsteigerungen gekommen ist.

**Kommunikations-Barriere** Sie kann zusammen mit der Spezifikationsbarriere oder unabhängig von dieser in der Phase der Anforderungsanalyse[4] auftreten. Ihr Problem ist nicht eine fehlende Qualifikation oder eine falsche Sicht auf das Programmsystem, sondern die Kommunikation zwischen Softwareentwicklern und den verschiedenen Benutzergruppen. Beide Seiten haben unterschiedliche Fachsprachen. Termini, die Softwareentwicklern selbstverständlich erscheinen (z. B. „Use Case“[5]), müssen den Anwendern erst erklärt werden. Die Softwareentwickler sind ebenfalls in den seltensten Fällen Fachleute in der Branche, für die sie die Software entwickeln. Sie

[3] Erstellung von Berichten, z. B. Umsatzbericht für ein Quartal

[4] Findet die Benutzerbeteiligung auch in späteren Phasen der Softwareentwicklung statt, so kann die Kommunikations-Barriere auch dort auftreten.

[5] Deutsch: Anwendungsszenario, dient der Anforderungsanalyse

müssen allerdings die Begriffe nicht nur verstehen, sondern auch richtig anwenden können (z. B. bei der Entwicklung der Oberfläche). Würden sich die beiden Fachsprachen nur auf unterschiedliche Begriffe beschränken, so wäre die Kommunikations-Barriere leicht zu überwinden: Beide Seiten könnten für die jeweils andere Seite ein Art Wörterbuch erstellen, das während der Entwicklung weiter wächst. In ihm könnte man jederzeit unklare Begriffe nachschlagen.

Doch die unterschiedliche Fachsprache ist nur ein Teil des Problems. Sie ist Ausdruck unterschiedlicher Denkstrukturen, Traditionen und Herangehensweisen in den verschiedenen Fachgebieten. Die Erstellung der Wörterbücher ist somit weites gehend sinnlos. Die Begriffe können nur unzureichend losgelöst von einem zusammenhängenden System erläutert werden. Man kann ja auch keine Sprache lernen, indem man die Übersetzungen der Wörter lernt. Man muss die Grammatik verstehen und ein Gefühl für die Sprache entwickeln. Mit rein sprachlichen Mitteln lässt sich die Kommunikations-Barriere somit kaum überwinden. Man muss also visuelle Kommunikationsmittel einsetzen.

**Optimierungs-Barriere** Die letzte von Rauterberg beschriebene Barriere ist schließlich die Optimierungs-Barriere. Sie gilt es zu überwinden, um die beschränkten Mittel, die zur Verfügung stehen, optimal einzusetzen. Das gilt sowohl im technischen als auch im sozialen Kontext. Während die Optimierungs-Barriere im technischen Kontext einigermaßen gut beherrscht werden kann[6], ist das für den sozialen Kontext, also die Einbindung in die Arbeitsumgebung der Anwender, nicht der Fall.

Ein Beispiel hierfür ist die Anforderungsanalyse. Setzt man traditionelle Softwareentwicklungsprozesse wie das Wasserfallmodell ein, so geht man davon aus, dass man sie als eine Phase am Anfang der Softwareentwicklung durchführen und vollständig und korrekt abschließen kann. Doch genau das wird durch die Optimierungs-Barriere verhindert. Selbst wenn man die Spezifikationsbarriere überwunden hätte und der Auftraggeber somit seine Spezifikationen vollständig und korrekt angeben kann und die Kommunikations-Barriere nicht existieren würde, die Entwickler also die Spezifikationen verstehen würden, gibt es immer noch das Problem, dass der Auftraggeber am Anfang der Entwicklung oftmals gar nicht weiß, was die genauen Anforderungen an das Programmsystem sind. Dieses Problem ist weniger auf mangelndes Interesse des Auftraggebers zurückzuführen, sondern ist substanzieller Natur. Die Anforderungen können erst im Rahmen eines Konkretisierungsprozesses vollständig herausgearbeitet werden.

**Fazit** Aus den drei eben beschriebenen Barrieren ergeben sich mehrere Anforderungen an einen Softwareentwicklungsprozess, der diese überwinden soll. So sollten Softwareentwickler Änderungen an den Spezifikationen in jeder Phase der Softwareentwicklung erwarten und mit ihnen umgehen können. Das ist einer der Gründe warum man iterativ-zyklisch vorgehen sollte. Die Iteration erreicht man, indem alle Phasen zusammen mehrmals durchlaufen werden können. Soll er auch noch zyklisch sein, so sollte man aus bestimmten Phasen in ausgewählte frühere Phasen zurückspringen können.

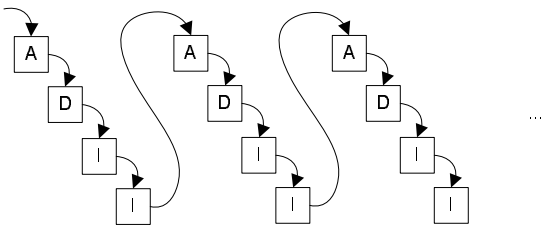


Abbildung 2: Iteratives Prozessmodell

[6] Nach Meinung des Autors aber nicht so gut wie Matthias Rauterberg beschreibt, da sich der technische Kontext nicht auf Algorithmen beschränkt

Wie ein Prozess aussehen kann, der iterativ vorgeht, zeigt Abbildung 2. Soll der Prozess auch noch zyklisches Vorgehen ermöglichen, so muss es zusätzliche Pfeile zwischen den einzelnen Phasen einer Iteration geben. Eine noch höhere Flexibilität lässt sich durch den parallelen Ablauf mehrerer Entwicklungsprozesse erreichen. Problematisch ist hierbei allerdings, dass diese in geeigneter Form synchronisiert werden müssen. Wie das geschehen kann, wird nach der Erläuterung des eigentlichen Entwicklungsprozesses im folgenden Abschnitt erläutert. Aus der Spezifikationsbarriere ergibt sich weiterhin, dass alle Benutzergruppen an der Festlegung der Spezifikation und der Evaluation beteiligt werden müssen. Um das erreichen zu können, muss man visuelle Kommunikationsmittel einsetzen. Ansonsten verhindert die Kommunikations-Barriere, die produktive Kommunikation.

# 3 Lösungsansätze

Abstrahiert man die Unterschiede zwischen traditionellem Wasserfallmodell und iterativ-zyklisch vorgehenden Modellen auf die Ebene der Systemtheorie, so kann man beide an Hand ihrer unterschiedlichen Lenkungsprinzipien unterscheiden. Beim Wasserfallmodell kommt das Lenkungsprinzip der Steuerung zum Einsatz. Das bedeutet, man legt am Anfang ein Ziel fest, bestimmt die Richtung in die man gehen muss um das Ziel zu erreichen und geht genau bis zum Ziel. Die Probleme, die beim Wasserfallmodell in Form der drei Barrieren auftreten, lassen sich ebenfalls auf Ebene der Systemtheorie darstellen. Um die Steuerung einsetzen zu können, muss man genau wissen, wie das zu lenkende System[7] reagiert, um die genaue Richtung bestimmen zu können, welche etwaigen Beeinträchtigungen es gibt und man muss die Zwischenziele und das Endziel kennen, um sie exakt anpeilen zu können.

All das ist aber bei der Softwareentwicklung nicht gegeben. Die Kenntnisse über das System sind am Anfang zu gering, um die genauen Reaktionen des Systems zu kennen. Den Softwareentwicklern fehlt auf Grund der Kommunikations- und Spezifikations-Barrieren möglicherweise der Überblick, wie das Programmsystem im Arbeitsablauf der Organisation eingesetzt werden soll. Mögliche Beeinträchtigungen sind auch noch nicht abzuschätzen, da der Umfang, der sich durch Probleme bei der Spezifikation oder durch Konkrektisierungen (Optimierungs-Barriere) ändernden Anforderungen, unbekannt ist. Schließlich kennt eben durch die Optimierungs-Barriere der Auftraggeber selbst noch nicht einmal das konkrete Ziel, das erreicht werden soll.

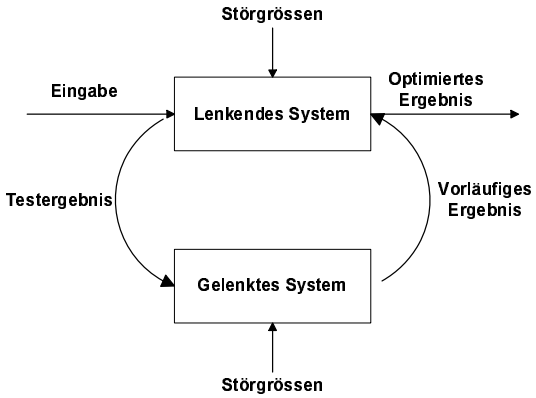


Abbildung 3: Lenkungsprinzip Regelung

Alles im allen ist die Steuerung als alleiniges Lenkungsprinzip bei der Softwareentwicklung unbrauchbar. Sinnvoller ist der Einsatz der Regelung(Abbildung 3). Will man mit ihr zum Ziel gelangen, so geht man jeweils einen Schritt in eine Richtung und überprüft das Ergebnis. Ist man am Ziel angelangt, so ist man fertig, ansonsten geht man in die Richtung, die näher am Ziel ist, weiter. Auf die Softwareentwicklung übertragen erzeugt man vorläufige Ergebnisse (z. B. Versionen) und überprüft, ob dieses Ergebnis schon dem optimierten Endergebnis entspricht oder, ob man an Hand des Testergebnisses noch Verbesserungen vornehmen



[7] Hier die Entwicklung des Programmsystems

kann. Störungen sind kein großes Problem, da sie nur lokal auftreten. Sie führen somit nur zu einer Verlangsamung des Prozesses und lenken ihn nicht wie bei der Steuerung in eine möglicherweise falsche Richtung ab. Auf Grund dieser Eigenschaften stellen die Barrieren der Softwareentwicklung für einen Prozess, der auf dem Lenkungsprinzip der Regelung basiert, keine den Ausgang des Projekts gefährdenden Probleme dar. Insbesondere die Optimierungs-Barriere lässt sich durch das schrittweise Vorgehen hervorragend umgehen.

Setzt man nur die Regelung als Lenkungsprinzip ein, so ist der Prozess wahrscheinlich sehr ineffizient. Vieles lässt sich auch schon im Voraus festlegen. Es ist daher sinnvoll, beide Lenkungsprinzipien miteinander zu kombinieren: Steuerung wird überall dort eingesetzt wo es möglich ist und die Regelung dient dann dazu auf Störungen zu reagieren. Man kann sich das Erreichen des Zieles dann folgendermaßen vorstellen: Zuerst bestimmt man mittels Steuerung die ungefähre Richtung[8] und geht mehrere Schritte in diese Richtung. Vom neuen Punkt aus bestimmt man mittels Steuerung wieder das ungefähre Ziel und geht wieder einige Schritte. Das macht man so lange, bis man am Ziel angekommen ist. Im Rahmen eines iterativ-zyklischen Prozessmodells würde man dann die Phase der Anforderungsanalyse in jeder Iteration für die Festlegung der jeweiligen Steuerungsrichtung benutzen.

## 3.1 Synchronisation

Finden mehrere Entwicklungsprozesse parallel statt, um z. B. Anforderungsänderungen auch in der Implementationsphase zuzulassen, so muss dafür gesorgt werden, dass die Entwicklungsprozesse synchronisiert werden. Ansonsten kann es dazu kommen, dass veraltete Anforderungen implementiert werden. Hierbei unterscheidet man zwischen drei verschiedenen Synchronisationsprinzipien. Beim funktionalen Synchronisationsprinzip wird die Reihenfolge der einzelnen Phasen analog zum Wasserfallmodell vorher festgelegt, was natürlich nicht sinnvoll ist. Beim informationellen Synchronisationsprinzip wird dagegen dafür gesorgt, dass jeder über den Stand in allen parallel ablaufenden Prozessen informiert wird. Beim personellen Synchronisationsprinzip nimmt dagegen jeder an jedem Prozess teil. Problematisch ist hierbei, dass das arbeitsteilige Vorgehen hierdurch verhindert wird, da Programmierer somit auch an der Anforderungsanalyse teilnehmen müssen.

[8] An Hand eines geschätzten Ziels

## 3.2 Globaler Optimierungszyklus

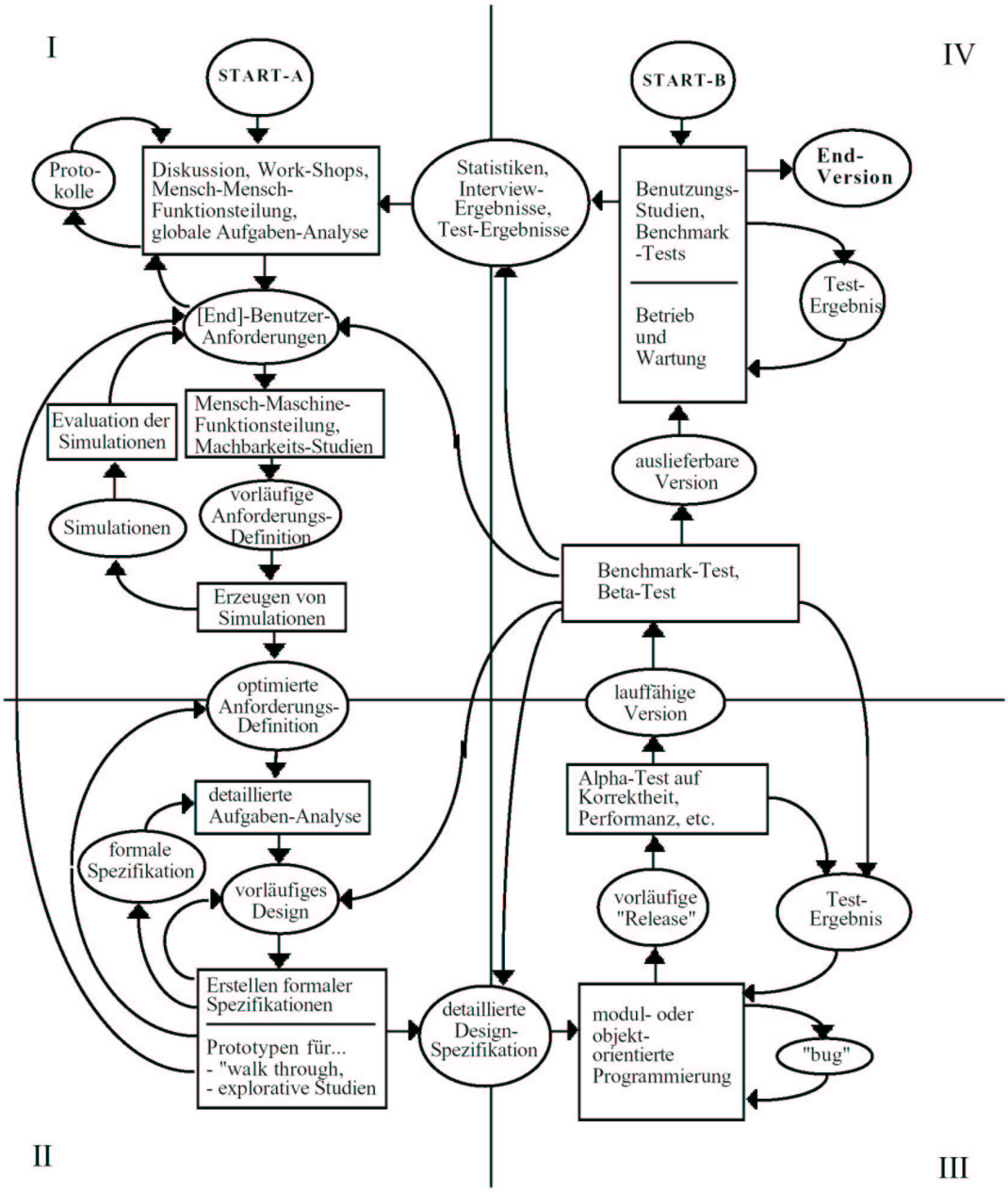


Abbildung 4: Globaler Optimierungszyklus ([Rau95])

Beim globalen Optimierungszyklus (Abbildung 4) handelt es sich um ein Prozessmodell, das von Matthias Rauterberg entwickelt worden ist, um die schon definierten Anforderungen an einen Softwareentwicklungsprozess zu erfüllen. Er soll also iterativ-zyklisch sein, indem das Lenkungsprinzip der Regelung eingesetzt wird, und er soll die Benutzerbeteiligung fördern. Hierbei wird eine Einteilung in vier Phasen, die der des Wasserfallmodells aus Abbildung 1 folgt, vorgenommen. Allerdings werden die Phasen hier noch weiter unterteilt und es wer-

den bestimmte Pfade zwischen diesen Stationen der einzelnen Phasen und den Phasen selbst angegeben.

Den Namen „Globaler Optimerungszyklus“ hat dieses Prozessmodell dadurch erhalten, dass es aus einem großen Zyklus besteht, der bei einem der beiden Startpunkte anfängt, durch jede Phase mindestens einmal geht und schließlich bei einer End-Version im vierten Quadranten endt. Bis dieses Ende erreicht wird können die einzelnen Phasen beliebig oft durchlaufen werden, bis ein optimales Ergebnis erreicht wird.

Neben dem globalen Optimierungszyklus existieren lokale Optimierungszyklen, die zusammengesetzt den globalen ergeben. Ein Beispiel hierfür findet sich im ersten Quadranten. Startet man in Startpunkt A, so geht man in eine Station über, in der man z. B. durch Diskussion die Benutzer-Anforderungen ermittelt. Das Ergebnis dieser Station wird dann überprüft und man kann, wenn es notwendig erscheint, wieder in die vorherige Station zurückkehren. Ist man dagegen der Meinung man hat die Benutzer-Anforderungen erst einmal vollständig erfasst, so kann man in die nächste Station der Anforderungsanalyse übergehen.

Zeigt sich später, dass man doch noch nicht alle Anforderungen erfasst hat, man also in einer späteren Phase (z. B. nach der Entwicklung und Evaluation eines Prototypen während der Designphase) neue Anforderungen gewinnt, so kann man erst einmal an das Ende der Benutzer-Anforderungs-Analyse zurückkehren und bei Bedarf weitere Diskussionen führen. Auf eine vollständige Erklärung aller Stationen und Phasen wird im Folgenden aus zwei Gründen verzichtet. Erstens zeigt Abbildung 4 die Zusammenhänge zwischen den Stationen sehr gut und eine verbale Beschreibung würde ohne signifikant neue Informationen zu bringen mehrere Seiten in Anspruch nehmen und zweitens werden die möglicherweise noch unklaren Stationen im Rahmen der Methoden der Partizipation im folgenden Abschnitt erläutert.

## 3.3 Methoden der Partizipation

Nachdem der Softwarentwicklungsprozess vorgestellt worden ist, müssen nun noch die Methoden der Partizipation beschrieben werden. Rauterberg unterscheidet hierzu vier grundsätzliche Methoden, die dann zu konkreten Methoden spezifiziert werden. Diese vier Methoden sind: Diskussion, Simulation, Prototyping und Versionen. Sie unterscheiden sich nicht nur in der Form der Beteiligung der Benutzer sondern auch in ihren Einsatzbereichen und den Ergebnissen die sie liefern und der Dauer bis ein konkreter Zyklus durchlaufen werden kann.

Beispielhaft sollen hier nur einmal Diskussion und Versionen kurz gegenübergestellt werden, die ausführliche Beschreibung der Methoden in ihren verschiedenen Ausprägungen findet dann im Folgenden statt. Die Diskussion ist eine direkte verbale oder visuelle Kommunikation. Ein Zyklus, also die Zeit, die man braucht um ein Ergebnis zu erreichen und zu validieren, dauert oft nur Minuten. Man kann die Diskussion am Besten am Anfang der Entwicklung einsetzen, wenn es noch keine anderen greifbaren Ergebnisse gibt. Andere Partizipationsmethoden funktionieren hier nicht, da man für deren Vorbereitung Informationen braucht, die man erst durch die Diskussion erhalten kann. Ganz im Gegensatz dazu steht die Methode der Versionen. Hier bekommen die Anwender ein einigermaßen fertiges Softwareprodukt, das sie direkt benutzen können. Da es direkt in der Arbeitsumgebung der Benutzer eingesetzt werden kann, gewinnt man durch diese Methode Informationen darüber wie gut sich das Programmsystem in diese integriert. Da die Benutzer das System sinnvoll einsetzen sollen, steckt in ihm ein sehr großer Entwicklungsaufwand, so dass sich ein Zyklus über Monate und Jahre hinziehen kann.

**Diskussion** Bei der Diskussion handelt es sich um die am häufigsten eingesetzte Partizipationsmethode. Es gibt sie in zwei Varianten: In der ersten beruht sie einzig und allein auf verbaler Kommunikation, während in der zweiten Variante zusätzlich noch visuelle Hilfsmittel wie zum Beispiel Flip-Charts oder die Meta-Plan-Technik[9] benutzt werden. Die zweite Variante hat den Vorteil, dass sie sich nicht so sehr auf das gesprochene Wort verlässt und damit die Kommunikations-Barriere weniger deutlich zum Tragen kommt. Dennoch kann es auch bei ihr leicht zu Missverständnissen kommen. Beide Methoden haben den Vorteil, dass sie sehr schnell durchzuführen sind. Man hat je nach Variante nach Sekunden bis Minuten(Variante 1) oder Minuten bis Stunden (Variante 2) ein Ergebnisdas man überprüfen kann.

Beide Varianten der Diskussion sind in erster Linie während der initialen Anforderungsanalyse, wenn man also noch keine vorzeigbaren Ergebnisse hat, sinnvoll. Später, wenn man etwas hat, über das man sprechen kann, sind die anderen Methoden meist sinnvoller. Es kann allerdings trotzdem gut sein, die Diskussionsmethode auch dann noch einzusetzen, da sie den Blick auf vollkommen unbeachtete Teilgebiete lenken kann, während man sich bei den anderen Methoden möglicherweise in Einzelheiten des konkreten Ergebnisses verstrickt.

**Simulation** Die Simulation dient dazu ein visuell wahrnehmbares Ergebnis zu erreichen, das das zu entwickelnde Programmsystem in geeigneter Form simuliert. Auch hier unterscheidet man zwischen zwei Varianten, die unterschiedliche Teilaspekte des zu simulierenden System abbilden. In der ersten Variante werden Handskizzen oder Maskenlayouts benutzt, um den Benutzern die Benutzer-Schnittstelle des Systems deutlich zu machen. Bei der zweiten Variante setzt man dagegen formalere Beschreibungstechniken, z. B. Ablaufpläne ein. Hierbei ist problematisch, dass sich diese formalen Techniken den Benutzern auf Grund fehlender Qualifikation oftmals nicht erschließen. Sie können daher nicht oder nur ungenügend beteiligt werden.

Problematisch ist bei beiden Varianten der Simulation außerdem, dass sie zwar das Programmsystem simulieren können, aber nicht den konkreten Arbeitskontext der Benutzer. Diesen fehlt dann der Bezugspunkt, um die Fähigkeiten und Eigenschaften des Systems sinnvoll bewerten zu können. Im Gegensatz zu den Diskussionmethoden kann man dafür aber bei der Simulation sehr detailliert vorgehen und praktisch das gesamte System simulieren. Dieser Detailreichtum wird aber mit einem sehr hohen Arbeitsaufwand erkauft, ohne dass dieser sich im späteren System niederschlägt, da man die Simulationen ja parallel zum Anwendungssystem beibehält. Will man also umfangreiche Simulationen durchführen, so erscheint es sinnvoller, die Prototyping- oder Versionen-Methode einzusetzen, da man die aus ihnen gewonnenen Ergebnisse im konkrekten Programmsystem einsetzen kann.

Simulationen sollte man vor allem im Rahmen des Übergangs von der Anforderungsanalyse zur Designphase einsetzen. Man kann somit Erkenntnisse, die man aus der Diskussion gewonnen hat, schneller umsetzen und evaluieren. Das ist aber wie gesagt nur sinnvoll, wenn die Simulationen nicht zu detailliert werden.

**Prototyping** Während die Simulation Ergebnisse liefert, die man von Hand z. B. auf einem Blatt Papier durcharbeiten muss, bietet das Prototyping die Möglichkeit bedienbare Computerprogramme zu erzeugen. Man unterscheidet hierbei drei Arten von Prototyping. Beim horizontalen Prototyping liegt der Schwerpunkt

[9] Spezielle Moderationstechnik, die sich visueller Hilfsmittel (Karten und Pinnwand) bedient um Informationen zu sammeln und zu ordnen

in der Darstellung der Dialogfolge und Dialogstruktur. Daher sind bei ihm keine anwendungsbezogene Funktionen implementiert. Der partielle vertikale Prototyp implementiert dagegen einige und der vollständige vertikale Prototyp alle Anwendungsfunktionen zumindest ansatzweise. Beide vertikale Prototypen dienen dazu, zu überprüfen, ob die Anforderungen an die Funktionalität korrekt umgesetzt wurden und ob die bisher gestellten Anforderungen ausreichen. Alle drei Varianten des Prototyping werden in erster Linie am Ende einer Design-Phase eingesetzt. Die in dieser Phase gewonnenen Erkentnisse werden dann in den Prototypen umgesetzt. Je nach Ergebnis des Einsatzes des Prototypen durch die Benutzer geht man dann entweder in die Implementationsphase über, verbessert das Design oder erstellt komplett neue Anforderungen.

Das Prototyping ist ein sehr wirksames Kommunikationsmittel zwischen Benutzern und Entwicklern, da die Benutzer keine zusätzlichen Qualifikationen benötigen. Weiterhin können sie ein System benutzen, das dem fertigen System sehr ähnlich sieht und sie können den Prototypen in ihrem Arbeitsumfeld einsetzen. Allerdings muss zwischen Anwendern und Entwicklern klar sein, was der Prototyp ist und was nicht. Gehen die Benutzer davon aus, dass es sich um ein vollständig benutzbares Produkt ähnlich einem Prototypen in der Autoindustrie handelt, so stellen sie zu hohe Anforderungen an den Prototypen. Tritt dieses Missverständnis auf, so könnte es sein, dass die Entwickler keine neuen Prototypen mehr erzeugen wollen, da sie bewusst unvollständige Software ausliefern und dafür Kritik von den Benutzern akzeptieren müssen.

Problematisch ist schließlich auch noch, dass ein Prototyp die Sicht auf grundsätzlich andere Lösungen verstellen kann. Man versucht dann nur die vorhandene Lösung immer weiter zu verbessern, obwohl es möglicherweise eine deutlich bessere Lösung gibt, die aber eine komplett andere Herangehensweise verlangt. Außerdem sind die Benutzer keine ausgebildeten Softwareentwickler. Gehen die eigentlichen Softwareentwickler auf alle ihre Vorschläge ein, so verschlechtern sie möglicherweise nur das Resultat. Der alleinige Einsatz von Prototypen kann also zu sub-optimalen Lösungen führen. Setzt man sie dagegen kombiniert mit anderen Methoden ein, die dafür sorgen, dass der gesamte Lösungsraum betrachtet wird, sind sie sehr sinnvoll.

**Versionen** Setzt man den globalen Optimierungszyklus als Prozessmodell ein, so benutzt man die Versions-Methode implizit mit. Durchläuft man den gesamten globalen Optimierungszyklus einmal, so bekommt man eine Version des Programmsystems. Durch Benutzungsstudien oder Benchmarks, kommt man dann entweder zu dem Ergebnis, dass das System noch verbessert werden muss und geht in eine der drei vorherigen Phasen über oder kommt zu einer End-Version. Man kann sie aber auch als Erweiterung der Prototyping-Methode sehen. Wird ein vertikaler Prototyp mit immer mehr Funktionalität erweitert, so geht er praktisch in eine Version des Programmsystems über.

Die Versions-Methode hat ähnliche Eigenschaften wie die Prototyping-Methoden. Doch für sie gilt noch stärker: Alleine eingesetzt macht sie wenig Sinn. Problematisch ist hierbei insbesondere die lange Zyklusdauer, die Monate bis Jahre dauern kann, und den Kontakt zwischen Benutzern und Entwicklern zum Abbruch bringen kann. Als Hauptvorteil der Versionen-Methoden gilt, dass die einzelnen Versionen im Arbeitsalltag der Benutzer eingesetzt werden. Erst hier können Probleme, die sich aus der Integration in den Arbeitsablauf einer Organisation ergeben, vollständig erkannt werden. Auch Verbesserungen an der Benutzer-Schnittstelle oder viele Bugs werden erst durch die tägliche Benutzung vieler Menschen deutlich.

**Zusammenfassung** Wie schon deutlich wurde, lässt sich keine Methode sinnvoll alleine einsetzen. Jede Methode hat ihre Stärken und Schwächen. Erst im Zusammenspiel und dem Einsatz in der richtigen Entwicklungsphase werden die Stärken betont und die Schwächen durch die anderen Methoden abgemildert. Eine Kombination der Methoden, wie sie der globale Optimierungszyklus in Abbildung 4 vorschlägt, scheint sehr sinnvoll.

Eine weitere interessante Möglichkeit der Benutzerbeteiligung wird bei [JM02] beschrieben. Sie erlaubt es, aus natürlich-sprachlichen Texten Modelle zu erzeugen. Die Benutzer können dann in diesen Texten bestimmte Anforderungen beschreiben, die dann automatisch in Modelle umgesetzt werden können.

## 3.4 Probleme partizipativer Methoden

Gerade weil die partizipativen Methoden so viel Erfolg beim Überwinden der Barrieren versprechen, können bei ihnen natürlich auch Probleme auftreten, da sie alte Arbeitsmuster durchbrechen. Diese Probleme beschreibt Matthias Rauterberg als sechs-dimensionales Konzept der Benutzer-Entwickler-Distanz. Es gibt also sechs verschiedene Formen der Benutzer-Entwickler-Distanz, die in unterschiedlicher Stärke in jedem Projekt vorkommen.

Die anwendungsbezogene Distanz tritt auf, wenn die Entwickler zu wenig über den Anwendungskontext wissen. Problematisch wird sie erst, wenn zur Partizipation Benutzer geschickt werden, die in den jeweiligen Fachabteilungen auf Grund fehlender Qualifikation entbehrlich sind oder die Qualifikation erst im Rahmen der Beteiligung an der Entwicklung erlangen sollen. Dann wird die Distanz zwischen Entwicklern und beteiligten Benutzern zwar sehr schnell überwunden sein, doch spiegelt das nicht die wirkliche Distanz zwischen Entwicklern und allen Anwendern wieder.

Man muss dem Auftraggeber also deutlich machen, dass man zur Partizipation gut ausgebildete Benutzer braucht, die den konkrekten Arbeitskontext genau kennen. Problematisch ist hierbei aber häufig, dass der Auftraggeber genau diese Benutzer nicht entbehren kann. Wenn sich jemand entscheidet, den Arbeitsablauf in seinem Unternehmen durch Software zu optimieren, so macht er dies nur, weil es der billigste Weg ist, um mehr Aufträge abwickeln zu können. Könnte er stattdessen einfach zehn neue Arbeiter einstellen, die im Endeffekt billiger und sofort verfügbar wären, so würde er diesen Weg gehen.

Damit der Auftraggeber im Zeitraum der Softwareentwicklung keinen Verlust macht, müssen seine bisherigen Arbeitnehmer genausoviel arbeiten wie bisher. Fällt einer von ihnen aus, so könnte sein möglicherweise vollkommen überlastetes Unternehmen wie ein Kartenhaus zusammenbrechen. Die Schwierigkeit liegt hier also darin, dem Auftraggeber deutlich zu machen, wie wichtig die Beteiligung qualifizierter Benutzer bei der Softwareentwicklung ist. Dann hat der Auftraggeber genug Informationen, um die beiden Interessen gegeneinander abzuwägen.

Schickt der Auftraggeber qualifizierte Benutzer zu den Treffen mit den Entwicklern, so kann es dennoch zu Problemen kommen, wenn diese wirklich so wichtig für das Unternehmen sind, wie bei der anwendungsbezogenen Distanz beschrieben. Dann kann nämlich die zeitliche Distanz auftreten. Die Benutzer beteiligen sich zwar, doch ist deren Zeit so kostbar, dass die Beteiligung nur zu festgelegten Zeitpunkten stattfinden kann. Gibt es dann Fragen von Seiten der Entwickler, so müssen sie entweder warten oder auf gut Glück weiterarbeiten. Diese Art der Distanz lässt sich kaum überbrücken, der Auftraggeber kann nicht einfach neue Leute einstellen, weil es sie entweder nicht gibt oder der Aufwand sie einzuarbeiten zu gross ist. Man muss dann einfach in Kauf nehmen, dass die Entwicklung der Software länger dauert.

Die qualifikatorische Distanz tritt auf, wenn den Benutzern die Qualifikation fehlt die Analyse- und Beschreibungstechniken der Entwickler einzusetzen oder zu verstehen. Sie können somit nur unzureichend beteiligt werden. Die Entwickler müssen sich dann selber zu Experten auf dem Gebiet des Anwendungskontextes ausbilden. Das kann innerhalb einer begrenzten Projektdauer nicht funktionieren. Es ist also sinnvoll, auf komplizierte Techniken zu verzichten. Schließlich müssen sich aber auch die Entwickler soweit in den Anwendungskontext einarbeiten, dass sie wenigstens die richtigen Fragen stellen können. Ansonsten kann es zu keiner sinnvollen Kommunikation kommen.

Findet die Entwicklung und Benutzung der Software nicht innerhalb der gleichen Organisation statt, so ist die organisationale Distanz sehr hoch. Sie existiert aber auch innerhalb einer Organisation zwischen verschiedenen Tochterfirmen oder Abteilungen. Je größer die Distanz zwischen Entwicklern und Auftraggebern ist, um so umfangreicher sind die vertraglichen Vereinbarungen, die beide Seiten absichern. Solche umfangreichen Vereinbarungen können natürlich die Benutzerbeteiligung hemmen, da z. B. die Entwickler den Benutzern nicht unbedingt allzu deutlich machen wollen, dass sie mit der Entwicklung im Rückstand sind. Auch die Benutzer wollen (oder sollen) möglicherweise nicht zu viel über Betriebsinterna berichten, da die Entwickler ja auch für Konkurrenten arbeiten können.

Ein großes Problem kann die motivationale Distanz sein. Sie tritt auf, wenn die Benutzer den Eindruck haben, dass sie zwar immer wieder befragt werden, aber eigentlich keinen wirklichen Einfluss auf die Entwicklung haben. Noch gefährlicher ist es, wenn die Benutzer den Eindruck haben, sie werden nur an der Entwicklung beteiligt, damit man sie durch die Software ersetzen kann. Solche Benutzer haben natürlich wenig Lust bei der Entwicklung zu helfen. Es ist daher unbedingt erforderlich, dass die Benutzer regelmäßig über den Entwicklungsstand der Software unterrichtet werden. Sie sehen so, dass sich ihre Ideen in der Software wiederfinden und dass die Software sie nicht ersetzen, sondern unterstützen soll.

Ist die organisationale Distanz ein großes Problem, dann kommt meistens auch noch die räumliche Distanz hinzu. Sie tritt auf, wenn Entwickler und Anwender weit voneinander entfernte Arbeitsstätten haben. Die Entwickler haben dann wenig Möglichkeit die Software im realen Einsatz zu sehen und für die Benutzer ist die Beteiligung an der Entwicklung mit einem hohen Reiseaufwand verbunden. In einem solchen Fall kann es sinnvoll sein die Arbeitsstätten eines oder mehrerer Benutzer komplett zu den Entwicklern zu verlagern. Die Entwickler können die Benutzer jederzeit befragen und durch die enge Zusammenarbeit kann eine ungezwungenere Kommunikation entstehen, als wenn die Benutzer extra zu den Entwicklern reisen müssen und die Kommunikation dann in einem festgelegten Rahmen stattfinden muss.

Neben den Distanzen, die Rauterberg beschreibt, gibt es beim Einsatz von partizipativen Methoden noch weitere Probleme. Bei der Beschreibung der motivationalen Distanz wurden schon Gründe genannt, warum sich Benutzer nur unzureichend beteiligen. Ähnliche Probleme kann es auch von Seiten der Softwareentwickler geben, wenn diese sich gegen die Benutzerbeteiligung wehren, weil sie darin keinen Sinn sehen. Ein Projektmanager sollte sich dann überlegen, ob es überhaupt Sinn macht, solche Leute weiter zu beschäftigen. Ein weiterer Grund kann aber auch sein, dass den Softwareentwicklern die nötige Qualifikation fehlt. Dann muss dafür gesorgt werden, dass die Methoden der Benutzerbeteiligung den entsprechenden Softwareentwicklern beigebracht werden. Ist die Qualifikation bei Einigen schon vorhanden, so sollte die Ausbildung der Übrigen am Besten im Rahmen des realen Einsatzes der Methoden durch erfahrene Softwareentwickler geschehen.

Unabhängig vom Einsatz partizipativer Methoden, sollten neben Fachleuten für die Softwareentwicklung und für den gegebenen Anwendungskontext auch Arbeits- und Organisationsexperten eingesetzt werden. Durch sie kann eine bessere Integration des Programmsystems in den Arbeitskontext geschehen, da sie abstrahiert von domänen-spezifischen Eigenschaften den eigentlichen Arbeitsablauf modellieren und bewerten können.

# 4 Moderne Softwareentwicklungsprozesse

Im Folgenden werden zwei moderne Softwareentwicklungsprozesse vorgestellt. Die vollständige Vorstellung beider Prozesse würden den Rahmen dieser Ausarbeitung sprengen. Daher wird jeweils nur eine kurze Einführung in den Prozess gegeben und dann auf die Mittel der Benutzerbeteiligung eingegangen.

## 4.1 Extreme Programming (XP)

Bei Extreme Programming handelt es sich um einen sogenannten agilen Softwareentwicklungsprozess. Das bedeutet nach [Fow02] in erster Linie zweierlei: Erstens, dass der Prozess adaptiv ist, und zweitens, dass er personenorientiert ist. Adaptiv bedeutet, dass man nicht alles im Voraus plant, sondern versucht, so zu arbeiten, dass man mit neuen Erkenntnissen und Anforderungen leicht umgehen kann. Beim Extreme Programming wird das als eines der Grundprinzipien angenommen und als „embrace change“[10] bezeichnet und z. B. durch ein iteratives Vorgehen erreicht. Insgesamt gibt es 15 dieser Grundprinzipien, die keine direkten Handlungsanweisungen sind, sondern nur Grundlagen, an denen man sich beim Einsatz von Extreme Programmierung orientieren kann. Weitere Grundprinzipien sind z. B. offene und ehrliche Kommunikation (insbesondere auch zwischen Entwicklern und Anwendern) und arbeiten mit den Instinkten der Menschen, nicht gegen sie (Menschen wollen erfolgreich sein, etwas lernen etc.). Die restlichen 12 Prinzipien kann man bei [Bec00] auf den Seiten 37ff nachlesen.

Mit den beiden letztgenannten Prinzipien sind wir auch schon bei der zweiten wichtigen Eigenschaft agiler Prozesse angelangt. Ein personen-orientierter Softwarentwicklungsprozess ist weniger starr an bestimmten Abläufen orientiert, sondern definiert eher bestimmte Praktiken (Best Practices), die erfolgversprechend sind. Beim Extreme Programming werden 12 dieser Praktiken definiert. Im Folgenden werden nur die Praktiken, die der Benutzerbeteiligung dienen, vorgestellt. Allerdings muss auch ein personen-orientierter Prozess eine gewisse Ordnung haben. Dazu wird beim Extreme Programming zwischen vier Aktivitäten unterschieden: Zuhören, Designen, Programmieren, Testen. Außerdem wird die Entwicklung grob in drei Phasen eingeteilt: Exploration, Commitment und Steering. Ein so umfangreiches Modell, wie es der globale Optimierungszyklus in Abbildung 4 mit Stationen innerhalb der Phasen und vordefinierten Pfaden zwischen diesen Stationen vorschlägt, fehlt aber.

Außerdem führt Kent Beck beim Extreme Programming neben dem eigentlichen Softwarentwicklungsprozess auch ein einfaches ökonomisches Modell der Softwareentwicklung ein (siehe [Bec00] S. 11ff), das sich an drei Faktoren und vier Variablen orientiert. Die drei Faktoren sind Einnahmen und Ausgaben, der Zinssatz und die Wahrscheinlichkeit, dass das Projekt erfolgreich ist. Um den ökonomischen Wert eines Projekts zu erhöhen, muss man nach diesem einfachen Modell nur einen der Faktoren ändern. Also entweder, die Ausgaben senken, die Einnahmen erhöhen, Geld später ausgeben und früher einnehmen, so dass man weniger Zinsen zahlen muss oder die Wahrscheinlichkeit, dass

[10] Deutsch: Veränderungen annehmen

das Projekt erfolgreich ist, erhöhen. Die eigentliche Entwicklung bestimmen dann die vier Variablen Kosten, Zeit, Qualität und der Bereich, den die Software umfassen soll. Ein Projektmanager muss diese vier Variablen in Einklang bringen. Hat er nur begrenzt viel Zeit zur Verfügung, um ein Ziel zu erreichen, so muss er entweder den Umfang der Software reduzieren oder die Qualität wird leiden.

Nachdem nun der eigentliche Prozess erklärt ist, werden im Folgenden die Praktiken, die der Benutzerbeteiligung dienen, erläutert. Das sind vier der zwölf Praktiken: Planspiel, kleine, häufige Veröffentlichungen, Tests durch den Kunden und der sogenannte On-Site Customer.

Das Planspiel dient dazu die technischen und wirtschaftlichen Überlegungen miteinander zu verbinden. Wirtschaftliche Überlegungen sind z. B. der Umfang der Entwicklungen und das Veröffentlichungsdatum, die beide festgelegt werden müssen. Die Informationen hierzu, wie z. B. Schätzungen für die Entwicklungsdauer bestimmter Eigenschaften des Programmsystems, liefern die technischen Überlegungen.

Um diese beiden Sichten auf das System zu vereinen, schreiben die Personen, die die wirtschaftliche Perspektive repräsentieren[11] „Geschichten“ auf, die jeweils eine gewünschte Eigenschaft des Programmsystems repräsentieren. Die Personen, die die technische Perspektive einnehmen[12] schätzen dann entweder den Arbeitsaufwand für diese Eigenschaft ab oder sagen, dass die Eigenschaft zu komplex ist und in mehrere Geschichten aufgeteilt werden muss. Als nächstes sortieren die Kunden die Eigenschaften nach Wichtigkeit und die Entwickler teilen den Kunden mit, wie schnell sie entwickeln können.

Die Kunden legen dann den Umfang der ersten Iteration fest, indem sie entweder ein festes Datum oder eine Teilmenge der Eigenschaften auswählen. Daraufhin können die Entwickler anfangen die geplanten Eigenschaften zu realisieren. Während einer Iteration können die Kunden Eigenschaften, die entwickelt werden sollen, gegeneinander austauschen, wenn die neue Eigenschaft nicht mehr Zeit in Anspruch nimmt als die alte. Die Entwickler können den Aufwand der einzelnen Eigenschaften anpassen oder, wenn sie sich überschätzt haben, die Kunden eine Teilmenge der Eigenschaften, die unbedingt fertig gestellt werden müssen, auswählen lassen. Am Ende einer Iteration steht dann ein Programmsystem, das genau die Eigenschaften hat, die vorher festgelegt wurden. Sind noch Eigenschaften übrig, die die Kunden im Programmsystem sehen wollen, so startet die nächste Iteration mit der Auswahl aus den vorhandenen gewünschten Eigenschaften oder durch Hinzufügen neuer gewünschter Eigenschaften durch die Kunden.

Das Planspiel macht nur Sinn, wenn es mehrere dieser Iterationen gibt, so dass die Kunden wichtige Eigenschaften zuerst entwickeln lassen. Diese Iterationen sollten relativ kurz sein, damit die Kunden schnell Ergebnisse sehen und die Anforderungen anpassen können. Kent Beck schlägt Veröffentlichungszyklen von einem bis zwei Monaten vor([Bec00]).

Durch Tests, die die Kunden ausführen, können diese sicherstellen, dass das Programmsystem genau die Eigenschaften hat, die sie gefordert haben. Diese Tests können entweder manuell durch die Kunden ausgeführt werden oder aber unter Zuhilfenahme der Entwickler automatisiert werden.

Die letzte etwas ungewöhnliche Praktik ist schließlich der On-Site Customer. Hierbei handelt es sich um einen Kunden, der seinen Arbeitsplatz bei den Entwicklern hat. Das sollte jemand sein, der das System auch später wirklich einsetzt. Ihn können die Entwickler dann jederzeit fragen. Problematisch ist hierbei natürlich,

[11] Im Folgenden als Kunden bezeichnet
[12] Im Folgenden als Entwickler bezeichnet

dass der Auftraggeber oftmals einen realen Benutzer kaum entbehren kann. Die Gründe hier für wurden bei der Erläuterung der zeitlichen Distanz beschrieben.

Die oben genannten Praktiken können einige der Probleme lösen. Hierzu zählen insbesondere die anwendungsbezogene und qualifikatorische Distanz, durch das relativ einfache Planspiel, die motivationale Distanz durch die häufigen Veröffentlichungen und die vom Kunden ausgeführten Tests und die räumliche Distanz durch den On-Site Customer. Gerade durch das Planspiel und die häufigen Veröffentlichungen wird auch die Flexibilität in Hinsicht auf sich ändernde Anforderungen erhöht. Leider werden außer dem Planspiel keine Techniken der Benutzerbeteiligung ausführlich erläutert. So werden nur die Methoden der Diskussion durch das Planspiel und der Versionen durch die häufigen Veröffentlichungen abgedeckt. Das Erzeugen von Simulationen und expliziten Prototypen wird dagegen nicht behandelt. Das stimmt aber auch mit der Definition von Extreme Programming als einen agilen Softwareentwicklungsprozess überein. Beide Methoden erzeugen nämlich Artifakte, die später nicht unbedingt Teil des Programmsystems sind, aber nebenbei gewartet werden müssen. Das XP-Prinzip hierzu lautet „travel light“[13].

## 4.2 Rational Unified Process (RUP)

Beim Rational Unified Process ([Kru98]) handelt es sich im Gegensatz zu Extreme Programming um einen deutlich „schwereren“ Prozess. Dennoch gibt es Ansätze, beide Prozesse miteinander zu kombinieren. Während ([Pol01] beschreibt, wie man die Techniken und Methoden von Extreme Programming im Rational Unified Process benutzen kann, wird bei [Mar02] beschrieben wie man aus dem Rational Unified Process einen Prozess erzeugen kann, der in etwa die Eigenschaften von Extreme Programming hat. Hierbei wird auch schon deutlich, dass es sich bei RUP nicht nur um einen einfachen Prozess handelt, sondern eher um ein Prozess-Framework, das man je nach Projektgröße anpassen kann. Außerdem wird er von Rational Software als Prozess-Produkt vermarktet.

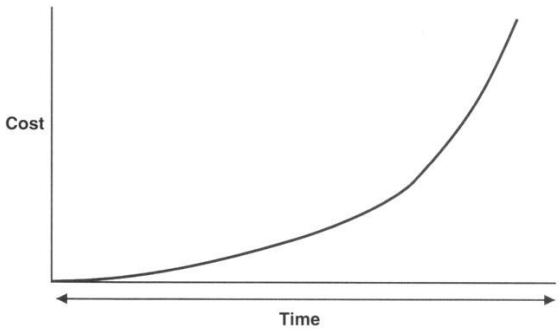


Abbildung 5: Cost of change (aus [Kru98])

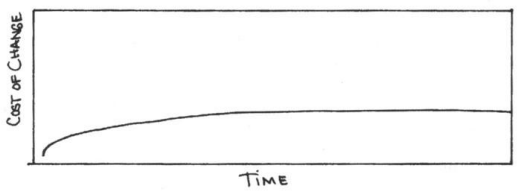


Abbildung 6: Cost of change (aus [Bec00])

Trotz der Möglichkeit XP mittels RUP zu emulieren, muss dennoch deutlich gemacht werden, dass es sich um zwei grundverschiedene Herangehensweisen an die Softwareentwicklung handelt. Während es sich beim RUP um einen architektur-zentrierten Prozess handelt, wird das Wort Architektur bei XP kaum verwendet. Anschaulich wird der Unterschied, wenn man die sowohl bei [Kru98] als auch bei [Bec00] vorkommenden Kurven betrachtet, die den Aufwand für Änderungen im Verlaufe des Prozesses zeigen. Während man beim RUP davon ausgeht, dass die Kosten etwas zu ändern,

[13] Deutsch: Mit wenig Gepäck reisen

exponentiell über die Projektdauer steigen (Abbildung 5), geht man beim XP davon aus, dass es zwar einen Anstieg gibt, dieser aber durch die Praktiken des XP, gering gehalten werden kann (Abbildung 6). Der Ansatz von XP widerspricht damit klassischen Lehrmeinungen, die davon ausgehen, dass man von Anfang an so planen sollte, dass man später möglichst wenig Änderungen am Grundkonzept vornehmen muss. Dass ein solcher Ansatz problematisch ist wurde schon im Rahmen der Barrieren der Softwareentwicklung erläutert. Auch RUP ist hier aber kein Rückschritt auf wasserfallmodellähnliche Prozesse, sondern erreicht seine Flexibilität durch ein iteratives Vorgehen. Das intensivere Planen und Entwerfen einer Architektur im Vergleich zum XP, dient vor allem bei größeren Projekten dazu, die Risiken zu minimieren. Wie sich XP in einem solchen Umfeld verhält, wurde noch nicht wissenschaftlich untersucht, doch geht man im Allgemeinen davon aus, dass es nur für Projektgrössen mit 10-20 Entwicklern produktiv ist.

Nachdem RUP jetzt mit XP verglichen worden ist und der Hauptunterschied, nämlichm, die Architektur-Basiertheit herausgestellt worden ist, soll nun der Prozess im Einzelnen erläutert werden. Hierbei wird man viele Elemente des XP, teils stärker formalisiert wiederfinden. So findet man auch bei RUP die Best Practices wieder. Allerdings bedeuten sie hier etwas anderes und sind eher mit den Grundprinzipien bei XP zu vergleichen. Sie definieren grundlegende Anforderungen an die Softwareentwicklung wie zum Beispiel das iterative Vorgehen oder die Modellierung mittels der UML.

Die eigentlichen Arbeitsanweisungen werden mittels Aktivitäten beschrieben. Aktivitäten sind hierbei komplexere Arbeitsvorgänge, die in einzelne Schritte aufgeteilt werden können. Das Ergebnis einer Aktivität ist dann ein Artifakt. Ein großer Unterschied zum Extreme Programming sind die sogenannten Worker. Das sind Rollen, die einzelne Mitglieder eines Entwicklungsteams einnehmen können. Dass es insgesamt 27 dieser unterschiedlichen Rollen gibt, macht deutlich, dass RUP sehr viel komplexer als XP ist. So gibt es zum Beispiel alleine vier Worker, die die verschiedenen Arten des Testens übernehmen. Das Zusammenspiel von Workern, Aktivitäten und Artifakten wird durch die Workflows gewährleistet. In ihnen wird definiert, welche Aktivitäten durch welche Worker ausgeführt werden und welche Aktivitäten aufeinander folgen. Die Workflows sind in neun Core Workflows, die innerhalb des Prozess-Produktes von Rational Software definiert sind, und in die Iteration Workflows, die bei einer konkreten Instanziierung eines Prozesses selbst definiert werden müssen, aufgeteilt.

Schließlich wird die Entwicklung beim RUP auch noch in vier Phasen eingeteilt. Entgegen dem Beispiel in Abbildung 2 besteht aber nicht jede Iteration aus allen Phasen, sondern die einzelnen Phasen aus mehreren Iterationen. Die Phasen Inception [14], Elaboration[15], Construction[16] und Transition[17] stellen somit den Lebenszyklus eines Softwareprodukts dar. Wie das aussieht und wie umfangreich die einzelnen Core Workflows in den einzelnen Iterationen eingesetzt werden zeigt Abbildung 7. Die Unterscheidung zwischen Core Process und Core Supporting Workflows spielt hier keine Rolle.

Beim RUP wird die Benutzerbeteiligung in erster Linie durch zwei Core Workflows sichergestellt: Im Business Workflow werden die Struktur einer Organisation und die Beziehungen in ihr modelliert. Beim Requirements Workflow werden die Anforderungen an das zu entwickelnde System definiert. Bei beiden Workflows

[14] Deutsch: Anfang
[15] Deutsch: Entwicklung
[16] Deutsch: Konstruktion
[17] Deutsch: Übergang

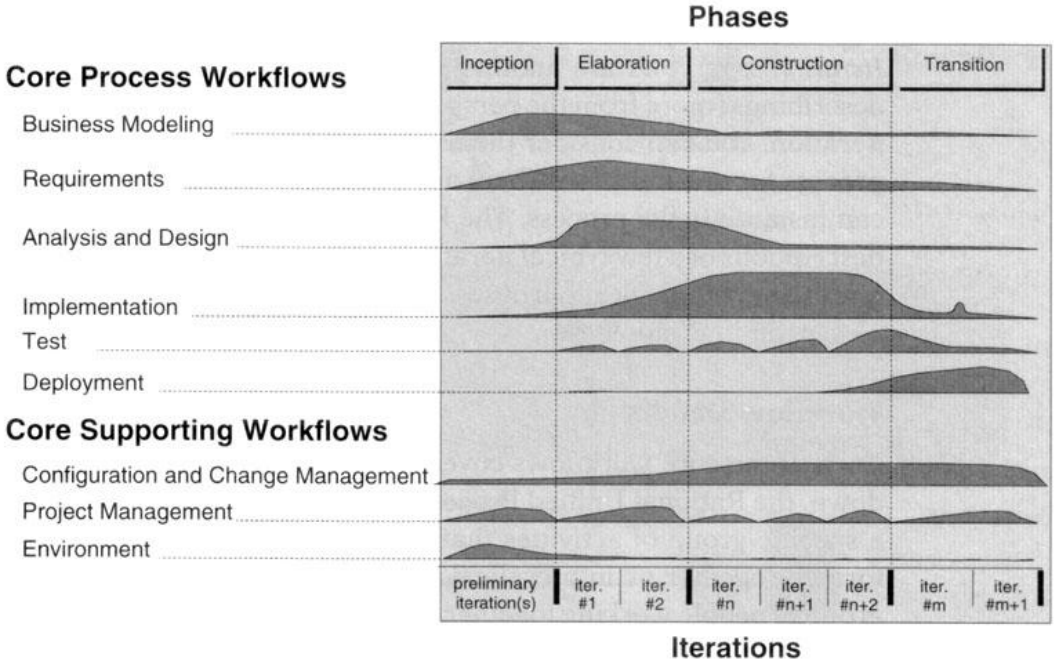


Abbildung 7: Phasen beim RUP (aus [Kru98])

werden die Benutzer mit einbezogen um die nötigen Informationen zu bekommen. Die Techniken hierzu werden leider bei [Kru98] nicht beschrieben. Ein wichtiger Teil der Benutzerbeteiligung ist aber auch die Durchführung von Akzeptanztests, die aus den Use Cases gewonnen werden. Ähnlich wie beim XP werden diese von den Benutzern selbst ausgeführt und ermöglichen diesen somit einen Einblick in den Stand der Entwicklungen. Abschließend sei noch angemerkt, dass es trotz der vielen verschiedenen Rollen, die der RUP definiert, keine explizite Benutzerrolle gibt, die Entwickler oder aber reale Benutzer im Rahmen der Entwicklung einnehmen können, um im Rahmen von Workflows bestimmte Ergebnisse zu bewerten. Hierzu gibt es nur die jeweiligen Analystenrollen, die aber natürlich eine andere Perspektive als die Benutzer haben.

# 5 Fazit

Wie hoffentlich deutlich geworden ist, ist die Beteiligung aller Benutzergruppen an der Softwareentwicklung unerlässlich, um erfolgreiche Software entwickeln zu können, die wirklich das macht, was die Benutzer erwarten. Dass diese Beteiligung der Benutzer nicht unproblematisch ist, ist ebenfalls klar. Hierbei gibt es zwei verschiedene Problemfelder, die zu beachten sind.

Erstens muss Auftraggebern und Benutzern die Wichtigkeit der Beteiligung erläutert werden. Software kann man nicht wie ein Auto kaufen, indem man eine Probefahrt macht und dann bezahlt. Sie ist ein hochkomplexes System, das in ein noch komplexeres System, nämlich den Arbeitsablauf einer Organisation, eingebunden werden muss. Insbesondere den Benutzern muss außerdem deutlich gemacht werden, dass die Software sie nicht ersetzen[18], sondern in ihrem Arbeitsalltag unterstützen soll. Je besser sie sich an der Softwareentwicklung beteiligen, um so besser wird die Software, die sie später tagtäglich einsetzen müssen.

Das zweite Problemfeld tritt bei den Softwareentwicklern auf. Im Rahmen heutiger Ausbildungen werden Methoden der Benutzerbeteiligung nur unzureichend gelehrt. Damit die Softwareentwickler sich aktiv mit den Benutzern auseinandersetzen können, muss sich das ändern. Außerdem sollte den Softwareentwicklern während ihrer Ausbildung auch die andere Seite der Softwareentwicklung, die Benutzerseite, erläutert werden. Nur so können sie verstehen, wie diese sich verhalten und warum sich zum Beispiel Anforderungen in späteren Phasen der Entwicklung ändern. Versteht man dies, so ist es um so leichter, mit Anforderungsänderungen umzugehen.

Problematisch ist, dass leider moderne Softwareentwicklungsprozesse Methoden der Partizipation kaum beschreiben. Es wird zwar erkannt, dass die Beteiligung der Benutzer wichtig ist und es werden auch Phasen der Benutzerbeteiligung eingeplant, doch bei der Beschreibung des konkreten Vorgehens und bestimmter Methoden, bleiben sie weit hinter dem von Matthias Rauterberg beschriebenen System und auch hinter dem eigenen Vorgehen in anderen Teilbereichen zurück. Das muss sich ändern, damit der Einsatz von Methoden der Partizipation im Rahmen der Softwareentwicklung systematisiert und überprüfbar gemacht werden kann.

[18] Natürlich nur wenn das wirklich nicht der Fall ist

## Literatur

[Bec00] KENT BECK: *Extreme Programming explained: embrace change*. Addison-Wesley, 2000.

[Boe76] BARRY W. BOEHM: *Software Engineering*. IEEE Transactions on Computers, 25(12):1226–1241, 1976.

[Fis96] CHARLES FISHMAN: *They Write the Right Stuff*. Fast Company: `http://www.fastcompany.com/online/06/writestuff.html`, 1996.

[Fow02] MARTIN FOWLER: *The New Methodology*. `http://www.martinfowler.com/articles/newMethodology.htm`, 2002.

[HS93] ERIKA HORN und WOLFGANG SCHUBERT: *Objektorientierte Software-Konstruktion: Grundlagen, Modelle, Methoden, Beispiele*. Carl Hanser Verlag, 1993.

[JM02] MATTHIAS JARKE und HEINRICH C. MAYR: *Mediengestütztes Anforderungsmanagement*. Informatik Spektrum, 25(6):452–464, 2002.

[KPM97] *What Went Wrong? Unsuccessful Information Technology Projects*. KPMG Canada, 1997.

[Kru98] PHILLIPPE KRUCHTEN: *The Rational Unified Process*. Addison-Wesley, 1998.

[Mar02] ROBERT C. MARTIN: *RUP vs. XP*. `http://www.objectmentor.com/resources/articles/RUPvsXP.pdf`, 2002.

[OAS96] *The performance of Information Technology and the role of human and organisational factors*. OASIG, 1996.

[Pol01] GARY POLLICE: *Using the Rational Unified Process for Small Projects: Expanding Upon eXtreme Programming*. Rational Software: `http://www.rational.com/media/products/rup/tp183.pdf`, 2001.

[Rau95] MATTHIAS RAUTERBERG: *Partizipative Konzepte, Methoden und Techniken zur Optimierung der Softwareentwicklung*, 1995.

[Sta95] *The Chaos Report(1994)*. Standish Group, 1995.