Wiederverwendung ohne Mythos:

Empirisch fundierte Leitlinien

für die Entwicklung wiederverwendbarer Software

Rupert Stützle

Institut für Informatik
der Technischen Universität München

Wiederverwendung ohne Mythos: Empirisch fundierte Leitlinien für die Entwicklung wiederverwendbarer Software

Rupert Stützle

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Bernd Radig

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Manfred Broy
2. Hon.-Prof. Dr. Ernst Denert

Die Dissertation wurde am 23.04.2002 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 11.10.2002 angenommen.

Kurzfassung

Wiederverwendung gilt allgemein als geeignetes Mittel, die Produktivität der Softwareentwicklung zu steigern. In den vergangenen Jahren entstand dazu eine umfangreiche Literatur, die zahlreiche Ansätze zur praktischen Umsetzung enthält. Dennoch bleiben Ausmaß und Vorteile der Wiederverwendung in der industriellen Praxis weit hinter den an sie geknüpften Erwartungen zurück. Es liegt nahe, die Gründe dafür zu erforschen, um – soweit möglich – Abhilfe zu schaffen.

Gegenstand dieser Dissertation sind empirisch fundierte Leitlinien für die Entwicklung wiederverwendbarer Software, deren Anwendung dem Erfolg in technischer und ökonomischer Hinsicht dient. Der Schwerpunkt liegt dabei auf betrieblichen Informationssystemen.

Grundlage der Leitlinien sind die Ergebnisse empirischer und auf Simulationen basierender Untersuchungen. Zunächst wird der Status quo der Wiederverwendung anhand in Software-Unternehmen erhobener empirischer Daten festgestellt. In Fallstudien werden dann die softwaretechnischen Aspekte mehrerer Projekte zur Entwicklung wiederverwendbarer Software analysiert; besonderes Augenmerk liegt dabei auf der Systemarchitektur. Daraus leiten sich softwaretechnische Erfolgsfaktoren ab. Schließlich werden Rentabilität und Investitionsrisiko solcher Projekte mit Hilfe eines neuentwickelten, auf der Kapitelwertmethode basierenden ökonomischen Modells – des ReValue-Modells – untersucht, das die Nutzung wiederverwendbarer Software mit Hilfe der Monte-Carlo-Methode simuliert, und Simulationsergebnisse für verschiedene exemplarische Szenarien präsentiert.

Die Ergebnisse der verschiedenen Untersuchungen werden zunächst dazu herangezogen, Hypothesen zu weitverbreiteten Annahmen zu testen und teilweise zu falsifizieren. Darüber hinaus werden sie im Zusammenhang interpretiert und systematisch softwaretechnische und ökonomische Leitlinien für die Entwicklung wiederverwendbarer Software abgeleitet. Neben den empirisch erhobenen Rahmenbedingungen dient als Basis die vorgestellte Verfeinerung des Wiederverwendbarkeitsbegriffs. Die Leitlinien gliedern sich in strategische Leitlinien, die auf einer systematischen Nutzenanalyse basieren, Leitlinien für die Bewertung von Projektvorhaben nach softwaretechnischen und ökonomischen Kriterien und softwaretechnische Leitlinien. Letztere konzentrieren sich auf die Realisierung der ökonomisch und technisch sinnvollen Variabilität und eine Methode der Zerlegung eines Systems in Komponenten.

Danksagung

Professor Dr. Manfred Broy danke ich für die großzügige Betreuung und die äußerst konstruktive inhaltliche Begleitung dieser Arbeit. Professor Dr. Ernst Denert gebührt ebensolcher Dank; darüber hinaus bin ich ihm dafür verpflichtet, daß er mich zur rechten Zeit mit Professor Broy in Verbindung gebracht und mir die Arbeit bei sd&m Research ermöglicht hat.

sd&m Research bot mir ein optimales Arbeitsumfeld, wofür ich zuerst Professor Dr. Johannes Siedersleben danke. Er war ein wertvoller, stets ansprechbarer Ratgeber, und ich habe viel von ihm gelernt. Insgesamt habe ich mich in der kollegialen Atmosphäre bei sd&m Research äußerst wohl gefühlt. Dagmar Mazuré hat durch ihre ebenso fürsorgliche wie zupackende Art wesentlich dazu beigetragen. Allen Kollegen danke ich für ihre Hilfs- und Diskussionsbereitschaft und die Anteilnahme an dieser Arbeit; hervorheben möchte ich dabei Helmut Duschinger und Gerd Beneken (wie ich Männer der ersten Stunde bei sd&m Research), mit denen ich manches Problem bei einer Tasse Kaffee gelöst habe, meinen Laufkollegen Tilman Seifert und Michael Wufka, dem keine Ungenauigkeit verborgen blieb.

Am Lehrstuhl von Professor Broy wurde ich herzlich aufgenommen, wofür ich allen dort danke. Auch als Gast habe ich den dort herrschenden Teamgeist deutlich gespürt. Dr. Christian Salzmann und Alexander Pretschner haben besonders dafür gesorgt, daß ich von Beginn an in das wissenschaftliche und insbesondere in das reiche soziale Leben des Lehrstuhls integriert war. Christian Salzmann bin ich für seine unermüdliche Hilfsbereitschaft und die kompetenten Ratschläge besonders verpflichtet.

McKinsey & Company hat mir in jeder Hinsicht den für diese Arbeit notwendigen Spielraum gegeben. Speziellen Dank schulde ich Detlev J. Hoch und Dr. Detlev Ruland, die mir geholfen haben, als es am notwendigsten war. Ebenfalls unterstützt haben mich – jeweils auf ihre Weise – Claudia Funke, Dr. Hiltrud Ludwig und Dr. Stephan Schönwälder. Wolfgang Limbeck schließlich danke ich für seine fachmännische Unterstützung, die weit über das übliche Maß hinausging.

Dankbar bin ich auch den Verwandten und Freunden, die mir, besonders in kritischen Zeiten, zur Seite standen. Hervorheben möchte ich meinen Onkel Prof. Dr. Werner Stützle, der mir mit seinem Rat in fachlicher wie in persönlicher Hinsicht geholfen hat, sowie Christoph von Ehrenstein, Dr. Jochen Hammerschmidt und Dr. Anselm Schubert für ihre freundschaftliche Unterstützung.

Andrea Helfrich danke ich für ihr liebevolles Verständnis, ihre tatkräftige Hilfe, wenn Not am Mann war, und dafür, daß sie mir immer wieder den Rücken gestärkt hat.

Meine Eltern und meine Schwester Cornelia haben mich von Beginn an kontinuierlich in allen Belangen unterstützt und waren mir dadurch – besonders in schwierigen Phasen – ein wichtiger Rückhalt. Dafür danke ich ihnen; ohne sie wäre diese Arbeit nicht entstanden.

Gewidmet ist diese Arbeit meinen Eltern.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Ziel dieses Kapitels ist, einen Überblick über Inhalt und Aufbau der Arbeit sowie die eigenständigen wissenschaftlichen Beiträge zu geben.

Zunächst führen wir in das Thema ein und stellen dar, wodurch die Arbeit motiviert ist. Dann formulieren wir die Aufgabenstellung, definieren den Fokus und geben die Ziele der Arbeit an. Wir ordnen sie daraufhin in den wissenschaftlichen Kontext ein und stellen den eigenen Beitrag dar. Schließlich geben wir einen Überblick über den Aufbau der Arbeit und stellen Pfade durch die Arbeit vor, die sich an Interessenlage und Vorwissen des Lesers orientieren.

Inhalt: **Seite**

1.1 Wiederverwendung – ein zentrales Thema des Software Engineering

Software Engineering ist von Beginn an eng mit der Idee der Wiederverwendung verwoben. Die erste Software-Engineering-Konferenz, die NATO Working Conference on Software Engineering, fand 1968 statt. Im Tagungsband wurde ein Beitrag von M.D. McIlroy mit dem Titel *Mass produced software components* [McI68] veröffentlicht, der die industrielle Massenproduktion wiederverwendbarer Softwarekomponenten zur Lösung der soeben proklamierten Softwarekrise vorschlug. Dies trug wesentlich dazu bei, daß Wiederverwendung zu einem vieldiskutierten Thema wurde. Das ist sie bis heute geblieben: Die ihr gewidmete Ausgabe von IEEE Software im September 1994 markiert den Beginn eines bis heute andauernden Arbeitsschubs, in dessen Folge viele Veröffentlichungen zu verschiedenen Aspekten der Wiederverwendung entstanden, darunter eine Reihe von Büchern [Kar95, Tra95, JGJ97, McC97, Rei97, Cou97, Bas97, Lim98].

Wiederverwendung ist ein Querschnittsthema, das alle wesentlichen Bereiche des Software Engineering berührt. In Anlehnung an die Definition in [Den91] sind dies:

- die technische Gestaltung, von der Spezifikation über die Konstruktion bis zur Programmierung wiederverwendbarer Software; außerdem die Nutzung derselben bei den genannten Schritten der Softwareentwicklung
- die Planung und Abwicklung der Projekte, in denen wiederverwendbare Software entwickelt oder genutzt wird.

In vielen dieser Bereiche wirft die Wiederverwendung spezifische Probleme auf, die Gegenstand wissenschaftlicher Arbeit sind. Die besondere Bedeutung der Wiederverwendung für das Gebiet des Software Engineering findet ihren Ausdruck auch darin, daß die Entwicklung und Einführung neuer Techniken regelmäßig mit der zu erzielenden Wiederverwendbarkeit begründet wird.[1]

Der Grund dafür wie auch für die anhaltende Popularität der Wiederverwendung allgemein liegt in den hohen Erwartungen an den durch sie zu erzielenden Nutzen. Es werden dramatische Verbesserungen bei Geschwindigkeit und Kosten der Softwareentwicklung und der Qualität des Ergebnisses erwartet (vgl. z.B. [JGJ97], 5): Für die Erhöhung der Produktivität beispielsweise werden ganzzahlige Faktoren angesetzt, die von einer Verdopplung ([Kar95], 11) bis zu einer Steigerung um Faktor zehn ([Gol95], 204; [Tra95], 132f) reichen.

[1] Beispielsweise wurde bei einer Konferenz zu Objektorientierung (Boston Object World) im Jahr 1995 Wiederverwendung neben kürzerer Entwicklungsdauer als Hauptgrund für die Einführung von OO-Techniken genannt (vgl. [McC97], 4); Wiederverwendung wird auch als treibender Faktor von Component-based Software Engineering angeführt (vgl. [Szy99], 3); Wiederverwendbarkeit wird insbesondere im Zusammenhang mit der neuen Enterprise JavaBeans-Technik als zu erreichendes Ziel apostrophiert ([Rom99], 3)

Sowohl die enge Verzahnung der Wiederverwendung mit dem gesamten Gebiet des Software Engineering als auch die hohen Erwartungen haben die vorliegende Arbeit maßgeblich beeinflußt.

1.2 Motivation der Arbeit

Wiederverwendung ist ein faszinierendes, schillerndes Thema; man könnte sagen: ein Mythos des Software Engineering (vgl. [Gur00][2]). Die Erwartungen an Wiederverwendung waren zeitweise so hoch, daß Wiederverwendung sogar zu der Wunderwaffe (engl.: silver bullet) erklärt wurde [Cox90], die es – wie von Brooks in [Bro87] überzeugend dargelegt – gar nicht geben kann. Doch auch aktuellere Veröffentlichungen sind wenig zurückhaltend.[3] Dem steht die Wirklichkeit in der industriellen Praxis diametral gegenüber: wenig ist umgesetzt, umfangreichen Bemühungen zum Trotz (vgl. [GAO95], 17; [Tra95], 5; [Som96], 396; [DvK95, Gur00, ZBP01]).[4]

Viele Vermutungen wurden bereits zur Erklärung dieses offensichtlichen Widerspruchs angestellt. Ob das Problem in der mangelnden Umsetzung der Forschungsergebnisse liegt, ist fraglich; eine mögliche Erklärung liegt im mangelnden wissenschaftlichen Fortschritt (vgl. [Tra95], 85)[5]. Bisher wurde die große Zahl an bestehenden Erklärungshypothesen – vom Not-invented-here-Syndrom [GrW95] über strategische Fehler [CCo94] und grundlegende Unterschiede zwischen Software Engineering und klassischen Ingenieurdisziplinen [ZBP01] bis hin zu mangelnder Erforschung der Systemsicht [DvK95] – jeden-

---

[2] Van Gurp bezeichnet Wiederverwendung als den heiligen Gral des Software Engineering: "Reuse of software assets has been, and continues to be the holy grail of software engineering". [Gur00]

[3] Sodhi und Sodhi beispielsweise erwarten eine signifikante Rendite durch Erhöhung von Qualität, Produktivität und Wartbarkeit im gesamten Entwicklungszyklus: "The importance of software reuse lies in its benefits of providing quality and reliable software in a relatively short time. These factors reduce the costs of software development and maintenance. The computer industry has demonstrated that software reuse generates a significant return on investment by reducing cost, time, and effort while increasing the quality, productivity, and maintainability of software systems throughout the software life cycle.“ ([SoS99], 5)

[4] Zur Illustration zitieren wir einige beispielhafte Aussagen wörtlich: Garlan et al. halten die systematische Konstruktion großer Systeme für ein trügerisches Ziel: “There is considerable research and development in reuse. [...] Yet the systematic construction of large-scale software applications from existing parts remains an elusive goal” ([GAO95], 17); Dusink und Katwijk stellen fest, daß Wiederverwendung trotz ständiger gegenteiliger Versprechungen die Erwartungen nicht erfüllt: "As nearly every paper starts with observing that reuse is still not mature while the problem will be solved in the paper in question, a further observation is that reuse still does not live up to its promise." [DvK95]; in einer aktuellen Podiumsdiskussion stellen Zand et al. u.a. fest, daß Wiederverwendung in der Softwareindustrie nicht weit verbreitet ist: "Contrary to belief, [practice in traditional engineering and software engineering] are different, and that difference is part of the reason why reuse is not widely practiced in the software industry." [ZBP01]

[5] Tracz formuliert die Meinung, daß sehr lange Stillstand geherrscht habe, drastisch: "[...] instead of 23 years of progress in advancing the state of the art in software reuse, researchers had repeated 1 year of progress 23 times" ([Tra95], 85)

falls keiner befriedigenden Überprüfung auf empirischer Basis unterzogen. Die bestehenden empirischen Untersuchungen stellen keine ausreichenden quantitativen Daten zur Verfügung und untersuchen wichtige Themen, beispielsweise Softwarearchitektur, gar nicht oder nicht ausreichend genau.

Warum der erwartete Erfolg der Wiederverwendung bisher ausbleibt, ist bis heute ungeklärt. Sollte er grundsätzlich erreichbar sein, so stellt sich die Frage nach der geeigneten Vorgehensweise. Wir halten eine Untersuchung dieser Fragen auf einer empirischen Basis für geboten.

1.3 Aufgabenstellung, Fokus und Ziel

Aus den vorangegangenen Überlegungen ergibt sich die Aufgabenstellung für diese Arbeit. Sie besteht darin,

- eine empirische Untersuchung des Status quo der Wiederverwendung durchzuführen, wobei die Daten ausreichend detailliert und, wo nötig, in quantitativer Form erhoben werden
- Leitlinien für die Entwicklung wiederverwendbarer Software abzuleiten, die den Erfolg in technischer und ökonomischer Hinsicht sicherstellen

Die Fokussierung der Leitlinien auf die Entwicklung, d.h. die Herstellung, wiederverwendbarer Software (unter Aussparung der Nutzung) hat zwei Gründe. Zum einen ist das Vorhandensein wiederverwendbarer Software eine notwendige Voraussetzung für erfolgreiche Wiederverwendung; außerdem wird aktuell ein Mangel vermutet ([GAO95]; [Tra95], 137; [JGJ97], 8). Zum anderen macht der Umfang des Arbeitsgebiets der Wiederverwendung eine Eingrenzung unumgänglich. Darüber hinaus beschränken wir die Betrachtung auf Wiederverwendung innerhalb eines Unternehmens; dies entspricht dem intuitiven und oft impliziten Verständnis des Begriffs.

Angesichts der vielen verschiedenen Arten von Software konzentrieren wir uns darüber hinaus auf betriebliche Informationssysteme, d.h. Softwaresysteme, die die Abläufe in allen Bereichen eines Unternehmens unterstützen und steuern ([Den91], 15). Sie sind wegen ihrer Durchdringung der Unternehmen in der heutigen Wirtschaft von großer Bedeutung und repräsentieren außerdem einen großen Softwaremarkt. Deshalb sind sie von besonderem Interesse.

Im Mittelpunkt des Interesses steht für uns die Wiederverwendung von Software in Form von Komponenten. Dies entspricht dem Stand der Forschung; allerdings legen wir nicht eine allgemeingültige Definition der Komponente zugrunde, sondern differenzieren nach der Granularität.

Bei der softwaretechnischen Betrachtung konzentrieren wir uns auf Komponenten, die systematisch zur Wiederverwendung aufbereitet werden und die in angepaßter Form genutzt werden. Dabei ist die sogenannte White-box-Wiederverwendung im Fokus, bei der eine Modifikation des Quellcodes möglich ist.

Einen Überblick über den Fokus der Arbeit gibt Tabelle 1-1.

Aspekt	Fokus
Perspektive der Wiederverwendung	Entwicklung (Herstellung) wiederverwendbarer Software
Einsatzgebiet	Wiederverwendung (Nutzung) innerhalb des Unternehmens
Domäne	Betriebliche Informationssysteme
Was wird wiederverwendet?	Software-Komponenten
Angestrebte Wiederverwendbarkeit	Systematische Wiederverwendbarkeit
Art der Nutzung	In angepaßter Form (mit Modifikation des Quellcodes: White-box-Wiederverwendung)

Tabelle 1-1: Fokus der Arbeit

Ziel der Arbeit ist, auf empirischer Basis zu einer differenzierten und abgewogenen Sicht der Wiederverwendung zu kommen und davon Handlungsanweisungen abzuleiten, die den Rahmenbedingungen in der industriellen Praxis Rechnung tragen. Im einzelnen sind dazu zunächst Daten in detaillierter Form zu erheben und zu interpretieren, um das Verständnis des Status quo der Wiederverwendung sicherzustellen. Daraus sollen dann Leitlinien für die Entwicklung wiederverwendbarer Software entwickelt werden, die Softwaretechnik und Ökonomie umfassen.

1.4 Wissenschaftlicher Kontext und eigener Beitrag

Die Literatur zur Entwicklung wiederverwendbarer Software ist umfangreich und vielfältig. Neben Veröffentlichungen, die spezifisch einzelne Aspekte – vor allem aus den Bereichen technische Architektur und Ökonomie – untersuchen, gibt es mittlerweile eine Reihe von Büchern, die sich umfassend damit auseinandersetzen. Die spezifischen Arbeiten entstammen verschiedenen Bereichen der Forschung, in denen Wiederverwendung eine wichtige Rolle spielt. Zum einen behandeln sie Themen aus dem Umfeld der technischen Softwarearchitektur: Entwurfsmuster, Komponenten und Frameworks, Variabilität und Produktlinien sowie Trennung der Zuständigkeiten[6] und Aspekt-orientierte Programmierung. Zum anderen beschäftigen sie sich mit ökonomischen und strategischen Aspekten der Wiederverwendung. Insgesamt ergibt sich Forschungsbedarf auf verschiedenen Gebieten. Im Folgenden fassen wir ihn kurz zusammen und arbeiten unseren wissenschaftlichen Beitrag heraus.

[6] Englisch: separation of concerns

- *Domäne.* Die spezifischen Probleme der Entwicklung wiederverwendbarer Software im Kontext betrieblicher Informationssysteme sind bisher unzureichend untersucht, weil sich die meisten Arbeiten auf andere Domänen konzentrieren. Indem sie auf betriebliche Informationssysteme fokussiert ist, trägt diese Arbeit dazu bei, diese Lücke zu schließen, und leistet dadurch einen wissenschaftlichen Beitrag.
- *Empirisches Fundament.* Wiederverwendung ist empirisch unzureichend untersucht. Zum einen sind vorhandene Daten größtenteils qualitativer Art, was besonders bei der Frage nach dem ökonomischen Nutzen ein großes Manko ist. Zum anderen sind in vielen Bereichen zu wenige Daten vorhanden, insbesondere hinsichtlich der aktuellen Rahmenbedingungen und der Architektur wiederverwendbarer Software. Wir leisten einen Beitrag, indem wir detaillierte empirische Untersuchungen anstellen. Zum einen erheben wir quantitative Daten zu aktuellem Vorgehen, Einschätzung der Wiederverwendung, Handlungsbedarf und ökonomischen Aspekten. Zum anderen erforschen wir qualitativ die softwaretechnischen Aspekte der Entwicklung wiederverwendbarer Software.
- *Softwareökonomie.* Die Methoden zur ökonomischen Bewertung der Entwicklung wiederverwendbarer Software berücksichtigen die Investitionsunsicherheit entweder nicht oder haben den Mangel, daß sie das für Projekte zur Softwareentwicklung nicht geeignete Capital Asset Pricing Model verwenden. Wir stellen dem eine neue Bewertungsmethode auf Basis des internen Zinssatzes entgegen, die die Unsicherheit auf praktikable Weise berücksichtigt. Zudem präsentieren wir ein ökonomisches Modell der Wiederverwendung, das – im Gegensatz zu den verfügbaren – alle wichtigen Größen modelliert. Auf Basis dieses Modells entwickeln wir einen Algorithmus zur Simulation der Nutzung wiederverwendbarer Software mit der Monte-Carlo-Methode. Wir stellen Simulationsergebnisse für eine Reihe exemplarischer Szenarien vor und tragen so dazu bei, ausreichende Daten zum ökonomischen Potential der Wiederverwendung bereitzustellen.
- *Strategische Relevanz der Wiederverwendung.* Die strategische Relevanz der Wiederverwendung ist unzureichend untersucht. Im Sinne einer Beseitigung dieses Defizits untersuchen wir sie mit Hilfe einer systematischen Nutzenanalyse und bestimmen – getrennt nach Produkt- und Projektgeschäft – den Beitrag des Nutzens zur Erreichung verschiedener strategischer Ziele.
- *Bewertung von Projektvorhaben.* In der Literatur existiert kein Verfahren zur Bewertung von Projektvorhaben, das softwaretechnische und ökonomische Kriterien verbindet. Wir stellen ein solches Verfahren bereit.
- *Softwaretechnik.* Es herrscht ein Mangel an Verfahren zur Zerlegung in Komponenten auf einer angemessenen Granularitätsebene, die eine Trennung der Zuständigkeiten sicherstellen. Wir helfen dem ab, indem wir, ausgehend von den Ergebnissen der softwaretechnischen Untersuchungen, ein Verfahren zur Zerlegung in Komponenten auf einer mittleren Granularitätsebene vorstellen. Darüber hinaus besteht Bedarf an einem durchgängigen Verfahren zur Bestimmung und Realisierung der technisch und ökonomisch

sinnvollen Variabilität. Das von uns vorgestellte Konzept der kontrollierten Variabilität erfüllt diese Anforderungen.

1.5 Aufbau der Arbeit

In *Kapitel 2* definieren wir die für diese Arbeit wesentlichen Begriffe und fassen die Grundlagen zusammen, auf denen sie aufbaut. Das Ziel ist, ein klares, eindeutiges Begriffssystem als Fundament der Arbeit bereitzustellen und den Fokus der Arbeit in detaillierterer Form darzustellen. Zunächst bestimmen wir den Begriff der Wiederverwendung und beleuchten verschiedene seiner Aspekte, insbesondere die Perspektiven der Wiederverwendung und die Merkmale wiederverwendbarer Software. Dann erläutern wir die Motivation für Wiederverwendung und deren historische Wurzeln. Anschließend werden für diese Arbeit essentielle Grundlagen zusammenfassend dargestellt. Dazu gehört das Gebiet der Softwaretechnik mit dem Modell des Entwicklungsprozesses und den Facetten der Softwarearchitektur. Auf ihr liegt besonderes Gewicht; die Betrachtung umfaßt auch Komponenten und Schnittstellen sowie Software-Kategorien. Darauf fassen wir die Grundlagen der Softwareökonomie zusammen. Schließlich gehen wir auf wichtige Merkmale der im Zentrum der Arbeit stehenden betrieblichen Informationssysteme ein und fassen den Fokus der Arbeit in einem Überblick zusammen.

Kapitel 3 faßt den Stand der Forschung zu den verschiedenen Aspekten der Entwicklung wiederverwendbarer Software zusammen. Ziel ist, den Forschungsbedarf zu bestimmen und die eigenständigen Beiträge der Arbeit herauszuarbeiten. Zuerst stellen wir die Literatur zur Entwicklung wiederverwendbarer Software vor und fassen ausgewählte Beiträge zusammen. Dabei gehen wir nach folgenden Bereichen gegliedert vor: umfassende und spezifische Beiträge, Grundlagen und empirische Untersuchungen sowie neuere Entwicklungen und Trends. Schließlich stellen wir den Forschungsbedarf zusammengefaßt dar, ordnen die Arbeit in das wissenschaftliche Umfeld ein, bestimmen ihren Beitrag und fassen sich aus der Literatur ergebende Forschungsfragen zu empirisch überprüfbaren Hypothesen zusammen.

Ziel von *Kapitel 4* ist, den Status quo der Wiederverwendung in der industriellen Praxis empirisch zu erheben und Schlüsse für das Vorgehen zu ziehen. Das Kapitel stellt die Ergebnisse einer Befragung von 55 Entwicklern zum Status quo der Wiederverwendung bei einem internationalen Softwareproduktunternehmen vor und analysiert sie. Anhand der Ergebnisse überprüfen wir einen großen Teil der in Kapitel 3.5.3 formulierten Hypothesen. Wir interpretieren die Ergebnisse im Zusammenhang und leiten allgemeine Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software ab.

Kapitel 5 hat das Ziel, empirisch fundierte softwaretechnische Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software zu bestimmen. Das Kapitel stellt die Ergebnisse von drei Fallstudien zur Softwaretechnik in Individualentwicklungsprojekten der sd&m AG vor. Dabei wird sowohl die Herstellung als auch die Nutzung untersucht. Ein besonderer Schwerpunkt liegt auf der Ar-

chitektur. Die Ergebnisse werden im Vergleich analysiert und Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software abgeleitet.

Ziel von *Kapitel 6* ist, eine Methode zu entwickeln, mit der das ökonomische Potential der Entwicklung wiederverwendbarer Software in Abhängigkeit von den jeweiligen Rahmenbedingungen bestimmt werden kann. Dabei sollen alle wesentlichen Faktoren berücksichtigt werden, um zu möglichst realistischen Aussagen zu gelangen. Hierzu stellt das Kapitel zunächst das ReValue-Modell[7] für die ökonomische Bewertung von Projekten zur Entwicklung wiederverwendbarer Software vor. Das Modell hat drei Bestandteile: Es stellt eine Bewertungsmethode bereit, die auf praktikable Weise die Investitionsunsicherheit berücksichtigt. Zusätzlich werden Herstellung und Nutzung in umfassender Weise ökonomisch modelliert. Schließlich umfaßt es einen Algorithmus, der die Nutzung mit Hilfe der Monte-Carlo-Methode simuliert. Auf dieser Grundlage präsentieren wir Simulationsergebnisse für verschiedene exemplarische Szenarien, analysieren sie und ziehen Schlußfolgerungen für das ökonomische Potential in Abhängigkeit von den Rahmenbedingungen.

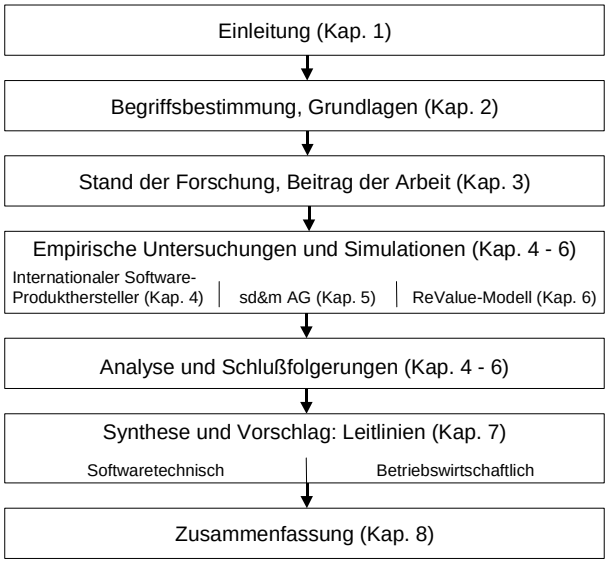

Abbildung 1-1: Aufbau der Arbeit

Kapitel 7 verfolgt das Ziel, als Synthese der bisherigen Ergebnisse der Arbeit Leitlinien für die Entwicklung wiederverwendbarer Software zu formulieren. Die Anwendung dieser Leitlinien soll den Erfolg der Entwicklung in softwaretechnischer und wirtschaftlicher Hinsicht bewirken. Zunächst stellen wir die Rahmenbedingungen und den Status quo der Wiederverwendung dar, insbesondere in Form der empirisch überprüften Hypothesen. Dann präsentieren wir eine Verfeinerung des Wiederverwendbarkeitsbegriffs, die den Zusammenhang der Softwareentwicklung berücksichtigt und sich an der notwendigen Investition orientiert. Basierend auf einer systematischen Nutzenanalyse stellen wir anschließend strategische Leitlinien vor, die den Beitrag wiederverwendbarer Software zur Erreichung strategischer Ziele umfassen. Dann geben wir Leitlinien für die Bewertung von Projektvorhaben nach softwaretechnischen und

[7] ReValue ist ein Akronym für 'Reusable Software Valuation Model'

ökonomischen Kriterien an. Schließlich formulieren wir softwaretechnische Leitlinien. Sie betreffen die softwaretechnisch und ökonomisch sinnvolle Realisierung von Variabilität und die Zerlegung des Systems in Komponenten.

Schließlich faßt *Kapitel 8* den Inhalt der Arbeit in seinen Grundzügen zusammen und gibt einen Ausblick auf sich ergebende zukünftige Forschungsthemen. Einen Überblick über den logischen Aufbau der Arbeit gibt Abbildung 1-1. In ihr werden Kapitel 4 bis Kapitel 6 dementsprechend parallel dargestellt.

1.6 Hinweise für den Leser: Pfade durch die Arbeit

Nicht jeder Leser muß alles lesen. Wir geben daher an, welche Teile der Arbeit je nach Vorwissen und Intention gegebenenfalls übersprungen werden können.

In Kapitel 2 sollte jeder Leser die Begriffsbestimmung lesen (2.1). Die restlichen Teile stellen Grundlagen des Software Engineering, der Softwareökonomie und betrieblicher Informationssysteme dar, die bei entsprechendem Vorwissen fakultativ sind. Speziell sei auf den Abschnitt über die für diese Arbeit äußerst wichtigen Software-Kategorien verwiesen (2.3.6), den jeder lesen sollte, der mit ihnen noch nicht vertraut ist.

In Kapitel 3 sind die wesentlichen Punkte in Abschnitt 3.5 enthalten (Zusammenfassung des Forschungsbedarfs, Einordnung und Beitrag der Arbeit, empirisch zu überprüfende Hypothesen), der von allen Lesern gelesen werden sollte. Der Rest des Kapitels geht auf die Literatur zu den verschiedenen Gebieten im Detail ein und kann je nach Kenntnis dieser Literatur übersprungen werden.

In Kapitel 4 kann sich der Leser für einen beschleunigten Durchlauf auf die Abschnitte 4.3 und 4.4 konzentrieren, die die Ergebnisse zusammenfassen. Analoges gilt für Kapitel 5 mit den Abschnitten 5.3 und 5.4.

In den restlichen Kapiteln kann ohne eine Einbuße an Verständnis auf nichts verzichtet werden, so daß sie von allen Lesern vollständig gelesen werden sollten.

Kapitel 2

Begriffe und Grundlagen

In diesem Kapitel definieren wir die für diese Arbeit wesentlichen Begriffe und fassen die Grundlagen zusammen, auf denen sie aufbaut. Das Ziel ist, ein klares, eindeutiges Begriffssystem als Fundament der Arbeit bereitzustellen und den Fokus der Arbeit in detaillierterer Form darzustellen.

Zunächst bestimmen wir den Begriff der Wiederverwendung und beleuchten verschiedene seiner Aspekte, insbesondere die Perspektiven der Wiederverwendung und die Merkmale wiederverwendbarer Software. Dann erläutern wir die Motivation für Wiederverwendung und deren historische Wurzeln. Anschließend werden für diese Arbeit essentielle Grundlagen zusammenfassend dargestellt. Dazu gehört das Gebiet der Softwaretechnik mit dem Modell des Entwicklungsprozesses und den Facetten der Softwarearchitektur. Auf ihr liegt besonderes Gewicht; die Betrachtung umfaßt auch Komponenten und Schnittstellen sowie Software-Kategorien. Darauf fassen wir die Grundlagen der Softwareökonomie zusammen. Schließlich gehen wir auf wichtige Merkmale der im Zentrum der Arbeit stehenden betrieblichen Informationssysteme ein und fassen den Fokus der Arbeit in einem Überblick zusammen.

Inhalt: **Seite**

2.1 Software-Wiederverwendung: Begriffsbestimmung

Software-Wiederverwendung ist ein Begriff, der in der Literatur nicht einheitlich gebraucht wird und eher diffusen Charakter hat. Zu Beginn dieser Arbeit bestimmen wir ihn daher mit seinen verschiedenen Facetten, um auf sicherer Grundlage arbeiten zu können.

Bevor wir den eigentlichen Kern – die Wiederverwendung − behandeln, kommen wir nicht umhin, auf den Begriff *Software* einzugehen, der im Kontext von Software-Wiederverwendung in der Literatur nicht eindeutig bestimmt ist (vgl. [JGJ97, Kar95, Sam97, McC97, Tra95]). Wir definieren Software wie folgt (vgl. [Pre97], 10; [Bal99], 31; [Bal96], 23):

> *Der Begriff Software umfaßt (1) Programmcode, d.h. Instruktionen und Datenstrukturen, (2) Dokumente, die das Programm selbst und seine Nutzung beschreiben, (3) Testdaten und Testprogramme, die dazu dienen, das fehlerfreie Funktionieren des Programmcodes zu überprüfen.*

2.1.1 Was ist Software-Wiederverwendung?

Der Begriff der Wiederverwendung ist scheinbar intuitiv verständlich, weil er auch in der Alltagssprache verwendet wird. Im wissenschaftlichen Kontext muß er jedoch genau definiert werden.

Definitionen von Wiederverwendung basieren in der Regel darauf, daß Vorhandenes verwendet wird, um etwas Neues zu schaffen (vgl. z.B. [Gol95], 203). Grundlage für unsere Begriffsbestimmung ist daher die Definition von Wiederverwendung als Nutzung bereits vorhandener Software bei der Softwareentwicklung im Gegensatz zu reiner Neuentwicklung (vgl. [Krg92]; [Fra94]; [Rei97], 3).

Diese Arbeit konzentriert sich auf betriebliche Informationssysteme, also Softwaresysteme, die die Abläufe in allen Bereichen eines Unternehmens unterstützen und steuern ([Den91], 15; eine detaillierte Behandlung erfolgt in 2.5). Daher verfeinern und konkretisieren wir die Definition der Wiederverwendung in diesem Kontext. Dazu betrachten wir die Zusammensetzung eines zu erstellenden betrieblichen Informationssystems (für eine beispielhafte Darstellung siehe Abbildung 2-1[8]).

Es besteht einerseits aus intern entwickelter Software; dabei handelt es sich entweder um Software, die spezifisch für diesen Zweck neu entwickelt wurde, oder um Software, die bereits vorhanden war und – eventuell in modifizierter Form – eingesetzt wird. Andererseits geht von außen bezogene Software in Form von kommerziellen Produkten und Open Source-Software in das System

[8] Die Zusammensetzung nach Bestandteilen ist von System zu System unterschiedlich; in dieser Hinsicht ist die Abbildung beispielhaft, nicht repräsentativ.

ein (vgl. [WiB98]). Wir nehmen dabei die Sicht der Einheit ein, die das System entwickelt. Diese Einheit kann ein Unternehmen sein, das Softwareprodukte herstellt oder Softwareprojekte durchführt, oder die EDV-Abteilung eines Unternehmens, dessen Kerngeschäft nicht Softwareentwicklung ist, beispielsweise einer Bank.

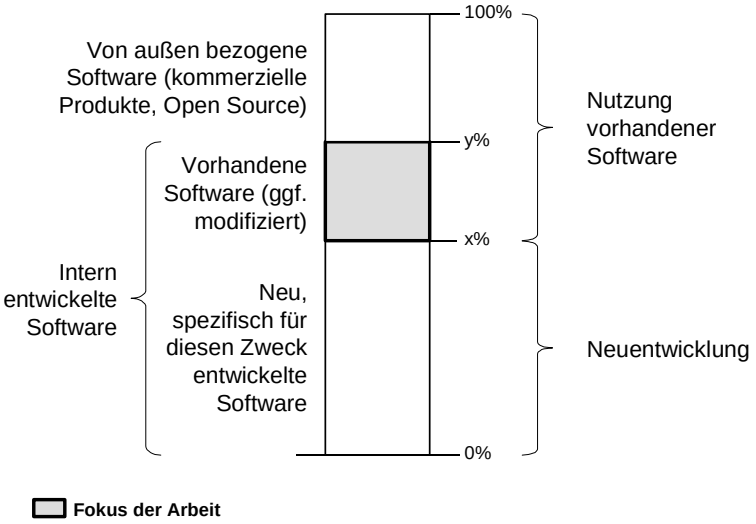

Abbildung 2-1: Zusammensetzung eines betrieblichen Informationssystems

Unsere Definition der Wiederverwendung (siehe Abbildung 2-2) geht von der Nutzung vorhandener Software aus. Zunächst unterscheiden wir nach der Herkunft der Software. Handelt es sich um intern entwickelte Software, so sprechen wir von *Wiederverwendung*. Wird extern entwickelte Software eingesetzt, so nennen wir dies *Verwendung* (vgl. [Den91]; [PoC93]).

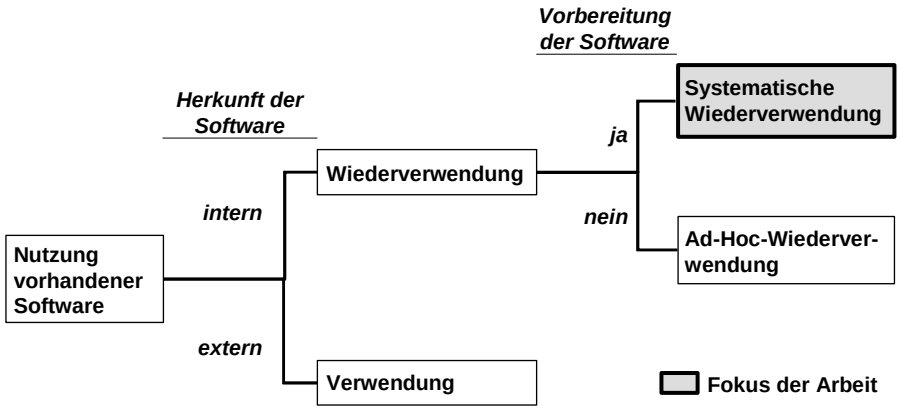

Abbildung 2-2: Definition von Wiederverwendung

Diese Abgrenzung entspricht dem Sprachgebrauch und dem intuitiven Verständnis, auch wenn sie in der Literatur nicht immer so klar vorgenommen wird. Sie ist auch sachlich sinnvoll, weil wesentliche Unterschiede bestehen, insbesondere in der Vermarktung und der Möglichkeit, Standards zu setzen; das wirkt sich wiederum auf die Rahmenbedingungen der Entwicklung aus.

Wir sprechen von systematischer Wiederverwendung, wenn Software genutzt wird, die explizit für die Wiederverwendung aufbereitet ist (vgl. [JGJ97], 16; [Lim98], 8). Wird hingegen Software genutzt, die nicht zur Wiederverwendung aufbereitet wurde, so nennen wir dies Ad-Hoc-Wiederverwendung[9] (vgl. [JGJ97], 16). Diese Art der Wiederverwendung tritt regelmäßig auf: viele Entwickler verwenden Software, die sie selbst in vorangegangenen Projekten entwickelt haben, wieder, wenn sie auf ähnliche Anforderungen stoßen (vgl. [Shm99]; [Kar95], 60; [CaE95], 2). Systematische Wiederverwendung ist weniger weit verbreitet [Gri99]: in vielen Entwicklungsprojekten findet man heute ausschließlich Ad-Hoc-Wiederverwendung ([Kar95], 5f).

Systematische Wiederverwendung ist immer noch ein sehr weitreichender Begriff, der im Sinne der Genauigkeit der Beschreibung weiter untergliedert werden sollte. Dazu sind jedoch zusätzliche Grundlagen notwendig, die wir in den folgenden Abschnitten legen, bevor wir in 2.1.6 einen differenzierten Begriff und ein Modell der Wiederverwendung vorstellen.

2.1.2 Merkmale wiederverwendbarer Software

Software ist wiederverwendbar, wenn sie in mehr als einem Kontext eingesetzt werden kann, d.h. in unterschiedlichen Entwicklungsprojekten oder im selben Projekt an verschiedenen Stellen (vgl. [Sam97], 258). Grundsätzlich definieren wir daher Wiederverwendbarkeit von Software als *Einsetzbarkeit in unterschiedlichen Kontexten* (vgl. [Kar95], 257; [Sam97], 108). Drei Merkmale sind dafür ausschlaggebend (vgl. [Kar95], 134ff; [Tra95], 137ff; [McC97], 279):

- Nutzbarkeit für Entwickler,
- Adaptierbarkeit und
- Portierbarkeit.

Dabei handelt es sich um diejenigen Qualitätsmerkmale von Software, die für deren Wiederverwendbarkeit spezifische Bedeutung haben. Wiederverwendbare Software muß diese Merkmale grundsätzlich in höherem Maß besitzen als nicht wiederverwendbare Software.

Darüber hinaus muß wiederverwendbare Software weitere Qualitätskriterien erfüllen, die sich jedoch nicht von den Kriterien unterscheiden, die nicht wiederverwendbare Software erfüllen muß, wenn sie an derselben Stelle eingesetzt wird. Diese Qualitätskriterien umfassen Spezifikationstreue der Funktionalität, Zuverlässigkeit, Benutzbarkeit für den Endbenutzer und Effizienz (zusammengesetzt aus Performance und Ressourcenverbrauch).

Die hier zugrunde gelegte Definition von Qualität orientiert sich an der ISO-Definition ([Bal98], 258ff; siehe auch [Kar95], 6), die wir um den Aspekt der Nutzbarkeit für Entwickler erweitern. Was die Frage der Messung der ver-

[9] auch als informelle Wiederverwendung bezeichnet (vgl. [Kar95], 5)

schiedenen Merkmale betrifft, gehen wir davon aus, daß auf bewährte Metriken zurückgegriffen wird[10]; wir beschäftigen uns damit nicht näher.

Nutzbarkeit für Entwickler

Die Nutzbarkeit für Entwickler ist insbesondere dann von Bedeutung, wenn ein anderer als der ursprüngliche Programmierer Änderungen und Anpassungen vornehmen soll, was bei Wiederverwendung regelmäßig der Fall ist. Sie setzt sich aus folgenden Merkmalen zusammen:

- *Verständlichkeit*: Damit ein Entwickler Software wiederverwenden kann, muß er verstehen können, welche Spezifikation sie erfüllt und wie er sie einsetzen kann.
- *Änderungseffizienz*: Sowohl bei der Codierung als auch beim Testen fällt unterschiedlich hoher Aufwand an, je nachdem, wie gut die Software auf Änderungen vorbereitet ist und welche Hilfsmittel zur Verfügung gestellt werden. Je geringer der für das Ändern notwendige Aufwand, desto höher ist die Änderungseffizienz.
- *Verfügbarkeit*: Je geringer der Aufwand dafür ist, von der Existenz wiederverwendbarer Software zu erfahren und sie dann auch in der jeweils aktuellen Version zu erhalten, desto höher ist ihre Verfügbarkeit.[11]

Wir gehen nun darauf ein, welche Maßnahmen für die einzelnen Aspekte der Nutzbarkeit für Entwickler notwendig sind. Um die *Verständlichkeit* sicherzustellen, muß die Software ausreichend dokumentiert sein. Der Qualität der Dokumentation kommt für die Wiederverwendbarkeit herausgehobene Bedeutung zu (vgl. [Bro95], 224; [Kar95], 84; [MCD00], 20). Wir unterscheiden zwei Arten der Dokumentation: die Dokumentation im Code und die Dokumentation in einem separaten Dokument. Auch die Unterstützung durch eine Support-Funktion – telefonisch oder online – und Code Reviews können zur Verständlichkeit beitragen. Ein wesentliches Mittel dafür, die *Änderungseffizienz* wiederverwendbarer Software sicherzustellen, ist die Bereitstellung von Hilfsmitteln für Regressionstests, d.h. Testdaten und Testprogrammen, die es ermöglichen, die korrekte Funktionsweise bei der Nutzung innerhalb eines bestimmten Rahmens sicherzustellen. Der Aufwand bei der Codierung kann durch Bereitstellung von Codegeneratoren verringert werden. Die *Verfügbarkeit* wiederverwendbarer Software schließlich kann auf unterschiedliche Arten hergestellt werden. Eine oft erwähnte Möglichkeit ist die Bereitstellung einer Wiederverwendungs-Bibliothek, auf die z.B. online zugegriffen werden kann (vgl. z.B. [Kar95], 79ff). Die Probleme der Klassifikation und der Suche sind dem Forschungsgebiet Information Retrieval zuzuordnen und können im Wesentlichen

---

[10] Verschiedene Metriken zur Messung von Software-Qualität existieren, u.a. die Indikatoren des FURPS-Modells (vgl. [Bal98], 262) und Indikatoren für einzelne Merkmale wie Anzahl der Defekte pro Programmzeile für die Zuverlässigkeit oder Mean time to change (MTTC) für die Adaptierbarkeit ([Pre97], 95).

[11] Dies ist im strengen Sinne kein Merkmal der Software selbst, sondern eine Eigenschaft der Organisation, in der sie entwickelt wird. Wir führen es der Vollständigkeit halber dennoch hier auf.

als gelöst betrachtet werden (vgl. [Gri99, ZBB99]). Sie werden daher in dieser Arbeit nicht behandelt.

Adaptierbarkeit

Adaptierbarkeit ist die Eigenschaft von Software, an unterschiedliche funktionale Anforderungen anpaßbar zu sein. Dies bezieht sich sowohl auf Anforderungen an die fachliche Funktionalität der Komponente selbst als auch auf ihre Fähigkeit, mit unterschiedlichen Systemen fachlich zusammenzuarbeiten, d.h. eine fachliche Systemintegration zu ermöglichen. Adaptierbarkeit ist deshalb ein wichtiges Merkmal, weil wiederverwendbare Software in unterschiedlichen Kontexten einsetzbar sein soll. Es ist wenig wahrscheinlich, daß die funktionalen Anforderungen dabei jeweils genau die selben sind. Deshalb sollte sie Mechanismen dafür zur Verfügung stellen, sie in einem vorher festgelegten Rahmen an die Anforderungen anzupassen, die bei einer konkreten Nutzung an sie gerichtet werden. Je weiter dieser Rahmen gespannt wird, desto größer ist die geforderte Adaptierbarkeit der Software und desto schwieriger ist es in der Regel auch, sie zu erreichen. Voraussetzung für Adaptierbarkeit von Software ist im Fall von A-Posteriori-Wiederverwendung, daß sie aus ihrem ursprünglichen Kontext herausgelöst werden kann.

Portierbarkeit

Die Portierbarkeit von Software beschreibt ihre Fähigkeit, in unterschiedlichen Umgebungen technisch einsetzbar zu sein, d.h. eine technische Systemintegration zu ermöglichen. Das bezieht sich zunächst auf die technische Systemumgebung der Komponente, aber auch auf die technische Realisierung der Kommunikation mit Nachbarsystemen, mit denen sie fachlich zusammenarbeitet (vgl. [Bro95], 6). Man unterscheidet grundsätzlich zwei Arten der Systemumgebung: die Laufzeitumgebung, in der ein Softwaresystem produktiv eingesetzt wird, und die Entwicklungsumgebung, in der es entwickelt und gewartet wird. Portierbarkeit bezieht sich zunächst auf die Laufzeitumgebung des Systems[12], die im einfachsten Fall aus der Hardware, d.h. aus den Rechnern und Netzwerkkomponenten, und aus der Betriebssystemsoftware besteht. Bei betrieblichen Informationssystemen gehören dazu in der Regel außerdem folgende Software-Komponenten, die die technische Infrastruktur darstellen: Datenbanksystem, Netzwerksystem, TP-Monitor oder Application Server und Web Server; ergänzt werden sie durch fachliche Nachbarsysteme, z.B. Systeme für ERP (Enterprise Resource Planning), CRM (Customer Relationship Management) etc. Die Entwicklungsumgebung setzt sich zusammen aus Compiler, Debugger und Konfigurationsmanagementsystem – oft in einem Produkt zusammengefaßt – und eventuell einer Testumgebung, die beispielsweise eine Testdatenbank und Testversionen von fachlichen Nachbarsystemen enthält. Falls die Software an eine geänderte Umgebung angepaßt werden soll, kann es ggf. notwendig sein, sie zusätzlich zur geänderten Laufzeitumgebung auch auf die geänderte Entwicklungsumgebung zu portieren.

---

[12] Den Begriff *Systemumgebung* verwenden wir daher im Folgenden synonym mit Laufzeitumgebung.

Variabilität und Klassifikation nach Art der Wiederverwendbarkeit

Adaptierbarkeit und Portierbarkeit können unter dem Oberbegriff der *Variabilität* zusammengefaßt werden. Entsprechend nimmt die Wiederverwendbarkeit mit steigender Variabilität zu. Eine Reihe von Mechanismen existieren, mit deren Hilfe Variabilität technisch realisiert werden kann. Sie setzen meist auf der Ebene der Architektur an; 2.3.5 gibt einen Überblick über die verschiedenen Variabilitätsmechanismen.

Da Software Merkmale der Wiederverwendbarkeit haben kann, ohne daß diese angestrebt wurde, klassifizieren wir intern entwickelte Software danach, ob Wiederverwendbarkeit angestrebt und ob sie tatsächlich gegeben ist (siehe Abbildung 2-3). In vielen Fällen wird Wiederverwendbarkeit nicht angestrebt. Ist die Software dennoch wiederverwendbar, spricht man von *ad hoc wiederverwendbarer Software*. Ist Wiederverwendbarkeit nicht gegeben, nennen wir die Software *nicht wiederverwendbar*.

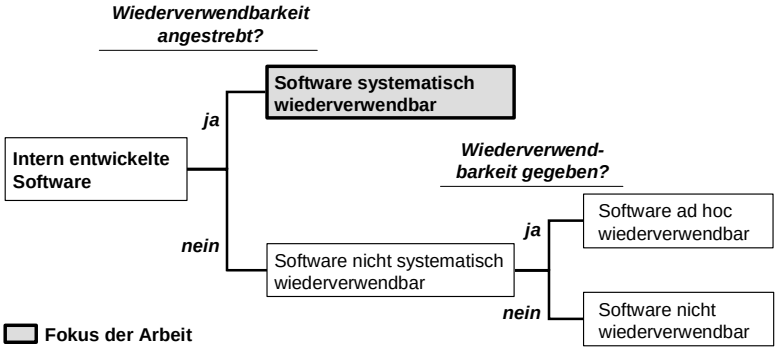

Abbildung 2-3: Klassifikation intern entwickelter Software

Wird Wiederverwendbarkeit angestrebt und nach den oben ausgeführten Kriterien systematisch hergestellt[13], spricht man von *systematisch wiederverwendbarer Software*. Auf sie konzentrieren wir uns; um die Notation zu vereinfachen, werden wir den Begriff 'wiederverwendbare Software' im Folgenden immer dann synonym mit 'systematisch wiederverwendbare Software' verwenden, wenn die explizite Abgrenzung zu ad hoc wiederverwendbarer Software nicht notwendig ist.

Die Maßnahmen, die ergriffen werden, um Software systematisch wiederverwendbar zu machen, erfordern einen erhöhten Aufwand im Vergleich zur Entwicklung nicht systematisch wiederverwendbarer Software; dies führt zu Zusatzkosten für Wiederverwendbarkeit (2.1.1; vgl. [Tra95], 74, 115; [Jon94]; [Sam97], 16). Diese zusätzlichen Kosten für die Wiederverwendbarmachung sind ökonomisch als Investition zu sehen, die sich rentieren muß ([BBo91]; [Kar95], 10; vgl. auch [McC97], 7f; eine detaillierte Behandlung erfolgt in 2.4.2).

[13] Wir gehen vom Erfolgsfall aus.

2.1.3 Was kann grundsätzlich wiederverwendet werden?

Wiederverwendung beschränkt sich nicht auf Code. In diesem Abschnitt stellen wir dar, was grundsätzlich wiederverwendet werden kann und arbeiten den Fokus dieser Arbeit heraus.

Im Sinne einer kompakten Schreibweise führen wir zunächst als Sammelbegriff für alle Zwischen- und Endprodukte, die bei der Entwicklung eines betrieblichen Informationssystems entstehen, den Begriff *Artefakte* ein (vgl. [BJR98, Jon94]; die verschiedenen Prozeßschritte und ihre Endprodukte werden in 2.3.1 definiert). Wir charakterisieren Artefakte dadurch, daß wir den Prozeßschritt angeben, bei dem sie entstehen, z.B. Artefakte der Spezifikation.

Die Menge der Artefakte, die wiederverwendet werden können, wird in einigen Publikationen sehr weit gefaßt und beschränkt sich nicht auf Software gemäß unserer Definition (vgl. 2.1.1). Teilweise ist ein Mangel an Strukturierung festzustellen, der dazu führt, daß unterschiedlichste Dinge genannt werden.[14] Je nach Art des Artefakts gibt es jedoch Unterschiede hinsichtlich des Status quo, des Nutzens und der bei der Nutzung zu überwindenden Hindernisse. Je weiter der Begriff gefaßt wird, desto weniger scharf läßt sich also herausarbeiten, welche Maßnahmen zur Verbesserung der Situation ergriffen werden sollten. Eine Klassifizierung der wiederverwendbaren Artefakte und entsprechende Fokussierung ist deshalb unabdingbar.

Klassifikation und Fokus

Wir schlagen die in Abbildung 2-4 dargestellte Klassifikation der potentiell wiederverwendbaren Artefakte vor, die auch einige Beispiele enthält. Dabei unterscheiden wir zunächst danach, ob ein Artefakt im Endprodukt des Entwicklungsprozesses[15] oder im Entwicklungsprozeß selbst wiederverwendet werden soll.

---

[14] Zur Illustration zitieren wir exemplarisch McClure [McC97], Jacobson et al. [JGJ97] und Lim [Lim98]. *McClure* gibt folgende Liste an ([McC97], xx): "Source Code, Specifications, Designs, Test Scripts, Project Plans, Documentation, Object Frameworks, Subroutines". *Jacobson et al.* sprechen von wiederverwendbaren Komponenten: " A component is a type, class or any other workproduct that has been specifically engineered to be reusable." ([JGJ97], 85). Die Definition von Workproducts ist sehr weit und geht explizit über Software hinaus: "A workproduct is a unit of code, a document or piece of software model that can be independently managed within a software engineering organization. Individual types, classes and other model elements from the various models, diagrams, and related documents are typical workproducts. Complete models, subsystems, and test models are also workproducts. Some workproducts are abstract, management-oriented entities and not always specific pieces of software. For example, a workproduct can be a configuration file, listing the names and versions of other software workproducts that are intended to be used together." ([JGJ97], 82). *Lim* zählt folgende reusable assets auf ([Lim98], 8): "Architectures, Source code, Data, Designs, Documentation, Estimates, Human interfaces, Plans, Requirements, Test cases".

[15] genauer muß es heißen: der ursprünglichen Entwicklung; siehe dazu Kapitel 2.3.1.

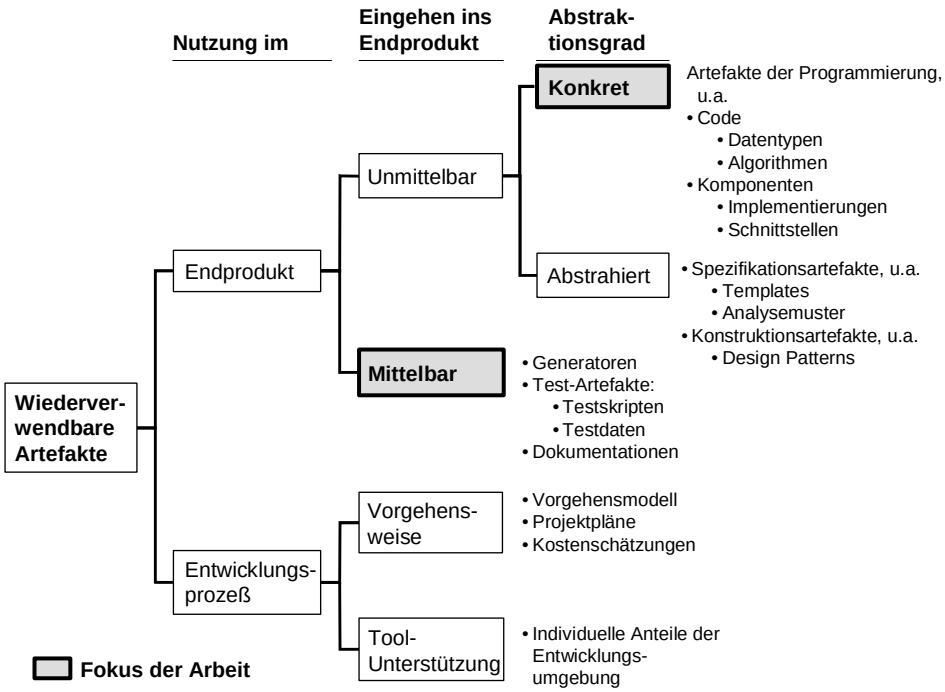

Abbildung 2-4: Einteilung wiederverwendbarer Artefakte

Bei den Artefakten, die im Endprodukt genutzt werden, unterscheiden wir danach, ob sie unmittelbar oder mittelbar in das Endprodukt eingehen. Unmittelbar in das Endprodukt eingehende Artefakte lassen sich nach ihrem Abstraktionsgrad unterscheiden, der mit zunehmendem Fortschreiten des Entwicklungsprozesses abnimmt (vgl. [Kar95], 254); der Entwicklungsprozeß kann dabei als eine Aneinanderreihung von Verfeinerungsschritten interpretiert werden. Abstrahiert in das Endprodukt eingehende Artefakte sind Artefakte der Vorstudie, Spezifikation und Konstruktion. Konkret gehen die meisten Endprodukte der Programmierung in das Endprodukt ein, sofern es sich nicht um Artefakte von Test und Dokumentation handelt. Bei den im Entwicklungsprozeß wiederverwendeten Artefakten schließlich unterscheiden wir Artefakte, die die Vorgehensweise betreffen und solche, die die Tool-Unterstützung betreffen.

Im Fokus dieser Arbeit sind Software-Komponenten; dies entspricht dem Stand der Technik (vgl. z.B. [Szy99]). Wir definieren sie grundsätzlich als klar abgrenzbare Einheiten der Entwicklung, die ihre Daten kapseln und Zugriff von außen nur über Schnittstellen ermöglichen (detailliert behandeln wir Komponenten einschließlich ihrer Vorteile für die Wiederverwendung in 2.3.3). Gemäß unserer Definition von Software (vgl. 2.1) umfaßt dies zunächst Programmcode, der unmittelbar in konkreter Form oder – als Generator – indirekt in das Endprodukt eingeht. Zusätzlich umfaßt es Dokumentationen sowie Testprogramme und Testdaten zu diesem Code.

Die Klassifikation der mittelbar in das Endprodukt eingehenden und der in abstrahierter Form unmittelbar eingehenden Artefakte ist dabei nicht eindeutig. Sie können jeweils zwei unterschiedliche Rollen einnehmen, die wir durch Metaphern aus der Chemie umschreiben: die des Katalysators, der die Nutzbarkeit für Entwickler der konkret wiederverwendeten Artefakte (Code, Kom-

ponenten) erhöht und die des Reagens, das selbständig wiederverwendet wird (siehe auch Abbildung 2-5). Der Schwerpunkt bei mittelbar wiederverwendbaren Artefakten liegt auf der Rolle des Katalysators; bei unmittelbar, abstrahiert wiederverwendbaren Artefakten liegt er auf der Rolle des Reagens. Unser Fokus entspricht der Rolle des Reagens.

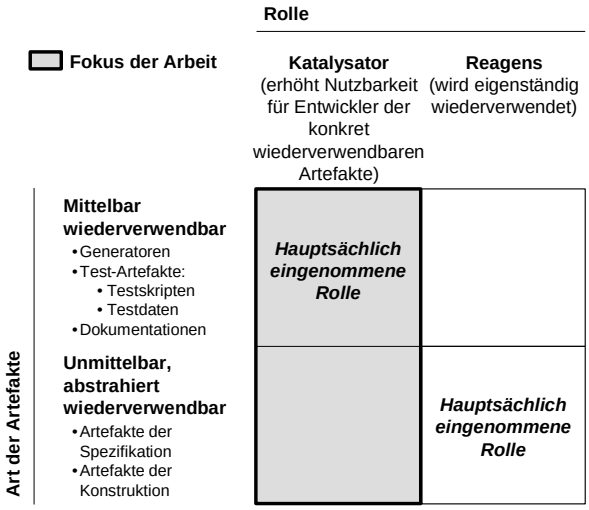

Abbildung 2-5: Rollen wiederverwendbarer Artefakte

Artefakte der Programmierung

Zu den wiederverwendbaren Artefakten der Programmierung gehören Codestücke (z.B. in Form von Unterprogrammen), Klassen (einzeln oder in Form ganzer Bibliotheken), Frameworks, Komponenten verschiedener Granularitätsklassen und Generatoren. Letztere haben eine Sonderstellung, weil sie – anders als die restlichen – mittelbar in das Endprodukt eingehen. Unmittelbar geht das jeweilige Generat in das Endprodukt ein.

Einen Überblick über die Artefakte der Programmierung gibt Abbildung 2-6; sie sind klassifiziert nach ihrer Granularität und danach, zu welchem Grad sie eine Vorgabe für die Struktur des Systems machen, in dem sie eingesetzt werden.

Wir definieren drei Typen von Komponenten auf unterschiedlichen Granularitätsstufen. Nach aufsteigender Größe sortiert sind dies: Elementarbausteine, Baugruppen und Systeme (für eine genaue Behandlung siehe 2.3.3). Ein Nachteil von Elementarbausteinen ist, daß eine große, unübersichtliche Zahl notwendig ist (vgl. [GrW95]). Daher liegt unser Fokus auf Baugruppen, die neben dem Code auch eine Strukturvorgabe liefern, was den Aufwand bei der Konstruktion reduziert.

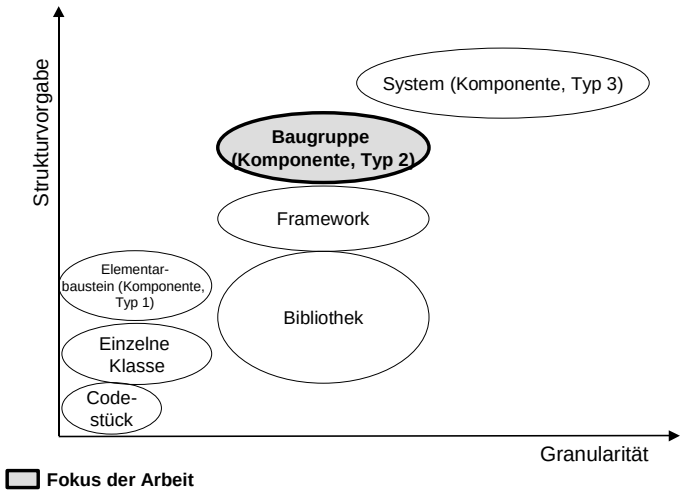

Abbildung 2-6: Wiederverwendbare Artefakte der Programmierung

Bibliotheken sind Ansammlungen von Klassen oder Elementarbaustein-Komponenten, die jeweils Lösungen für häufig auftretende Probleme allgemeiner Art bereitstellen, z.B. Sortierung, Iteration. Die einzelnen Komponenten werden meist unabhängig voneinander eingesetzt, was dazu führt, daß die Struktur des Gesamtsystems nur zu einem geringen Grad beeinflußt wird. Die Wiederverwendung erfolgt in der Regel ohne Modifikation. Gebräuchliche Bibliotheken sind beispielsweise die Java Generic Library (JGL) oder die Standard Template Library (STL) der Programmiersprachen C und C++.

Ein Framework ist ebenfalls eine Menge von Klassen. Im Gegensatz zu Bibliotheken dienen diese Klassen im Zusammenspiel der Lösung verwandter Probleme [JoF88]. Der Grad der Strukturvorgabe ist höher als bei Bibliotheken, aber geringer als bei Baugruppen. Frameworks gehören in der Regel zur Kategorie der T-Software (siehe 2.3.6) und erlauben in der Regel Anpassungen in einem gewissen Rahmen, oft als Spezialisierung durch Implementierungsvererbung. Typische Beispiele sind GUI-Frameworks wie Swing für die Programmiersprache Java oder Middlware-Frameworks wie Corba ([FSJ99], 9).

Artefakte der Spezifikation und Konstruktion

Bei den Artefakten der Spezifikation und Konstruktion unterscheiden wir zwei Rollen: die des Reagens und des Katalysators (s.o., vgl. Abbildung 2-5). Erstere entspricht eigenständiger Wiederverwendung, die der Schwerpunkt in der Klassifizierung, aber nicht im Fokus unserer Arbeit ist. Typische eigenständig wiederverwendbare Artefakte der Spezifikation sind Templates und Analysis Patterns (vgl. [Fow99]); bei der Konstruktion sind es Entwurfsmuster.

Zu den Artefakten, die die Rolle eines Katalysators spielen, also dazu dienen können, die Nutzbarkeit für Entwickler zu erhöhen, gehören alle Arten der Dokumentation des fachlichen und technischen Entwurfs eines Systems. Diese Art der Nutzung ist im Fokus unserer Arbeit.

Artefakte von Test und Dokumentation

Die Menge der Artefakte von Test und Dokumentation gehört zu den Endprodukten der Programmierungsphase, geht jedoch in der Regel nur mittelbar in das Endprodukt ein. Entscheidendes Ziel ist dabei, die Nutzbarkeit für Entwickler der wiederverwendbaren Software zu erhöhen. Diese Art der Nutzung ist im Fokus unserer Arbeit (vgl. Abbildung 2-5). Die Artefakte sind im Einzelnen Testskripten bzw. Testtreiber, Testdaten und Dokumente der Dokumentation (vgl. [McC97], xx; [Lim98], 8; [Pou97], 2; [Jon94]). Die Bereitstellung von Testtreibern und -daten für Regressionstests spielt bei wiederverwendbarer Software eine wichtige Rolle. Auch diese Artefakte können grundsätzlich eigenständig wiederverwendet werden; mit dieser Art der Nutzung beschäftigen wir uns nicht näher (vgl. Abbildung 2-5). In vielen Fällen beschränkt sie sich auf Templates.

Im Entwicklungsprozeß wiederverwendete Artefakte

Zu den Artefakten, die im Entwicklungsprozess hinsichtlich der Vorgehensweise wiederverwendet werden können, gehören Projektpläne und Kostenschätzungen (vgl. [Jon94]; [Lim98], 8; [McC97], xx). Hinsichtlich der Tool-Unterstützung können u.a. individuelle Anteile der Entwicklungsumgebung und Fehlerdatenbanken (vgl. [Pou97], 2) wiederverwendet werden. Die Nutzung dieser Artefakte ist nicht Schwerpunkt unserer Arbeit.

2.1.4 Perspektiven der Wiederverwendung: Herstellung und Nutzung wiederverwendbarer Software

Bei genauer semantischer Betrachtung steht der Begriff Software-Wiederverwendung strenggenommen für die Nutzung wiederverwendbarer Software bei der Entwicklung eines Softwaresystems. In der Literatur wird der Begriff hingegen weiter gefaßt: auch die Herstellung wiederverwendbarer Software wird unter Software-Wiederverwendung subsumiert (vgl. z.B. [JGJ97, McC97, Tra95]). Wir halten es jedoch für notwendig, begriffliche Klarheit herzustellen, indem wir die beiden Perspektiven trennen. Grund hierfür ist, daß Herstellung und Nutzung unterschiedliche Prozesse erfordern und daher auch getrennt betrachtet werden sollten (vgl. [Kar95], 33).

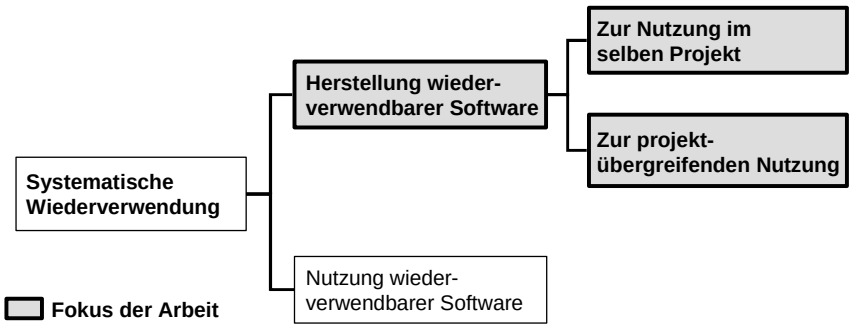

Abbildung 2-7: Perspektiven der Wiederverwendung

Demnach unterscheiden wir die zwei grundlegenden Perspektiven der systematischen Wiederverwendung: Herstellung und Nutzung wiederverwendbarer Software (vgl. [Kar95], 33ff). Einen Überblick gibt Abbildung 2-7. Die Herstellung – unser Fokus – wird oft auch als *Entwicklung für Wiederverwendung*, die Nutzung als *Entwicklung mit Wiederverwendung* bezeichnet (vgl. [JGJ97], 19f). Den beiden Perspektiven entsprechen die Rollen des Herstellers und des Nutzers wiederverwendbarer Software.

Der von Software-Wiederverwendung erwartete Nutzen entsteht bei der Nutzung wiederverwendbarer Software (eine detaillierte Untersuchung findet sich in 2.2.3); bei der Herstellung hingegen entsteht zusätzlicher Aufwand durch Maßnahmen, die die Wiederverwendbarkeit sicherstellen (vgl. [Tra95], 74, 115; zu den Maßnahmen siehe 2.1.2). Die dafür entstehenden zusätzlichen Kosten müssen durch die Einsparungen bei der Nutzung zumindest aufgewogen werden, was in 2.1.5 und 2.4 erörtert wird.

Nutzung

Auch bei der Nutzung entsteht zunächst Aufwand für verschiedene Tätigkeiten. In jedem Fall muß der nutzende Entwickler die wiederverwendbare Software finden und sich in ihre Nutzung einarbeiten. Ziel ist dabei zumindest das Verständnis der Schnittstellen. Sollen Anpassungen vorgenommen werden, so muß die Einarbeitung weiter gehen. Natürlich entsteht für das Durchführen der Anpassung selbst auch Aufwand. Zudem kann Aufwand dafür entstehen, das System an die wiederverwendete Software anzupassen.

Eine Übersicht über die Arten der Nutzung wiederverwendbarer Software zeigt Abbildung 2-8. Wir unterscheiden zunächst danach, ob die Software für die Nutzung angepaßt wird. Findet sie ohne Anpassung statt, kann dies in Form der *Black-box-Wiederverwendung* geschehen. In diesem Fall wird nur Binärcode, nicht aber Quellcode ausgeliefert. Der Nutzer kennt infolgedessen nur die Funktionalität, nicht jedoch die Details der Implementierung (vgl. [Gol95], 208). Wird der Quellcode ausgeliefert, nennen wir das *Glass-box-Wiederverwendung*: der Nutzer kann die Implementierung zwar sehen, darf aber keine Änderungen vornehmen (vgl. [Gol95], 208).

Wird die Software für die Nutzung angepaßt, kann dies durch Modifikation des Quellcodes geschehen, was man *White-box-Wiederverwendung* nennt (vgl. [Gol95], 208). Darauf liegt unser Fokus; es ist ein für interne Wiederverwendung von Komponenten gut geeignetes Vorgehen, weil Anpassungen möglich sind, diese aber nicht alle antizipiert werden müssen. Zwei Optionen gibt es hinsichtlich der Wartung [Sie96]: sie kann mit einer gemeinsamen Versionsführung zentral oder aber dezentral durchgeführt werden. Bei kurzfristiger Betrachtung sind die Kosten von zentraler Wartung höher.

Anders ist es bei einer Anpassung ohne Modifikation des Quellcodes. Man spricht in diesem Fall von *Customizing*, das bei kommerziell verfügbaren Produkten üblich ist (vgl. [Buc99], 301f). Hier müssen alle Änderungen antizipiert werden, um sie durch eine entsprechende Parametrisierung bzw. entsprechende user exits vorzubereiten.

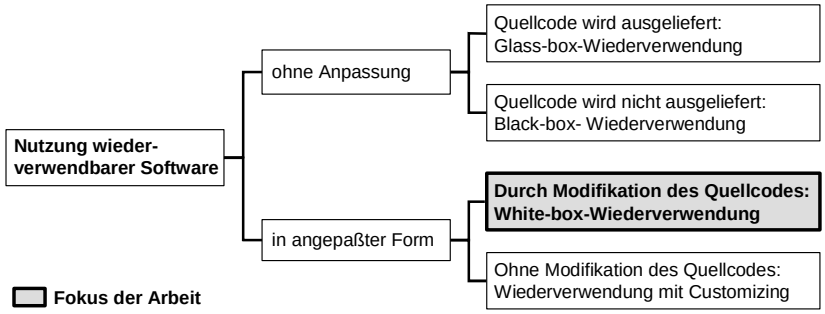

Abbildung 2-8: Arten der Nutzung wiederverwendbarer Software

Herstellung

Unser Ziel ist, Leitlinien für die Entwicklung wiederverwendbarer Software zu formulieren; wir konzentrieren uns also auf die Herstellung (siehe auch 1.3). Wir können sie aber nicht vollkommen getrennt von der Nutzung betrachten, da die geplante Nutzung immer eine Richtschnur für die Herstellung von Software ist, insbesondere wenn es darum geht, die Funktionalität zu spezifizieren. Die Nutzung untersuchen wir jedoch nur in dem Maß, als wir Erkenntnisse für die Herstellung gewinnen können. Dies umfaßt z.B. die geplante Art der Nutzung wiederverwendbarer Software. Mit allen Aspekten der Nutzung, die keine Relevanz für die Herstellung an sich haben, z.B. mit den organisatorischen Implikationen oder der operativen Umsetzung, beschäftigen wir uns nicht. Bei unserer Fokussierung unterscheiden wir nicht danach, ob die Nutzung innerhalb des selben Projekts oder projektübergreifend geplant ist (vgl. Abbildung 2-7).[16]

Es gibt zwei Alternativen bei der Entwicklung wiederverwendbarer Software, die sich darin unterscheiden, ob Wiederverwendbarkeit von Beginn an angestrebt wird oder nicht (vgl. [Kar95], 37). Grundsätzlich kann Software, die zunächst nicht systematisch wiederverwendbar entwickelt wurde[17], nachträglich zur systematischen Wiederverwendung aufbereitet werden. Diese Art von Software nennen wir *a posteriori* wiederverwendbar. Um sie zur Wiederverwendung aufzubereiten, wird im Anschluß an die ursprüngliche Entwicklung ein Projekt durchgeführt, das die Software so modifiziert oder erweitert, daß deren Wiederverwendbarkeit im o.g. Sinne erhöht wird; wir sprechen auch von *Wiederverwendbarmachung* (siehe Abbildung 2-9). Voraussetzung ist selbstverständlich, daß sich die fragliche Software dafür eignet.

[16] Projektübergreifende Wiederverwendung wird auch "[Projekt-]extern", Wiederverwendung im selben Projekt "[Projekt-]intern" genannt (vgl. [Lim98], 396); letzteres sollte nicht mit [Firmen-]interner Wiederverwendung im von uns definierten Sinn verwechselt werden.

[17] Dies nennen wir die *ursprüngliche* Entwicklung (vgl. 2.3.1)

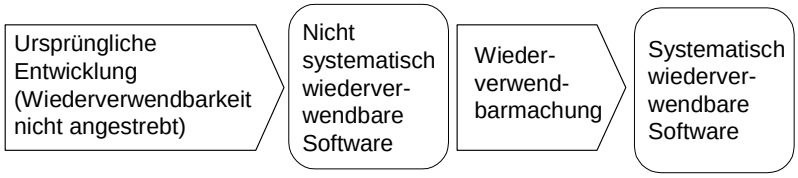

Abbildung 2-9: A-Posteriori-Wiederverwendbarkeit

Wird Software hingegen von vornherein so entwickelt, daß sie wiederverwendbar ist, nennen wir sie *a priori* wiederverwendbar. In diesem Fall sind ursprüngliche Entwicklung und Aufbereitung zur Wiederverwendung nicht getrennte Aktivitäten, sondern fallen zusammen; die Entwicklung wiederverwendbarer Software umfaßt beide (siehe Abbildung 2-10).

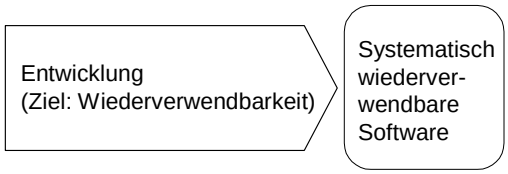

Abbildung 2-10: A-Priori-Wiederverwendbarkeit

Für die Aufbereitung zur Wiederverwendung werden wir im folgenden auch den Begriff *Wiederverwendbarmachung* verwenden. Die dabei durchzuführenden Maßnahmen werden im vorangegangenen Abschnitt (2.1.2) erläutert.

Der A-Posteriori-Wiederverwendbarkeit sind durch die Designentscheidungen bei der ursprünglichen Entwicklung natürliche Grenzen gesetzt ([Kar95], 37). Teilweise wird daher in der Literatur A-Priori-Wiederverwendbarkeit als überlegener Ansatz propagiert (vgl. [Lim98], 403). Allerdings setzt die erfolgreiche Entwicklung von a priori wiederverwendbarer Software große Weitsicht bei Spezifikation und Design voraus, da zukünftige Anforderungen antizipiert werden müssen (vgl. [GHJ94], 23), was in der Regel sehr schwierig ist [OsT01].

2.1.5 Grundlegendes ökonomisches Modell der Wiederverwendung

In diesem Abschnitt stellen wir die Grundlagen des in dieser Arbeit verwendeten ökonomischen Modells der Wiederverwendung vor. Zunächst gehen wir auf das Modell des Nutzungsprozesses ein (siehe Abbildung 2-11): Auf die Herstellung einer wiederverwendbaren Komponente folgt eine Anzahl von Nutzungen, deren jährliche Rate wir durch den Parameter λ_0 beschreiben. Die Nutzungsprojekte können dabei parallel ablaufen, wie in Abbildung 2-11 schematisch gezeigt.

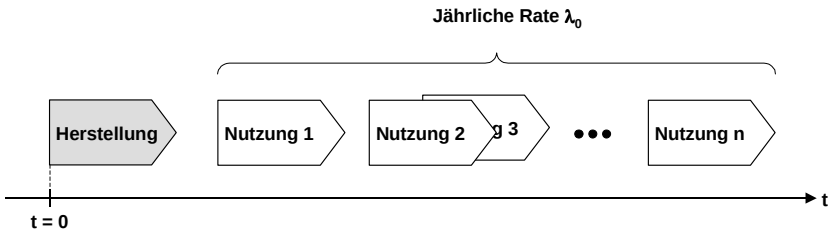

Abbildung 2-11: Nutzungsprozeß

Die Kosten der Herstellung zeigt Abbildung 2-12. Die Wiederverwendbarmachung kann dabei – wie in 2.1.4 dargelegt – a posteriori oder a priori erfolgen. Im ersten Fall entstehen für die ursprüngliche Entwicklung die Kosten K_e, die wir für den zweiten Fall fiktiv ebenfalls ansetzen (diese *ursprünglichen Entwicklungskosten* K_e sind die Bezugsgröße für unsere gesamte Wirtschaftlichkeitsbetrachtung). Die Kosten für die Wiederverwendbarmachung werden durch den *Faktor der Investition in Wiederverwendbarkeit* i_{wv} beschrieben. Die *Investition in Wiederverwendbarkeit* I_{wv} beträgt in beiden Fällen $I_{wv} = i_{wv} K_e$.

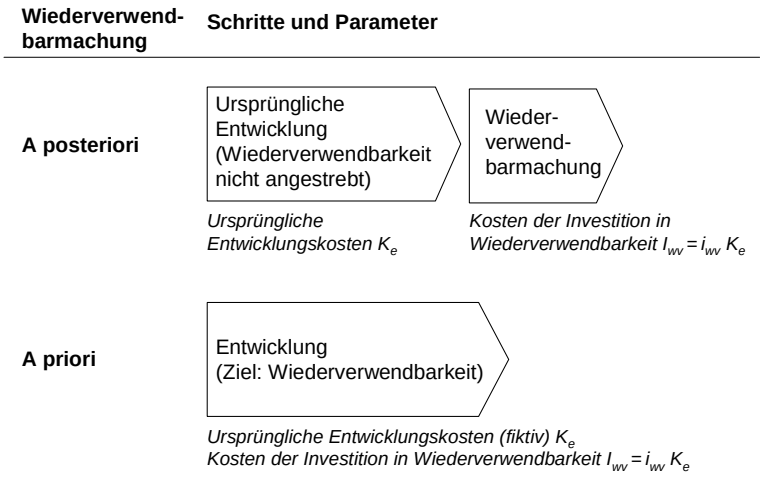

Abbildung 2-12: Kosten der Herstellung wiederverwendbarer Software

Die Entstehung der Ersparnis pro Nutzung wiederverwendbarer Software ist in Abbildung 2-13 dargestellt. Die bei jeder Nutzung anfallenden *Nutzungskosten* K_n (für Suche, Einarbeitung und Anpassung, vgl. 2.1.4) werden relativ zu K_e durch den *Nutzungskostenfaktor* κ beschrieben und betragen $K_n = \kappa K_e$.

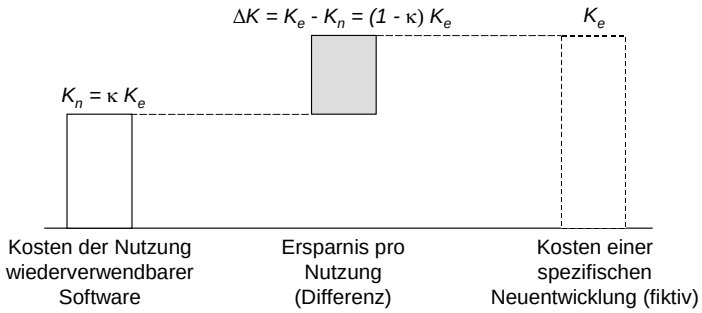

Abbildung 2-13: Entstehung der Kostenersparnis bei der Nutzung

Dabei gehen wir davon aus, daß die Alternative eine Neuentwicklung wäre, bei der fiktive Entwicklungskosten in Höhe der ursprünglichen Entwicklungskosten K_e anfallen. Die *Kostenersparnis pro Nutzung* ΔK beträgt also $\Delta K = (1-\kappa)K_e$.

Den zeitlichen Verlauf der Geldflüsse bei Herstellung und Nutzung im Überblick zeigt schematisch Abbildung 2-14.

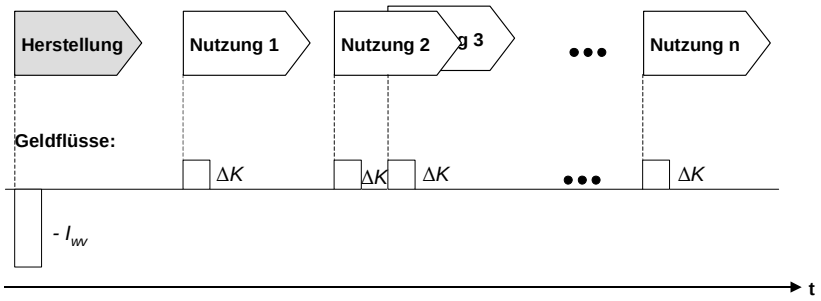

Abbildung 2-14: Geldflüsse im Überblick

2.1.6 Differenzierter Begriff der Wiederverwendung

Wie in 2.1.1 festgestellt, sollte der Begriff der systematischen Wiederverwendung weiter ausdifferenziert werden, wozu verschiedene Merkmale herangezogen werden können. Zunächst betrachten wir die Nutzungsrate (siehe Abbildung 2-15), anhand derer wir zwei Varianten unterscheiden.

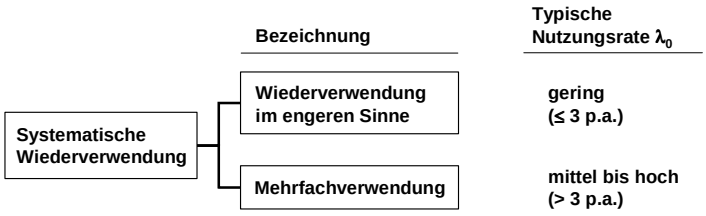

Abbildung 2-15: Varianten systematischer Wiederverwendung

Ist die erwartete Nutzungsrate gering, so bezeichnen wir dies als *Wiederverwendung im engeren Sinne*, weil es der intuitiven semantischen Bedeutung des Begriffs enspricht. Als Obergrenze legen wir eine Anzahl von drei Nutzungen pro Jahr fest, wobei wir uns an den Erkenntnissen der in Kapitel 4 und Kapitel 5 präsentierten Fallstudien orientieren (andere Werte im einstelligen Bereich sind je nach den konkreten Umständen vertretbar; die exakte Festlegung der Grenze ist letzlich eine Frage der Definition). Wenn eine mittlere bis hohe Nutzungsrate erwartet wird, d.h. über drei Nutzungen pro Jahr, sprechen wir von *Mehrfachverwendung*.

Die zwei Varianten der Wiederverwendung unterscheiden sich in drei Merkmalen:

- *Wiederverwendbarmachung*. Während sie bei Wiederverwendung im engeren Sinne ausschließlich a posteriori erfolgt, wird sie bei Mehrfachverwendung hauptsächlich a priori durchgeführt.

- *Investition in Wiederverwendbarkeit*. Die Investition bei Wiederverwendung im engeren Sinne ist geringer als bei Mehrfachverwendung. Dies hat zwei Gründe: zum einen steigt die erreichbare Nutzungsrate λ_0 mit zunehmender Wiederverwendbarkeit, die wiederum mit der Investition zunimmt, zum anderen liegt die Obergrenze für die Wirtschaftlichkeit der Investition wegen der höheren Ersparnis bei Mehrfachverwendung höher. Wir gehen davon aus, daß typischerweise die absolute Höhe der Investition bei Wiederverwendung im engeren Sinne geringer und bei Mehrfachverwendung höher ist als die ursprünglichen Entwicklungskosten K_e. Die Ergebnisse der Fallstudien aus Kapitel 5 sprechen für diese Festlegung. Allerdings sind nicht alle Kombinationen von Werten für λ_0 und i_{wv} wirtschaftlich sinnvoll.
- *Nutzungskosten*. Mit steigender Investition in Wiederverwendbarkeit nehmen die Nutzungskosten pro Nutzungsereignis ab, weil Suche, Einarbeitung und Anpassung erleichtert werden. Daher liegen die Nutzungskosten bei Mehrfachverwendung wegen der höheren Investition grundsätzlich niedriger als bei Wiederverwendung im engeren Sinne. Allerdings ist davon auszugehen, daß ein Teil dieses Vorteils dadurch aufgezehrt wird, daß der Anwendungsbereich größer wird und die Investitionskosten überproportional mit der Nutzungsrate steigen.

Aus der jeweiligen Kombination von Nutzungsrate λ_0, Faktor der Investition in Wiederverwendbarkeit i_{wv} und Nutzungskostenfaktor κ kann man berechnen, ob die Investition wirtschaftlich lohnend ist. In einer Überschlagsrechnung[18] bestimmen wir daher die Obergrenze, die diese Kosten nicht überschreiten dürfen. Dabei gehen wir vereinfachend davon aus, daß die Entwicklungskosten ohne Wiederverwendung in jedem Fall den Kosten für eine spezifische Entwicklung entsprechen. Außerdem beschränken wir die Betrachtung auf ein Jahr und rechnen mit Nominalbeträgen, was zu – in diesem Fall vertretbaren – Ungenauigkeiten führt. Eine genaue Betrachtung erfordert ein wesentlich aufwendigeres Vorgehen, das in 2.4 beschrieben wird; exakte quantitative Ergebnisse stellt Kapitel 6 vor. Für die hier gemachten allgemeinen Aussagen ist die Genauigkeit der Überschlagsrechnung jedoch ausreichend. Einige exemplarische Ergebnisse für die Obergrenze, die κ nicht überschreiten darf, zeigt Tabelle 2-1. Wir gehen dabei davon aus, daß der Aufwand für Wiederverwendbarkeit überproportional mit der Nutzungsrate steigt (s.o.).

Nutzungsrate λ_0	Investition in Wiederverwendbarkeit (Faktor) i_{wv}	Obergrenze für Nutzungskostenfaktor κ
1	**0,1**	**0,9**
4	**1**	**0,75**
10	**5**	**0,5**

Tabelle 2-1: Obergrenze für κ (Ergebnisse der Überschlagsrechnung)

[18] $\kappa \leq 1 - \frac{i_{wv}}{\lambda_0}$

In Tabelle 2-2 sind die beschriebenen Merkmale der Varianten systematischer Wiederverwendung zusammengestellt. Für den Nutzungskostenfaktor geben wir nur den jeweiligen Maximalwert an. Grundsätzlich ist davon auszugehen, daß er bei Mehrfachverwendung niedriger liegt als bei Wiederverwendung im engeren Sinne.

	Nutzungsrate λ_0	Wiederverwendbarmachung	Investition in Wiederverwendbarkeit (Faktor) i_{wv}[19]	Nutzungskostenfaktor κ[19]
Wiederverwendung im engeren Sinne	**≤ 3**	**A posteriori (ausschließlich)**	**< 1**	**< 1**
Mehrfachverwendung	**> 3**	**A priori (Schwerpunkt)**	**≥ 1**	**< 0,75**

Tabelle 2-2: Merkmale der Varianten systematischer Wiederverwendung

Dieselbe Betrachtung führen wir für Ad-Hoc-Wiederverwendung durch (siehe Tabelle 2-3). Hier entstehen keine Zusatzkosten für Wiederverwendbarkeit. Die Obergrenze für die Nutzungskosten entspricht daher unabhängig von der Zahl der Nutzungen den Kosten für eine spezifische Entwicklung.

	Nutzungsrate λ_0	Investition in Wiederverwendbarkeit (Faktor) i_{wv}	Nutzungskostenfaktor κ
Ad-Hoc-Wiederverwendung	**beliebig (in der Regel gering)**	**keine**	**1**

Tabelle 2-3: Quantitative Betrachtung von Ad-Hoc-Wiederverwendung

Auch die Wiederverwendbarkeit läßt sich analog zur Wiederverwendung nach der Höhe der Investition weiter untergliedern, wie dies in Abbildung 2-16 illustriert ist.

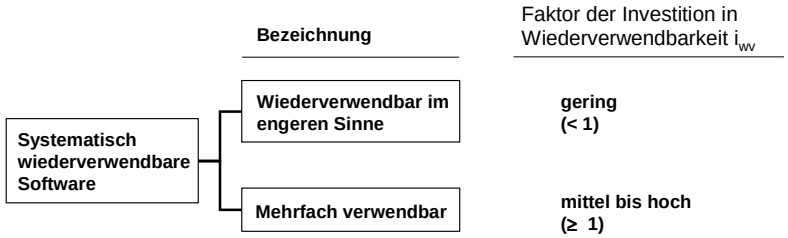

Abbildung 2-16: Untergliederung systematischer Wiederverwendbarkeit

[19] Typische Werte

2.2 Motivation und Rahmenbedingungen von Wiederverwendung

Voraussetzung für das Verständnis der Probleme bei der Entwicklung wiederverwendbarer Software ist die genaue Kenntnis von Motivation und Rahmenbedingungen, unter denen sie erfolgt. Grundlage für beide ist der historische Hintergrund. Wir gehen daher im Folgenden auf den historischen Ursprung, die aktuellen Rahmenbedingungen sowie erwarteten Nutzen und Hindernisse ein.

2.2.1 Historische Wurzeln: Die Software-Krise

Im Jahr 1968 fand in Garmisch die NATO Working Conference on Software Engineering statt. Ziel der Konferenz war es, das noch junge Gebiet der Softwareentwicklung zu einer Ingenieurdisziplin zu machen und notwendige Schritte festzulegen. Den Ausgangspunkt bildete dabei eine Analyse der Situation der Softwareentwicklung im industriellen Maßstab. Die Ergebnisse der Analyse waren niederschmetternd: der Bedarf an Software werde nicht erfüllt, die vorhandene Software sei von unzureichender Qualität und zu teuer und die Projekte seien schwer beherrschbar und hielten insgesamt ihren Zeitplan nicht ein [Cox90]. Zur Beschreibung dieses Zustands wurde der Begriff "Software-Krise" geprägt. Wesentliche Facetten der Diagnose des Zustands sind im Folgenden zusammengefaßt (vgl. [Frc95], 10ff):

- Es bestand ein Anwendungsstau: der Bedarf an Anwendungssoftware überstieg die Verfügbarkeit,
- Termin- und Kostenrahmen von Softwareprojekten wurden grundsätzlich überzogen und
- die Qualität der entwickelten Software war nicht ausreichend.

Eine wesentliche Möglichkeit zur Lösung dieser Probleme sah man darin, bewährte Praktiken aus Ingenieurdisziplinen wie Maschinenbau oder Elektrotechnik zu übernehmen (vgl. [McI68]). Wiederverwendung standardisierter Komponenten gilt allgemein als solche Praxis; sie war und ist z.B. im Maschinen- oder Automobilbau gang und gäbe. Einen vielbeachteten ersten Anstoß dazu, sie in die Softwareentwicklung zu übertragen, gab McIlroy [McI68]. In späteren Analysen wurde der geringe Grad an Wiederverwendung in der Softwareentwicklung als eine wesentliche Ursache des konstatierten Mangels sowohl an Produktivität als auch an Qualität genannt (vgl. [Cox90]).

Es gab sogar Versuche, detaillierte Parallelen zu anderen Disziplinen herzustellen. Als Metapher für ein Softwaresystem wurde beispielsweise der Halbleiter-Chip vorgeschlagen [Cox87]: so wie ein Chip aus einer Vielzahl ähnlicher Blöcken mit festen Schnittstellen modular aufgebaut wird, sollte auch Software konstruiert werden. In jüngeren Veröffentlichungen zum Thema der Komponenten spielt diese Idee jedoch keine Rolle mehr (vgl. z.B. [Szy99]).

2.2.2 Aktuelle Rahmenbedingungen der Softwareentwicklung

Wenn man den Status quo der Wiederverwendung verstehen will, kommt man nicht umhin, die Umstände zu betrachten, unter denen Software entwickelt wird. Wir fassen daher im Folgenden die wesentlichen Rahmenbedingungen in der heutigen Softwareindustrie kurz zusammen. Dabei gehen wir zunächst auf die Gemeinsamkeiten von Produkt- und Projektgeschäft ein und behandeln anschließend die Unterschiede.

Die sog. Eintrittsbarrieren[20] in der Softwareindustrie sind niedrig (vgl. [Hoc99], 11ff): die notwendigen Investitionskosten sind gering, die Märkte sind weltweit nicht reguliert, es gibt keine Standortanforderungen hinsichtlich der Verfügbarkeit von Rohstoffen oder der Verkehrsanbindung. Die einzige Eintrittsbarriere ist der Bedarf an ausreichend ausgebildetem Personal, das jedoch mittlerweile weltweit verfügbar ist, wie der Erfolg der indischen und israelischen Softwareindustrie zeigt ([Hoc99], 10). Der hieraus resultierende Konkurrenzdruck ist sehr hoch.

Ein weiteres wesentliches Merkmal der Softwareindustrie ist die kurze Dauer der Innovationszyklen, die im Durchschnitt bei 18 bis 24 Monaten liegt. Dies führt zu extrem kurzen unternehmerischen Planungshorizonten (vgl. [Hoc99], 11f). In Verbindung mit dem hohen Konkurrenzdruck führt dies dazu, daß in der Regel unter großem Zeitdruck entwickelt wird. Die in der Regel als 'time to market' bezeichnete Entwicklungsdauer spielt – besonders im Produktgeschäft – die entscheidende Rolle für die Konkurrenzfähigkeit am Markt. Die Anforderungen an Innovationskraft und dafür notwendige Produktivität in der Entwicklung sind deshalb hoch (vgl. [JGJ97], 11). Dies wird verschärft durch den Mangel an Fachpersonal, der bei längerfristiger Betrachtung als eine der größten Herausforderungen überhaupt zu sehen ist (vgl. [Hoc99], 68f).

Im Zusammenhang mit den kurzen Innovationszyklen steht die hohe Änderungsgeschwindigkeit der technischen Basis, auf der Software entwickelt und genutzt wird. Einige wichtige Änderungen wollen wir kurz erwähnen. Zunächst wird die Rechenleistung der Computer immer höher und die Speicherkapazitäten immer größer. Neben den Rechnern selbst ändert sich die Infrastruktur: zunehmend sind Rechner innerhalb und außerhalb des Unternehmens vernetzt. Eine geänderte technische Basis führt in der Regel zu geänderten Anforderungen an die eingesetzte Software. Zur Illustration geben wir drei Beispiele: Graphische Benutzeroberflächen (GUIs) wurden nur durch die höhere Leistungsfähigkeit der Rechner möglich; mittlerweile erfolgt die Anbindung dieser GUIs in zunehmendem Maße über das Internet (vgl. [KRW01]). Die zunehmende Verteilung von Komponenten schließlich ist die Folge der starken Vernetzung.

Zusätzlich zu den kurzen Innovationszyklen erschwert ein weiterer Faktor eine stetige, auf langfristigen Erfolg angelegte Planung: viele Softwareunternehmen sind börsennotiert und müssen darauf achten, daß ihr Aktienkurs keine zu starken Ausschläge nach unten zeigt. Die Börse jedoch reagiert negativ auf Ver-

[20] Eintrittsbarrieren werden definiert als Bedingungen, die den Markteintritt in einem bestimmten Geschäft schwierig machen (vgl. [Fri94], 47).

ringerungen des Ergebnisses, selbst wenn diese nur kurzfristiger Natur sind. Dieses Phänomen ist in der Softwareindustrie besonders ausgeprägt (vgl. [FSS99]), weil die Erwartungen angesichts der Erfolge in der jüngeren Vergangenheit besonders hoch sind.

Neben diesen Gemeinsamkeiten gibt es eine Reihe von Unterschieden in den Rahmenbedingungen der zwei wesentlichen Bereiche der Softwareindustrie, nämlich Produkt- und Projektgeschäft; siehe dazu Tabelle 2-4.

Kriterium	Produktgeschäft	Projektgeschäft
Grenzkosten der Produktion[21]	Fast Null	Fast konstant
Marktstruktur	Konzentriert, global	Fragmentiert, regional
Marketing	Massenmarketing	Individuell
Kundenverhältnis	Kurz-bis mittelfristig	Langfristige Bindung
Ziele in der Entwicklung	Abdecken der wichtigsten Anforderungen	Individuelle Anforderungen möglichst genau erfüllen
Entscheidender Indikator für Unternehmenserfolg	Marktanteil	Auslastung

Tabelle 2-4: Wesentliche Unterschiede zwischen Produkt- und Projektgeschäft (nach [Stü99])

2.2.3 Nutzen der Wiederverwendung

Von Software-Wiederverwendung wird großer Nutzen erwartet, vorrangig – aber nicht nur – in Form gesteigerter Produktivität (vgl. [Kar95], 8; [Tra95], 132f; [Den91], 17; siehe auch 3.5.3). Goldberg und Rubin erwarten beispielsweise einen Produktivitätssprung in Höhe einer Größenordnung ([Gol95], 204); Karlsson führt Beispiele an, bei denen eine Verbesserung der Produktivität um Faktor zwei erreicht wurde ([Kar95], 11). Der Begriff der Produktivität bleibt dort unklar, wie auch in anderen Veröffentlichungen (vgl. [Lim98], 102; [Sam97], 12; [Pou97], 7). Wir greifen deshalb auf die Produktivitätsdefinition nach Taylor zurück (vgl. [Hel01], 11): Produktivität entspricht dem Verhältnis von Ergebnis zu dafür aufgewendeter Arbeit. Das Ergebnis ist dabei die Menge an Software, gemessen z.B. in Function Points, die entwickelt wird. Die dafür aufgewendete Arbeit kann beispielsweise in Bearbeitertagen gemessen werden. Somit ist die Produktivität umgekehrt proportional zu den Entwicklungskosten.

[21] Grenzkosten sind definiert als zusätzliche Kosten, die für die Herstellung einer zusätzlichen Einheit des Produkts entsteht ([Fri94], 361).

Wie bereits in 2.1.2 erwähnt, entsteht dieser Nutzen durch Einsatz wiederverwendbarer Software. Eine umfassende Analyse zeigt, daß er in drei grundsätzlichen Arten entsteht (vgl. [Sam97], 11ff; daneben auch [Lim98], 102; [Pou97], 7): als reduzierter Aufwand für Entwicklung und Wartung, als erhöhte Qualität der entwickelten Software und in Form übergeordneter Effekte. Diese Arten sind nicht notwendigerweise unabhängig voneinander, worauf wir im Folgenden eingehen. Jede Art des Nutzens kann wiederum in verschiedenen Ausprägungen auftreten, für die dasselbe gilt. Einen Überblick hierüber gibt Abbildung 2-17.

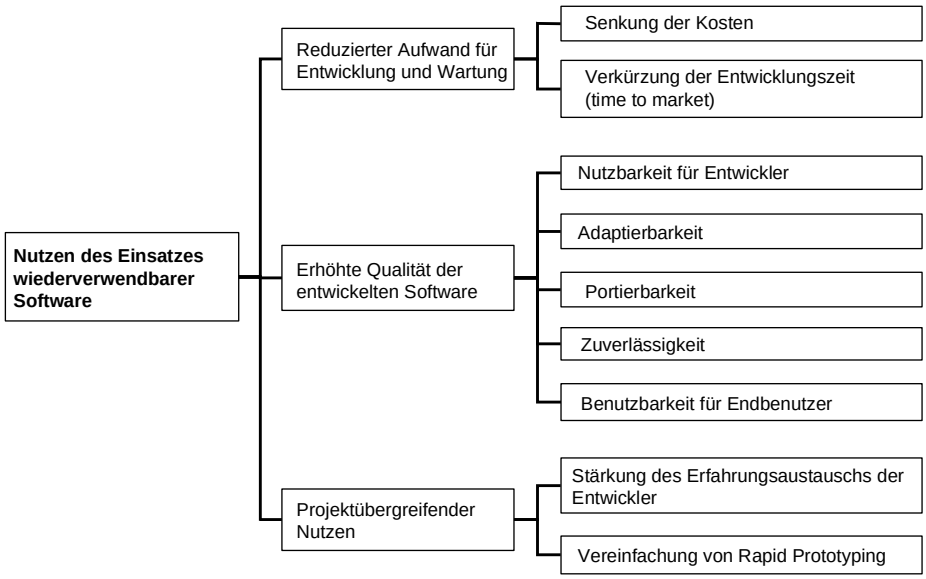

Abbildung 2-17: Nutzen des Einsatzes wiederverwendbarer Software

Reduzierter Aufwand hat grundsätzlich zwei Ausprägungen: Senkung der Kosten und Verkürzung der Entwicklungszeit (oft als time to market bezeichnet). Die Verkürzung der Entwicklungszeit ist allerdings nur zu realisieren, wenn ein entsprechendes Entwicklungsteam zur Verfügung steht und die verbleibenden Aufgaben ausreichend parallelisiert werden können (vgl. [Bro95], 16ff). Zurückzuführen ist der reduzierte Aufwand zunächst auf den geringeren Anteil der Neuentwicklung. Dies betrifft neben der Programmierungsphase[22] (Codierung und Test) und Integrationstest auch Spezifikation und Konstruktion, vor allem, wenn auch Artefakte dieser Phasen wiederverwendet werden (vgl. 2.1.3). Eine überproportionale Verbesserung kann dadurch eintreten, daß der Kommunikationsaufwand wegen der möglichen Verkleinerung des Teams abnimmt ([Sam97], 13). Daneben wird der individuelle Wartungsaufwand bei zentraler Wartung wiederverwendbarer Software reduziert.

Die verschiedenen Ausprägungen *erhöhter Qualität* sind Nutzbarkeit für Entwickler, Adaptierbarkeit, Portierbarkeit, Zuverlässigkeit und Benutzbarkeit für den Endbenutzer. Nutzbarkeit für Entwickler setzt sich zusammen aus Verständlichkeit, Änderungseffizienz und Verfügbarkeit (vgl. 2.1.2). Erhöhte Qualität kann auf zwei Arten entstehen. Zunächst bedeutet erhöhte Wiederver-

[22] Die Phasen der Entwicklung werden in 2.3.1 behandelt.

wendbarkeit gleichzeitig erhöhte Qualität, denn die Merkmale, die wiederverwendbare Software auszeichnen (Nutzbarkeit für Entwickler, Adaptierbarkeit, Portierbarkeit), sind Qualitätsmerkmale (vgl. 2.1.2). Daneben können zwei weitere Merkmale positiv beeinflußt werden: Zuverlässigkeit und Benutzbarkeit für den Endbenutzer. Erhöhte Zuverlässigkeit entsteht, wenn die Nutzer wiederverwendbarer Software gefundene Programmierfehler an den Hersteller melden und dieser die Korrekturen übernimmt. Aufgrund dieses Rückflusses werden wiederverwendbare Komponenten im Laufe ihres Lebens gründlicher getestet als nicht systematisch wiederverwendbare Komponenten und werden somit immer zuverlässiger. Die Benutzbarkeit für den Endbenutzer kann sich dadurch erhöhen, daß ein Softwaresystem aufgrund mehrfachen Einsatzes wiederverwendbarer Komponenten einheitlicher und dadurch leichter zu bedienen wird. Dies ist beispielsweise, aber nicht nur, bei der Wiederverwendung von Oberflächenelementen der Fall. Auch die Anwendungslogik kann als Nebeneffekt der Wiederverwendung homogenisiert werden, so daß die Bedienung intuitiver wird.

Erhöhte Qualität kann den Aufwand bei unverändertem Anteil der Neuentwicklung zusätzlich verringern: bessere Verständlichkeit verringert den Aufwand bei der Einarbeitung, höhere Zuverlässigkeit senkt den Testaufwand, Adaptierbarkeit und Portierbarkeit wirken sich auf den Aufwand bei Konstruktion, Programmierung und Wartung aus.

Darüber hinaus kann sich *projektübergreifender Nutzen* in zwei Ausprägungen einstellen ([Sam97], 13f): zum einen kann der Erfahrungsaustausch der Entwickler gestärkt werden, so daß sie gegenseitig voneinander lernen. Zum anderen vereinfacht sich Rapid Prototyping, wenn wiederverwendbare Komponenten verfügbar sind, die dafür eingesetzt werden können.

Der Nutzen ist nur im Fall erhöhter Produktivität unmittelbar finanziell wirksam. Das heißt nicht, daß sich die anderen Effekte – vor allem Verkürzung der Entwicklungszeit, erhöhte Kundenzufriedenheit und Vergrößerung des potentiellen Marktes – finanziell nicht auswirken. Sie können u.a. eine Umsatzsteigerung bewirken. Diese Auswirkung ist jedoch nur mittelbar und damit wesentlich schwerer zu quantifizieren, zumal keine standardisierten Schätzverfahren wie für die Aufwandsschätzung verfügbar sind.

Welche Bedeutung die unterschiedlichen Ausprägungen des Nutzens haben und wie der Nutzen ausgeschöpft werden sollte, sind strategische Fragen, deren Beantwortung von Art und Situation des Unternehmens abhängen. Wir beschäftigen uns damit ausführlich in Kapitel 7.

2.2.4 Hindernisse für Wiederverwendung

Eine Reihe von Faktoren wird dafür verantwortlich gemacht, daß Wiederverwendung erschwert bzw. verhindert wird [Bro95, CaE95, CCo94, GAO95, JGJ97, Sam97, Tra95]. Wir fassen die wesentlichen Faktoren – kurz Hindernisse genannt – hier in einem Überblick zusammen. Die jeweilige Gewichtung wird im Verlauf der Arbeit näher zu untersuchen sein.

Exakt definiert umfassen die Hindernisse alle Faktoren, die dazu führen, daß wiederverwendbare Software in geringerem Maß genutzt wird als aus unternehmerischer Sicht sinnvoll ist. Der Ursprung kann darin liegen, daß bereits die Herstellung wiederverwendbarer Software behindert wird, so daß nicht in ausreichender Menge Software zur Verfügung steht, die hinreichend wiederverwendbar ist. Ist dies nicht der Fall, so gibt es eine Reihe von Faktoren, die die Nutzung dieser Software behindern können. Einen Überblick über die Einteilung der Hindernisse gibt Abbildung 2-18. Wir gehen im Folgenden auf die einzelnen Faktoren näher ein.

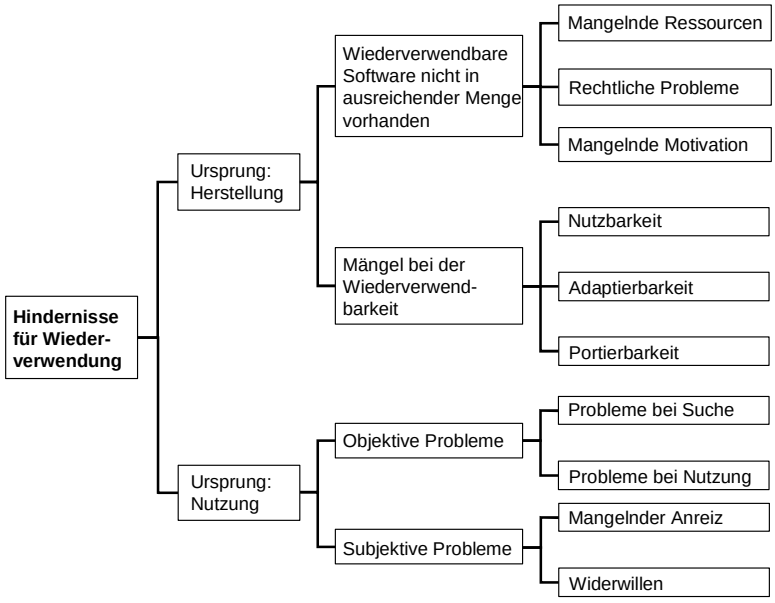

Abbildung 2-18: Hindernisse für Wiederverwendung

Hindernisse, deren Ursprung in der Herstellung liegt. Es gibt zwei Arten von Hindernissen, die ihren Ursprung in der Herstellung haben:

1. Wiederverwendbare Software ist nicht in ausreichender Menge vorhanden ([GAO95]; [Tra95], 137; [JGJ97], 8). Drei wesentliche Gründe sind dafür möglich:

   (a) *Mangelnde Ressourcen*: Es stehen keine ausreichenden finanziellen Ressourcen (in Form eines Projektbudgets) für die Herstellung wiederverwendbarer Software zur Verfügung. Sie sind notwendig, weil Wiederverwendbarmachung von Software zusätzliche Kosten verursacht, denen keine sofortigen Einnahmen gegenüberstehen ([Jon94], [Tra95], 74, [Sam97], 16). Diese Kosten sind nicht vernachlässigbar, vielmehr liegen sie in der Regel in derselben Größenordnung wie die Kosten für die ursprüngliche Entwicklung (vgl. [Tra95], xi, [JGJ97], 23).

   (b) *Rechtliche Probleme*: Software, die sich zur Wiederverwendung eignen würde, darf nicht genutzt werden, weil dadurch Rechte verletzt würden. Dies ist z.B. der Fall, wenn ein Kunde das Recht an der für ihn entwickelten Software hat und deren Wiederverwendung in einem Projekt für einen anderen Kunden vereitelt (vgl. [CaE95], 6).

(c) *Mangelnde Motivation*: Der dritte Grund ist psychologischer Natur: die Motivation der Entwickler ist nicht ausreichend. Das kann daran liegen, daß die Entwickler dadurch abgeschreckt werden, daß sie Wartungs- und Weiterentwicklungsaufgaben übernehmen müssen ([CaE95], 5); bei viel genutzter wiederverwendbarer Software kann dies dazu führen, daß dies zur einzigen Aufgabe wird. Darüber hinaus wird als möglicher Grund angeführt, daß keine ausreichenden Anreize für die Entwicklung wiederverwendbarer Software vorhanden sind (vgl. [Sam97], 15; [CaE95], 5).

2. Es bestehen Mängel in der Wiederverwendbarkeit ([Sam97], 17). Diese können jedes der drei Merkmale der Wiederverwendbarkeit (siehe 2.1.2) betreffen:

   (a) *Nutzbarkeit für Entwickler*. Zunächst kann die Dokumentation mangelhaft sein, so daß die Verständlichkeit nicht gegeben ist (vgl. [Bro95], 224). Außerdem kann aufgrund von Qualitätsmängeln die Zuverlässigkeit in Frage gestellt sein ([Sam97], 17). Den Aspekt der mangelnden Verfügbarkeit behandeln wir als Hindernis für die Nutzung (s.u.).

   (b) *Adaptierbarkeit*. Verschiedene Faktoren erschweren die Sicherstellung ausreichender Adaptierbarkeit. In vielen Fällen fehlt eine adäquate Strukturierung im Hinblick auf angemessene Komponenten [BrS02]. Oft sind die verschiedenen Kontexte unbekannt, in denen die Software eingesetzt werden soll, zudem sind sie Änderungen unterworfen; außerdem können unterschiedliche Anforderungen kollidieren (vgl. [CaE95], 6f). Das kann zu implizit falschen Annahmen und damit zu "architectural mismatch" führen [GAO95].

   (c) *Portierbarkeit*. Das wesentliche Hindernis für Portierbarkeit ist die mangelnde Entkopplung der Anwendung von ihrer technischen Implementierung. Wir gehen darauf in 2.3.6 näher ein.

Hindernisse, deren Ursprung in der Nutzung liegt. Hindernisse, die die Nutzung behindern, können zwei Gründe haben: Probleme objektiver oder subjektiver Art.

1. *Objektive Probleme*. Zwei verschiedene Arten objektiv bestimmbarer Probleme können auftreten. Wenn wiederverwendbare Software vorhanden ist, aber nicht genutzt wird, kann das einerseits daran liegen, daß es Probleme bei der Suche gibt, d.h. daß sie nicht gefunden werden kann ([GAO95], [JGJ97], 8; [Sam97], 17). Es müssen also Maßnahmen durchgeführt werden, um die Verfügbarkeit sicherzustellen. Das eigentliche Hindernis besteht in dem Aufwand, der mit diesen Maßnahmen verbunden ist. Andererseits kann die Nutzung selbst problematisch sein. Das Hindernis ist in diesem Fall die mangelnde Wiederverwendbarkeit. Diese ist allerdings ein Hindernis, das bei der Herstellung entsteht; wir haben sie deswegen dort zugeordnet (s.o.).

2. *Subjektive Probleme*. Es können zweierlei Probleme auftreten, die im subjektiven Empfinden der Entwickler liegen. Zum einen kann aktiver Widerwillen gegen Wiederverwendung bestehen; als Ursache wird häufig der

sogenannte *Not-invented-here-Effekt* angeführt (vgl. z.B. [Shm99]; [Sam97], 15; [Tra95], 126; [McC97], 19). Er zeigt sich darin, daß Softwareentwickler Software grundsätzlich mißtrauisch gegenüberstehen, wenn sie sie nicht selbst entwickelt haben. Frakes und Fox zweifeln in [FrF95] die Bedeutung des Not-invented-here-Effekts an. Zum anderen kann es sein, daß die Entwickler Anreize für die Nutzung wiederverwendbarer Software erwarten, die nicht in ausreichendem Maß zur Verfügung stehen (vgl. [Tra95], 148).

In vielen Fällen lassen sich Kausalketten aufstellen. So kann z.B. der Mangel an wiederverwendbarer Software darin begründet sein, daß keine ausreichenden finanziellen Ressourcen für deren Entwicklung zur Verfügung stehen (s.o.); dies kann wiederum darin begründet sein, daß das Top-Management die Wiederverwendungsbemühungen nicht ausreichend unterstützt (dieser Punkt wird häufig als Hindernis angeführt, vgl. z.B. [Tra95], 148; [Sam97], 15; [Fra94], 17). Unser Ziel ist hier, die unmittelbaren Gründe zu identifizieren; wir machen daher an diesem Punkt keine weiterführenden Untersuchungen im Hinblick auf die zu Grunde liegende Kausalität.

2.3 Softwaretechnische Grundlagen

In dieser Arbeit bauen wir auf einer Reihe von softwaretechnischen Grundlagen auf, deren wichtigste wir in diesem Kapitel kurz zusammenfassen. Für viele Bereiche der Softwaretechnik sind die Begriffe bisher nicht vereinheitlicht (vgl. z.B. den Softwareentwicklungsprozeß). Ziel ist daher insbesondere, bei mehreren möglichen Definitionen oder Modellen die für diese Arbeit maßgebliche Alternative darzustellen.

2.3.1 Prozeßmodell

Untersuchungen im Bereich der Softwaretechnik liegt in der Regel explizit oder implizit ein Modell des typischen Softwareentwicklungsprozesses zugrunde. Dieses *Prozeßmodell* wird oft auch Projektmodell oder Vorgehensmodell genannt (vgl. [Pre97], 31; [Bal96], 54). Als Basis für die empirischen Untersuchungen benötigen wir ein Modell, auf das sich verschiedene Entwicklungsprozesse abbilden lassen. Daher definieren wir ein Referenzmodell, das von methodischen Details abstrahiert, insbesondere auch von denen einer bestimmten Art des objektorientierten Vorgehens, z.B. des Unified Software Development Process [BJR98]. Gemäß unserer Fokussierung auf betriebliche Informationssysteme wählen wir das in [ZME01] definierte sequentielle Modell als Referenzmodell (vgl. auch [Den91], 38ff und [Tau01].[23] Es ist in Tabelle 2-5 im Überblick dargestellt.

Es ist üblich, im Softwareentwicklungsprozeß ein iteratives Element vorzusehen, d.h. die Möglichkeit, innerhalb des sequentiellen Modells in frühere Pha-

[23] Eine oft übersehene ergiebige Quelle zum Prozeß ist [Kaf98] (vgl. [Stü00b];-).

sen zurückzuspringen bzw. zwischen den Phasen hin- und herzuspringen (vgl. [BJR98], 86ff). Dementsprechend ist es nicht zwingend, daß die Phasen streng sequentiell aufeinander folgen. In der Praxis kann es notwendig sein, eine Folgephase zu beginnen, bevor die vorhergehende abgeschlossen ist; das sequentielle Modell ist auch dann im Sinne der Projektsteuerung sinnvoll ([Tau01], vgl. auch [PaC85]). Wir orientieren uns deshalb nicht an der chronologischen Abfolge der Phasen, die wir daher als *logische Phasen* bezeichnen. Vorrangig sind für uns vielmehr die Endprodukte, die den verschiedenen Phasen zugeordnet werden können (siehe Tabelle 2-5). Diese Endprodukte sind Artefakte im Sinne der Definition in 2.1.3.

Den von der Vorstudie bis einschließlich zur Systemintegration reichenden Teil des Prozesses bezeichnen wir als *ursprüngliche Entwicklung*; ihr Endprodukt ist das Softwaresystem in seiner ersten Version. An sie schließen sich Wartung und Weiterentwicklung an, die parallel laufen und ohne scharfe Grenze ineinander übergehen. Sie können Elemente aus jeder der ursprünglichen Entwicklungsphasen beinhalten. Die Wartung dient dazu, das System bei unveränderter Funktionalität zu verbessern, insbesondere, indem Programmierungsfehler beseitigt werden. In der Weiterentwicklung wird die Funktionalität des Systems – oft auf Kundenwunsch – erweitert.

<table>
<tr><th colspan="3">Prozeßphase</th><th colspan="2">Endprodukt</th></tr>
<tr><td rowspan="5">Ursprüngliche Entwicklung</td><td colspan="2">Vorstudie</td><td colspan="2">Grobe Systembeschreibung</td></tr>
<tr><td colspan="2">Spezifikation</td><td colspan="2">Fachlicher Systementwurf</td></tr>
<tr><td colspan="2">Konstruktion</td><td colspan="2">Technischer Systementwurf</td></tr>
<tr><td colspan="2">Programmierung</td><td colspan="2">Getestete und dokumentierte Programme</td></tr>
<tr><td colspan="2">Systemintegration</td><td colspan="2">Im Kontext getestetes und dokumentiertes System</td></tr>
<tr><td colspan="2">Wartung</td><td>Weiter-entwicklung</td><td>Verbessertes System</td><td>System mit erweiterter Funktionalität</td></tr>
</table>

Tabelle 2-5: Logische Phasen und Endprodukte des Prozeßmodells

2.3.2 Softwarearchitektur

Die Architektur eines Softwaresystems spielt insbesondere bei wiederverwendbarer Software eine wichtige Rolle, weil durch sie die Wiederverwendbarkeit weitgehend festgelegt wird [Gri98, Coh99, Kar98]. Sie ist grundsätzlich ein abstrahiertes Modell dieses Systems (vgl. [BCK98], 23). Verschiedene Sichten auf dieses Modell sind möglich, die unterschiedliche Aspekte betonen; grundsätzlich läßt sich unterscheiden zwischen der logischen Sicht und der technischen Sicht. Die logische Sicht stellt die Funktionalität des Systems dar, während die technische Sicht deren softwaretechnische Realisierung beschreibt. Die technische Architektur läßt sich weiter untergliedern in Entwicklungssicht, Prozeßsicht und physische Sicht (vgl. [Kru95]).

Ein entscheidendes softwaretechnisches Problem bei der Entwicklung wiederverwendbarer Software ist die Zerlegung des Systems in kleinere Einheiten (vgl. [Bro95], 224; [ShG96], 16; [Kar98]). Dafür ist die Entwicklungssicht ausschlaggebend. Wenn wir im Folgenden von Architektur sprechen, so meinen wir deshalb die technische Architektur in der Entwicklungssicht. Wir definieren den Begriff wie folgt (vgl. [Den91], 11; [Pfl01], 200f; [BCK98], 23):

> *Softwarearchitektur ist die Struktur eines Softwaresystems. Sie besteht in seiner systematischen Zerlegung in abgrenzbare Einheiten und den Beziehungen zwischen diesen Einheiten.*

Je nach ihren Eigenschaften werden die Einheiten der Zerlegung unterschiedlich bezeichnet. Während lange Zeit der Begriff Modul gebräuchlich war [Par72, Par85, Den91], wird heute überwiegend von Komponenten gesprochen [Szy99, Sam97, MCD00, Hop00, HeS99, Grf98]; diese Konvention übernehmen wir. Wichtig ist in diesem Zusammenhang, daß wir Komponenten lediglich als Einheiten der Zerlegung betrachten – nicht im Kontext verteilter Verarbeitung und auch nicht im Kontext industrieller Produktion. In 2.3.3 setzen wir uns näher mit ihnen auseinander.

Verschiedene gängige Arten der Zerlegung werden als Architekturstile bezeichnet (vgl. [ShG96], 19ff). Einer dieser Stile ist die für betriebliche Informationssysteme meist gewählte Schichtenarchitektur (vgl. 2.5).

2.3.3 Komponenten und Schnittstellen

Wir verstehen Komponenten als Einheiten der Zerlegung von Softwaresystemen (s.o.). Der Begriff ist in der Literatur nicht einheitlich definiert (vgl. [Szy99, DSW98, Grf98, Sam97, Hop00, HeS99]). Trotzdem lassen sich folgende häufig genannte Merkmale identifizieren:

- Komponenten sind klar abgrenzbar: sie können unabhängig entwickelt und eingesetzt werden (vgl. [Szy99], 34; [DSW98], 387; [Hop00]; [HeS99], 7)
- Sie stellen Schnittstellen nach außen zur Verfügung und kapseln ihre Daten (vgl. [Szy99], 34; [DSW98], 387; [Hop00]; [HeS99], 7; [Grf98], 31; [Sam97], 2)
- Sie haben definierte Abhängigkeiten von ihrem Kontext (vgl. [Szy99], 34; [DSW98], 387)

Wesentlich für uns ist, daß die Definition Komponenten auf verschiedenen Granularitätsebenen zuläßt, so daß größere Komponenten aus kleineren aufgebaut werden können. Die Kompositionsfähigkeit von Komponenten wird in einigen gebräuchlichen Definitionen explizit als Merkmal angeführt ([Szy99], 30; [HeS99], 7). Objektorientierte Klassen haben dieses Merkmal nicht ([BrS02]), so daß Klassen mit Komponenten nicht gleichgesetzt werden sollten, was trotzdem häufig geschieht ([Szy99], 29).

Ganz bewußt wählen wir eine enge Definition, die den Komponentenbegriff nicht auf andere Artefakte des Softwareentwicklungsprozesses ausweitet, da diese sich in vielen für die Wiederverwendung wichtigen Merkmalen unter-

scheiden, z.B. hinsichtlich der Wiederverwendbarmachung. In der Literatur wird der Begriff hingegen auf alle Artefakte ausgedehnt, ohne jedoch das Vorgehensmodell daran anzupassen (vgl. [JGJ97], 85).

Aufbauend auf der o.g. Basisdefinition definieren wir drei Typen von Komponenten unterschiedlicher Granularität: Elementarbausteine, Baugruppen und Systeme (vgl. [StS00]). Wir unterscheiden sie anhand von drei Kriterien: Größe, Abhängigkeiten von anderen Komponenten und Ausführbarkeit.

Elementarbausteine sind der kleinste Komponententyp. Sie haben beliebige Abhängigkeiten von anderen Komponenten und sind nicht für sich ausführbar. Ein Beispiel sind objektorientierte Klassen.

Baugruppen sind Komponenten mittlerer Größe. Sie bestehen aus Elementarbausteinen und/oder anderen Baugruppen. Sie haben klar begrenzte Abhängigkeiten von anderen Komponenten und sind nicht für sich ausführbar. Die Kundenverwaltung eines gut strukturierten betrieblichen Informationssystems ist beispielsweise eine Baugruppe.

Systeme sind die größten Komponenten. Sie sind aus Baugruppen zusammengesetzt, haben minimale Abhängigkeiten von anderen Komponenten und sind für sich ausführbar. Ein Beispiel für ein System ist ein Datenbankmanagementsystem.

Unterschiedliche Middleware-Standards zur technischen Realisierung von verteilten Komponenten existieren, beispielsweise EJB [Mon01, ZBe00]. Eine allgemeingültige Definition sollte davon jedoch abstrahieren, so daß die technische Realisierung bewußt kein Bestandteil unserer Definition ist. In unserer Betrachtung spielt der Aspekt der physischen Verteilung von Komponenten keine Rolle.

Wiederverwendbarkeit spielt im Zusammenhang mit Komponenten eine wichtige Rolle. Bereits als der Begriff der Komponente von McIlroy aufgebracht wurde [McI68], stand Wiederverwendbarkeit im Zentrum.[24] Dies setzt sich fort: auch aktuell werden Komponenten teilweise explizit über ihre Wiederverwendbarkeit definiert (vgl. [JGJ97], 85; [Pre97], 13; [Amb98], 106).

Schnittstellen definieren wir als syntaktische Beschreibung der Funktionen, die eine implementierende Komponente zur Verfügung stellt (vgl. [GHJ94], 16). Wie diese Funktionen bereitgestellt werden, ist das Geheimnis der Implementierung, die deshalb austauschbar ist. Schnittstellen sind somit ein hervorragendes Mittel, um Komponenten zu entkoppeln, d.h. die gegenseitigen Abhängigkeiten von Implementierungsdetails zu reduzieren (vgl. [GHJ94], 18). Darüber hinaus ist es sinnvoll, die syntaktische zumindest um eine informelle semantische Beschreibung des Verhaltens zu ergänzen. In der Regel wird diese Aufgabe teilweise vom Namen der Schnittstelle sowie den Namen der Funktionen und der übergebenen Parameter übernommen. Zur Implementierung von Schnittstellen komplementär ist die Nutzung bereitgestellter Funktionen durch Aufruf von Schnittstellen. Jede Komponente kann beide Rollen wahrnehmen.

---

[24] Die von ihm vorgeschlagene industrielle Massenproduktion ist jedoch kein Gegenstand dieser Arbeit, weil wir uns auf unternehmensinterne Wiederverwendung von Komponenten beschränken (vgl. 1.3).

Ein und dieselbe Implementierung kann gleichzeitig mehrere Schnittstellen implementieren, die unterschiedlichen Funktionen entsprechen (dasselbe gilt natürlich sinngemäß für die Nutzung). Eine beispielhafte technische Realisierung von Schnittstellen in dem hier besprochenen Sinne stellt das Interface-Konzept der Programmiersprache Java dar ([ArG97], 20f), das allerdings nur auf der Ebene von Klassen funktioniert. Da die beschriebene Entkopplung von Komponenten die Variabilität im Sinne des Geheimnisprinzips (vgl. 2.3.4) erhöht, kann man sie zu einem *Prinzip der Konstruktion wiederverwendbarer Software* erklären ([GHJ94], 18):

> *Programmiere gegen eine Schnittstelle, nicht gegen eine Implementierung.*

Dieses Prinzip gilt nicht nur, wie von Gamma et al. in [GHJ94] postuliert, wenn objektorientierte Entwurfsmethoden verwendet werden, sondern allgemein [SiD00].

2.3.4 Geheimnisprinzip, Trennung der Zuständigkeiten

Ein wesentliches Qualitätskriterium für eine Architektur, insbesondere bei der Entwicklung wiederverwendbarer Software, ist die Frage, wie gut sie auf Änderungen vorbereitet ist. Dabei geht es darum, zu vermeiden, daß sich Änderungen, die eigentlich nur einen kleinen Teil des Systems betreffen, wie eine Welle im gesamten System fortpflanzen. Dieser Effekt wird im Englischen als *ripple effect* bezeichnet (vgl. [MCD00], 21).

Das von Parnas [Par72] formulierte *Geheimnisprinzip (engl.: information hiding)* dient dem Ziel, diesen Effekt zu vermeiden. Es ist ein Kriterium für die Zerlegung eines Softwaresystems, die in der Architektur des Systems resultiert. Parnas verwendet dabei den Begriff Modul für die Einheiten, aus denen das System aufgebaut ist. Das Geheimnisprinzip besagt, daß Module dadurch charakterisiert sind, daß sie eine Entwurfsentscheidung kennen, die sie vor allen anderen Modulen geheimhalten. Die jeweiligen Schnittstellen sollen so gewählt werden, daß sie möglichst wenig von der Implementierung des Moduls verraten.

Bei der Anwendung des Geheimnisprinzips geht es darum, Merkmale des Systems, die sich wahrscheinlich unabhängig voneinander ändern, in unterschiedlichen Modulen anzusiedeln, von denen sie geheimgehalten werden [PCW83]. Über die Schnittstellen zwischen Modulen sollen nur Merkmale bekanntgemacht werden, deren Änderung unwahrscheinlich ist. Dadurch wird erreicht, daß die von einer Änderung jeweils betroffene Anzahl von Modulen minimiert wird. Dadurch werden Änderungen erleichtert, die vielfach erforderlich sind: Zunächst treten sie bei Iterationen in der ursprünglichen Entwicklung und bei der Wartung auf. Daneben erfordern auch Anpassungen bei der Nutzung wiederverwendbarer Software teilweise Änderungen (vgl. 2.1.4).

Die Anwendung des Geheimnisprinzips führt außerdem zu einer *Trennung der Zuständigkeiten (engl.: separation of concerns)*[25], denn jedes Modul bekommt eine klar begrenzte Zuständigkeit zugewiesen und Überschneidungen werden vermieden [PCW83]. Trennung der Zuständigkeiten als Prinzip der Softwareentwicklung wurde u.a. von Dijkstra [Dij76] formuliert; verschiedene spezialisierte Interpretationen sind Gegenstand aktueller Forschung (vgl. [TOH99, Ald00, OsT01]). Die Trennung der Zuständigkeiten bewirkt durch die Entkopplung der Komponenten eine Reduzierung der Komplexität des Systems. Dies hat neben der Erleichterung von Änderungen weitere Vorteile: (1) Die Entwicklung kann parallel erfolgen, (2) das System ist besser zu verstehen, (3) das System ist leichter zu testen.

Objektorientierte Programmierung gilt als Mittel, Trennung der Zuständigkeiten zu erzielen; die in der Regel unterstützte Implementierungsvererbung[26] verletzt jedoch das Geheimnisprinzip, da Kinder einer Klasse Einschränkungen durch eine Schnittstelle umgehen können [Sny86, BrS02]. Dadurch kann der ripple effect hervorgerufen werden [But99].

Das Geheimnisprinzip ist eine wichtige Grundlage der Entwicklung wiederverwendbarer Software, da seine Anwendung auf zwei Arten die Wiederverwendbarkeit verbessert: Erleichterung von Änderungen bedeutet erhöhte Variabilität, Reduzierung der Komplexität entspricht verbesserter Nutzbarkeit für Entwickler (vgl. 2.1.2). Bei der Zerlegung des Systems in Komponenten (vgl. 2.3.2, 2.3.3) interessiert uns insbesondere, wie eine möglichst gute Trennung der Zuständigkeiten zwischen den Komponenten erreicht werden kann.

2.3.5 Variabilität: Mechanismen und Analyse

Bei der Konstruktion wiederverwendbarer Software ist Variabilität ein wichtiges Ziel (vgl. 2.1.2; [vGu00]). Zu ihrer technischen Realisierung werden sog. Variabilitätsmechanismen eingesetzt; die wichtigsten davon sind (vgl. [SiD00]; [JGJ97], 100f):

- Austausch von Schnittstellenimplementierungen (vgl. 2.3.3)
- Parametrierung
- Vererbung
- Generierung

Neben der Art des Mechanismus spielt auch der Zeitpunkt eine Rolle, zu dem Variabilität umgesetzt werden kann; er ist mit dem Aufwand verknüpft. Am aufwendigsten ist Variabiliät, die Änderungen am Quellcode erfordert und somit zum Zeitpunkt der Übersetzung durch den Compiler wirksam wird. Weniger aufwendig ist es, wenn fertig übersetzte Komponenten ausgetauscht wer-

---

[25] Die Begriffe werden oft synonym verwendet [Par94]. Zur Übersetzung des Begriffs ins Deutsche vgl. [SiD00, BSi00]; gemeint ist eine Trennung der Zuständigkeiten *für bestimmte Aufgaben*; eine mögliche alternative Übersetzung wäre *Aufgabentrennung*.

[26] Im Gegensatz zu Verhaltensvererbung über Schnittstellen [BrS02] (engl. auch: declaration inheritance [Har01])

den, so daß die Wirksamkeit mit dem Binden eintritt. Das Binden schließlich kann statisch, d.h. vor dem Programmstart, oder dynamisch, d.h. während der Laufzeit des Programms, erfolgen.

Bei der Konstruktion spielt die Variabilitätsanalyse eine wichtige Rolle ([SiD00]; vgl. auch [CHW98]). Sie hilft dabei, die Architektur gezielt auf Variationen in den Anforderungen vorzubereiten, indem diese priorisiert werden und geprüft wird, inwiefern die Architektur daran angepaßt werden kann. Ihr Ergebnis ist gleichzeitig ein Maß für die Qualität der untersuchten Architektur [SiD00].

Das Problem der optimalen Realisierung von Variabilität ist in dieser Arbeit von großer Bedeutung. In den Fallstudien in Kapitel 5 untersuchen wir die Architektur wiederverwendbarer Systeme eingehend auf die verwendeten Mechanismen. Ein umfassendes Konzept der Variabilität wird in 7.3.3 vorgestellt.

2.3.6 Software-Kategorien

Software-Kategorien [SiD00] dienen der Trennung der Zuständigkeiten. Sie helfen also, Software gezielt auf Änderungen vorzubereiten und spielen insofern für die Entwicklung wiederverwendbarer Software eine entscheidende Rolle (vgl. 2.3.4). Wir fassen das Konzept hier kurz zusammen.

Die verschiedenen Kategorien werden danach unterschieden, wodurch das jeweilige Stück Software – bevorzugt, aber nicht notwendigerweise, eine Komponente – bestimmt ist. Grundsätzlich kann Software bestimmt sein von der fachlichen Anwendung (kurz: der Anwendung) und von den technischen Schnittstellen[27] (kurz: der Technik). Daraus ergeben sich folgende Kategorien mit unterschiedlichen Bedingungen für ihre Wiederverwendbarkeit (zu ihrer graphischen Darstellung siehe Abbildung 2-19):

- *0-Software* (Null-Software) ist weder von der Anwendung noch von der Technik bestimmt, also neutral. Sie ist wegen des hohen Abstraktionsgrades sehr gut wiederverwendbar, macht aber zwangsläufig nur einen Teil des Systems aus, da sowohl Anwendungsfunktionalität als auch deren technische Umsetzung für betriebliche Informationssysteme immer benötigt werden. Beispiele sind Klassenbibliotheken für abstrakte Konzepte wie die Standard Template Library STL für die Programmiersprache C++ oder die Java Generic Library JGL für die Programmiersprache Java.
- *A-Software* ist allein von der Anwendung bestimmt. Sie kann dann wiederverwendet werden, wenn gleiche oder ähnliche Anwendungslogik ganz oder teilweise benötigt wird. Es gibt zwei grundsätzliche Möglichkeiten, dies technisch umzusetzen: statische Bindung und dynamische Bindung. Letztere kann entweder lokal oder per Fernaufruf (z.B. über Middleware-Standards wie COM, CORBA oder EJB) realisiert werden.

[27] Oft als APIs (Abkürzung für Application Programming Interfaces) bezeichnet

- *T-Software* ist allein von der Technik der vom jeweiligen System genutzten Schnittstellen (APIs) zu Systemumgebung[28] und Nachbarsystemen[28] bestimmt. Sie kann wiederverwendet werden, wenn dieselbe technische Schnittstelle in einem anderen Kontext verwendet wird. Einige gebräuchliche Schnittstellen sind beispielsweise: Java Database Connectivity (JDBC), Open Database Connectivity (ODBC) für Datenbanken; bei graphischen Benutzeroberflächen die entsprechenden Teile der Microsoft Foundation Classes (MFC) für C++ und Swing für Java.
- *AT-Software* ist sowohl von der Technik als auch von der Anwendung bestimmt. Sie kann kaum wiederverwendet werden, da sie immer sowohl technisch als auch fachlich an neue Kontexte angepaßt werden muß. AT-Software verletzt insofern das Geheimnisprinzip (siehe 2.3.4), als sie von unterschiedlichen Änderungen beeinflußt wird und ist daher zu vermeiden.

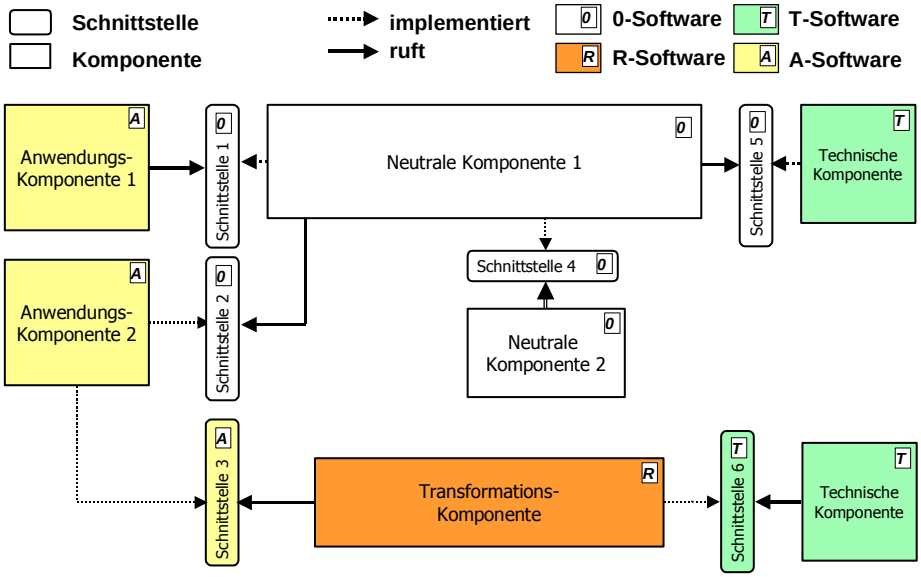

Abbildung 2-19: Graphische Notation für Komponenten und deren Kategorien (Beispiel)

Wenn man Anwendung und Technik trennt, muß zwischen fachlichen Komponenten (A-Software) und ihren technischen Repräsentationen (T-Software) transformiert werden. Zu diesem Zweck benötigt man *R-Software* (R steht für Repräsentation). Sie ist eine milde Art von AT-Software, die stereotyp ist und deshalb in der Regel mit Hilfe von Metainformation generiert werden kann. Ein Beispiel für den Einsatz von R-Software ist in Abbildung 5-2 (S. 122) dargestellt.

Bei der Komposition von Komponenten ergibt sich eine resultierende Kategorie für das Kompositum. Die Regeln für die Komposition sind ähnlich wie bei Blutgruppen: Innerhalb derselben Kategorie X kann beliebig kombiniert wer-

[28] zur Definition siehe 2.1.2

den, wobei die resultierende Kategorie wieder X ist. Werden 0-Software-Komponenten mit Software einer beliebigen Kategorie Y kombiniert, so hat das Kompositum auch die Kategorie Y. Komposition von A-Software und T-Software erzeugt als resultierende Kategorie AT-Software. Werden AT-Software-Komponenten mit Software einer beliebigen Kategorie kombiniert, so ist die resultierende Kategorie immer AT-Software.

Ein wichtiges allgemeines Ziel bei der Konstruktion ist, die Kategorien zu trennen. Darüber hinaus sollte AT-Software völlig vermieden und T-Software auf ein Minimum reduziert werden. Durch dieses Ziel ist gleichzeitig ein Qualitätsmaß für Architekturen gegeben.

Eine adäquate graphische Darstellungsform für Komponenten und deren Kategorien besteht nicht, insbesondere nicht in der Unified Modeling Language UML [BrS02]. Daher verwenden wir in dieser Arbeit eine eigene graphische Notation für die Darstellung von Software-Architekturen, die es ermöglicht, Komponenten und deren Software-Kategorien (siehe 2.3.6) darzustellen. Sie ist beispielhaft in Abbildung 2-19 dargestellt.

Die Software-Kategorien sind eine wichtige Grundlage dieser Arbeit, insbesondere für die in Kapitel 5 und Kapitel 6 durchgeführten Untersuchungen. In Kapitel 5 untersuchen wir anhand von Fallstudien, welche spezifischen Implikationen die Software-Kategorien und die hier zusammengefaßten Regeln für die Entwicklung wiederverwendbarer Software, insbesondere für die Architektur, haben.

2.3.7 Softwaretechnische Besonderheiten der Entwicklung wiederverwendbarer Software

Bei der Entwicklung wiederverwendbarer Software treten einige Besonderheiten verglichen mit der Entwicklung nicht wiederverwendbarer Software auf, die wir in Tabelle 2-6 zusammengefaßt darstellen. Dabei orientieren wir uns an den Stufen des in 2.3.1 definierten Softwareentwicklungsprozesses. Die Besonderheiten unterscheiden sich je nach Variante der Herstellung; wir haben sie daher für a priori und a posteriori wiederverwendbare Software getrennt aufgeführt.

Variante der Herstellung	Besonderheiten gegenüber Entwicklung nicht wiederverwendbarer Software
A priori wiederverwend-bare Software	• Spezifikation: Zusätzliche Personen stellen Anforderungen, dadurch in der Regel mehr Anforderungen • Konstruktion: grundsätzlich ist die Zahl der erforderlichen Iterationen höher, da mehr Personen am Prozeß beteiligt sind • Programmierung: höherer Aufwand für Codierung, Test und Dokumentation • Wartung wesentlich aufwendiger: Höhere Anzahl und größere Vielfalt der Anfragen, spezieller Prozeß notwendig • Systemintegration: höherer Testaufwand
A posteriori wiederverwend-bare Software	• Spezifikation: Zusatzaufwand für Spezifikation zusätzlicher Funktionalität • Konstruktion: Zusatzaufwand für Modifikation des bestehenden technischen Systementwurfs, insbesondere für zusätzliche Variabilitätsmechanismen • Programmierung – Zusatzaufwand für Modifikationen und Neuerstellung einiger Teile – Testen und Dokumentation insgesamt aufwendiger • Wartung wesentlich aufwendiger: Höhere Anzahl und größere Vielfalt der Anfragen, spezieller Prozeß notwendig • Systemintegration: höherer Testaufwand

Tabelle 2-6: Softwaretechnische Besonderheiten der Entwicklung wiederverwendbarer Software

2.4 Grundlagen der Softwareökonomie

Die Softwareökonomie beschäftigt sich mit der Anwendung allgemeiner ökonomischer Methoden auf die Softwareentwicklung und Anpassung an deren spezifische Aspekte.

Einen bis heute gültigen Grundstein legte Boehm [Boe81, Boe84]; das von ihm entwickelte COCOMO-Modell (COnstructive COst MOdel) setzte einen bis heute gültigen Standard für die Schätzung von Kosten bei der Software-Entwicklung, der auch bei der Bewertung wiederverwendbarer Software Anwendung findet (vgl. [Kar95], 163f; [JGJ97], 386).

Der wesentliche Vorteil einer ökonomischen im Gegensatz zu einer rein technischen Beurteilung ist, daß sie eine balancierte Betrachtung ermöglicht, indem sie alle Einflußfaktoren berücksichtigt. Entsprechend werden bei der Entscheidung zwischen mehreren konkurrierenden Lösungen nicht nur die technischen Aspekte berücksichtigt, so daß die beste Lösung im Rahmen der zur Verfügung

stehenden Ressourcen gefunden werden kann [Boe84]. Darüber hinaus werden Methoden für das Auffinden dieser Lösung bereitgestellt.

2.4.1 Definition der Softwareökonomie

Wirtschaft beschäftigt sich allgemein damit, menschliche Bedürfnisse angesichts knapper dafür verfügbarer Mittel optimal zu befriedigen. Dabei wird das ökonomische Prinzip zugrundegelegt; in Form des Minimalprinzips formuliert besagt es, daß ein gegebener Güterertrag mit geringstmöglichem Einsatz von Produktionsfaktoren zu erwirtschaften ist (vgl. [Wöh96], 1). Diese Aufgabe stellt sich insbesondere bei der Softwareentwicklung, da die Nachfrage nach dem knappen Gut Software schneller anwächst als das Angebot (vgl. [Lim98], 1ff); der wesentliche Produktionsfaktor ist in diesem Fall die Arbeitszeit der Entwickler.

Softwareökonomie definieren wir auf dieser Basis – in Anlehnung an ([Wöh96], 2) – wie folgt:

> *Softwareökonomie umfaßt alle planvollen menschlichen Tätigkeiten, die unter Beachtung des ökonomischen Prinzips mit dem Zweck erfolgen, die bestehende Knappheit an Software zu verringern.*

Grundsätzlich unterscheidet man zwischen Makroökonomie, die das Handeln auf Ebene der Volkswirtschaft eines ganzen Landes untersucht, und Mikroökonomie, die sich mit dem Handeln auf der Ebene des Einzelnen oder einer Organisation (z.B. eines Unternehmens) beschäftigt [Boe84]. Diese Arbeit konzentriert sich auf mikroökonomische Aspekte der Softwareentwicklung.

2.4.2 Wiederverwendbarmachung als Investition

Ökonomisch gesehen ist die Wiederverwendbarmachung von Software eine Investition [BBo91, WiB98]. Dabei legen wir diese Definition zugrunde:

> *Eine Investition ist die Bindung von finanziellen Mitteln für einen Vermögensgegenstand zum Zweck der Erzielung von Einnahmen aus diesem Vermögensgegenstand zu späteren Zeitpunkten. [www3]*

Im Folgenden arbeiten wir genau heraus, worin im Kontext der Wiederverwendung die Investition und die erzielten Einnahmen bestehen.

Wie in 2.1.2 ausgeführt, sind die Kosten der Entwicklung systematisch wiederverwendbarer Software in der Regel höher als bei der Entwicklung nicht systematisch wiederverwendbarer Software. Wiederverwendbarmachung führt also zu zusätzlichen Kosten. Die zur Deckung dieser zusätzlichen Kosten bereitgestellten Geldmittel sind die *Investition in Wiederverwendbarkeit* ([BBo91], vgl. auch [McC97], 7f). Es sind größtenteils Personalkosten (vgl. [Kar95], 9), die normalerweise in Form von Opportunitätskosten entstehen: das Personal könnte in derselben Zeit in anderen Entwicklungsprojekten arbeiten, die direkt oder indirekt Einnahmen erwirtschaften (vgl. [Fri94], 422).

Wir wenden uns nun den Einnahmen aus der Investition in Wiederverwendbarkeit zu. Mit wiederverwendbarer Software, die für die Nutzung in der internen Entwicklung vorgesehen ist, können mittelbar Einnahmen im Sinne einer Umsatzsteigerung erzielt werden, beispielsweise durch verkürzte Entwicklungsdauer ('time to market') oder erhöhte Kundenzufriedenheit; diese Einnahmen sind jedoch schwer zu quantifizieren (vgl. 2.2.3). Unmittelbar werden keine Einnahmen im Sinne eines Umsatzes mit Kunden erzielt, sondern der Aufwand für Entwicklung und Wartung gesenkt, was – ceteris paribus – einer Senkung der Kosten entspricht. Wir zeigen im Folgenden, daß das effektiv einer Erhöhung der Einnahmen entspricht.

Ökonomisches Ziel jedes Unternehmens ist allgemein formuliert die Maximierung des Gewinns[29] G, der sich aus der Differenz zwischen Einnahmen E und Kosten K berechnet:

$$G = E - K$$

Die Verringerung der Kosten K um einen bestimmten Betrag ΔK bewirkt eine Erhöhung des Gewinns G um ΔG, die in ihrer Auswirkung genau einer Erhöhung der Einnahmen E um denselben Betrag ΔK entspricht. Dies zeigt die folgende einfache algebraische Umformung:

$$\Delta G = E - (K - \Delta K) = (E + \Delta K) - K$$

Übertragen auf die ökonomische Analyse der Wiederverwendung heißt das: die beim m-ten Nutzungsereignis eines Stücks Software erzielte Erhöhung ΔG_m des Gewinns ist eine Einnahme aus der Investition in die Wiederverwendbarmachung der Software im Sinne der oben genannten Definition einer Investition.

Die Einnahmen aus der Nutzung wiederverwendbarer Software werden in der Regel nicht zu einem einzigen Zeitpunkt, sondern zu mehreren über einen längeren Zeitraum verteilten Zeitpunkten – den Nutzungszeitpunkten – realisiert (vgl. [Kar95], 176f). Deshalb sollten die Kosten für die Wiederverwendbarmachung von Software als längerfristige Investition behandelt werden, die mit Hilfe von Techniken für Investitionsentscheidungen bewertet werden (vgl. [Lim98], 136f). Diese Techniken stellen wir im nächsten Abschnitt vor.

2.4.3 Methoden zur Analyse von Investitionen

Da wir die Kosten der Wiederverwendbarmachung als Investition betrachten, gehen wir nun auf Methoden zur Analyse dieser Investition ein. Das Ziel der Analyse ist, eine ökonomische Grundlage für die Entscheidung über die Investition bereitzustellen. Grundsätzlich geht es dabei darum, eine Kosten-Nutzen-Analyse durchzuführen, also Kosten und Nutzen finanziell zu quantifizieren und gegeneinander abzuwägen [WiB98].

---

[29] Wir betrachten hier den operativen Gewinn, nicht den sich nach steuerlichen Effekten und bilanztechnischen Manipulationen ergebenden Bilanzgewinn. Dabei gehen wir vom Erfolgsfall aus, in dem ein Gewinn erzielt wird. Will man den allgemeinen Fall betrachten, so muß man den Begriff des *Ergebnisses* verwenden. Das Ergebnis kann sowohl die Form eines Gewinns als auch eines Verlustes annehmen.

Für die Quantifizierung der Kosten gibt es eine Reihe von Schätzmethoden, unter anderem die oben erwähnte COCOMO-Methode [Boe81] und ihre Weiterentwicklungen sowie die Function Point-Methode. In der Praxis spielt die Erfahrung der Entwickler eine wesentliche Rolle. Aus diesen Gründen beschäftigen wir uns mit dem Problem der Schätzung der Kosten nicht, sondern gehen davon aus, daß eine Schätzung vorliegt. Durch diese Kostenschätzung wird auch der unmittelbar finanziell wirksame Nutzen quantifiziert, der durch reduzierten Aufwand entsteht (vgl. 2.2.3, 2.4.2). Die Quantifizierung des nur mittelbar finanziell wirksamen Nutzens ist wesentlich schwieriger und nur für den Einzelfall zu leisten (vgl. 2.2.3).

Liegen die Schätzungen vor, so sind Kosten und Nutzen abzuwägen. Die dafür verwendete Analysemethode muß zwei Kriterien erfüllen: (1) sie muß es ermöglichen, verschiedene Investitionen zu vergleichen und (2) sie muß eine Regel beinhalten, nach der entschieden werden kann, ob eine Investition akzeptabel ist oder nicht [Her68]. Verschiedene Methoden stehen zur Auswahl, unter anderem Amortisationsperiode, Durchschnittsrendite und die Kapitalwertmethode, die den Zeitwert der Geldflüsse durch Abzinsung berücksichtigen.

Die Investition in wiederverwendbare Software ist dadurch gekennzeichnet, daß die Einnahmen zu verschiedenen Zeitpunkten stattfinden (vgl. 2.4.2); daneben werden auch die Wartungskosten zu verschiedenen Zeitpunkten wirksam (vgl. [Kar95], 176f). Um die Vergleichbarkeit zu gewährleisten, muß also der Zeitwert des Geldes berücksichtigt werden. Dazu wird in der Regel die Kapitalwertmethode herangezogen ([Boe84]; [Her68]; [Lim98], 136f; [BrM95], 12ff), bei der als Vergleichsgröße der Kapitalwert (engl.: Net Present Value) oder der interne Zinssatz (engl.: Internal Rate of Return) ermittelt werden (vgl. [Wöh96], 757ff).

Daneben sind die zugrundeliegenden Schätzungen in der Regel mit Unsicherheit behaftet (vgl. [Boe84, PoC93]), so daß ein Investitionsrisiko besteht [BBo91]. Die Entscheidungsregel muß diese Unsicherheit der Investition berücksichtigen. Computersimulationen sind ein geeignetes Mittel zur Analyse dieser Unsicherheit [Her68].

2.4.4 Grundlagen der Kapitalwertmethode

Im Folgenden fassen wir wesentliche Grundlagen der Kapitalwertmethode für die Bewertung von Investitionen zusammen.

Der Zeitwert des Geldes: Berechnung des Barwerts

Die Bewertung eines Geldflusses orientiert sich daran, zu welchem Zeitpunkt er geschieht. Die Begründung dafür ist, daß Geld umso mehr Zinsen erwirtschaftet, je früher es zur Verfügung steht ([BrM95], 12)[30]. Um Geldflüsse zu verschiedenen Zeitpunkten vergleichbar zu machen, berechnet man ihren Bar-

[30] Dies entspricht dem ersten grundlegenden Prinzip: "a dollar today is worth more than a dollar tomorrow" ([BrM95], 12).

wert (engl.: Present Value), der die Zinsverluste bei Geldflüssen in der Zukunft berücksichtigt.

Zum Zeitpunkt $t = 0$ beträgt der Barwert B eines Geldflusses C_0, der in der Zukunft zum Zeitpunkt $t = t_0$ erwartet wird (vgl. [Wöh96], 757):

$$B = \frac{C_0}{(1+z)^{t_0}}$$

Dabei ist z der jährliche Zinssatz, der mit dem Geldbetrag erwirtschaftet werden könnte; er entspricht den Kosten für das Kapital. Die Zeit t wird in Jahren gemessen. Dieses Verfahren entspricht einer umgekehrten Zinseszinsrechnung, bei der der Faktor für die Verzinsung in Form einer geometrischen Reihe ausgedrückt wird. Es wird Abzinsung oder Diskontierung genannt. Entsprechend nennen wir z *Diskontierungszinssatz*. Der Faktor

$$f_D = \frac{1}{(1+z)^{t_0}}$$

wird *Diskontierungsfaktor* genannt.

Kapitalwert einer Investition

Bei der Bewertung einer Investition muß entschieden werden, ob eine ausreichend hohe Rendite für die Investitionssumme erwirtschaftet wird. In vielen Fällen wird zum Zeitpunkt $t = 0$ die Investitionssumme I_0 investiert, d.h. es fließt Geld ab. Durch die Investition werden in der Folge Geldeinnahmen G_i zu den Zeitpunkten $t = t_i$ erwirtschaftet. Diese müssen diskontiert werden. Der Kapitalwert K (engl.: Net Present Value, NPV) einer Investition berechnet sich aus der Summe von Anfangsinvestition und den Barwerten der Einnahmen wie folgt (vgl. [BrM95], 30; [Wöh96], 757ff):

$$K = -I_0 + \sum_i \frac{G_i}{(1+z)^{t_i}}$$

Zur Entscheidung darüber, ob die Investition getätigt werden soll, wird die NPV-Regel herangezogen: Man soll investieren, wenn der Kapitalwert größer als Null ist ([BrM95], 14). Das Ergebnis hängt natürlich wesentlich vom Diskontierungszinssatz z ab, der für die jeweilige Investition anzusetzen ist und ein sehr wichtiger Parameter ist. Er muß umso höher gewählt werden, je größer das mit der Investition verbundene Risiko ist ([BrM95], 13, 161f)[31].

Wir betrachten die Wiederverwendbarmachung von Software als längerfristige Investition (vgl. 2.4.2). Kapitalwert-basierte Bewertungsmethoden eignen sich daher gut für die ökonomische Analyse der Entwicklung wiederverwendbarer Software (vgl. [Lim98], 136f). Auch aus praktischen Gründen sind sie eine gute Wahl: Bei der Bestimmung des Kapitalwerts einer Investition werden nur tatsächliche Mittelflüsse zur Anrechnung gebracht; deren buchhalterische Behandlung spielt keine Rolle ([BrM95], 96ff). Insbesondere wird immer mit dem

[31] Das wird als zweites grundlegendes Prinzip formuliert: "a safe dollar is worth more than a risky one" ([BrM95], 13).

Zeitpunkt gerechnet, zu dem der Geldfluß tatsächlich stattfindet. Dadurch wird die Berechnung gegenüber Methoden vereinfacht, die mit Größen der Buchhaltung arbeiten, z.B. die Economic Value Added (EVA)-Methode (vgl. [www2]).

Rendite, interner Zinssatz

Die Rendite *R* einer Investition *I* wird oft als Meßgröße für deren Profitabilität herangezogen (vgl. [BrM95], 271). Im einfachsten Fall ergibt sie sich aus einer einzigen Einnahme *E*, die durch diese Investition erwirtschaftet wird. Sie kann dann berechnet werden als das Verhältnis des Gewinns *G* aus dieser Investition zur Investition (vgl. [BrM95], 14):

$$R = \frac{G}{I} = \frac{E - I}{I}$$

Das Ergebnis dieser Berechnung ist ein Zinssatz für den Zeitraum zwischen Mittelabfluß für die Investition und Mittelzufluß durch die Einnahme. Er kann auf die übliche Größe eines Zinssatzes pro Jahr umgerechnet werden.

Wenn die Mittelab- und -zuflüsse über die Zeit verteilt sind, so muß die Rendite in Form des internen Zinssatzes (engl.: Internal Rate of Return) berechnet werden [PoC93]. Er ist definiert als derjenige Diskontierungszinssatz, bei dem der Kapitalwert Null wird ([BrM95], 80). Der interne Zinssatz wird im Finanzwesen häufig als Meßgröße genutzt und gilt als praktisch und gut handhabbar; Schwierigkeiten können entstehen, wenn der Kapitalwert – aufgetragen als Funktion des Diskontierungszinssatzes – mehrere Nullstellen hat (vgl. ([BrM95], 80ff). In diesem Fall ist eine genaue Untersuchung des Kurvenverlaufs notwendig.

Aus zwei Gründen ist es wünschenswert, statt des Kapitalwerts den internen Zinssatz als Meßgröße für die ökonomische Analyse von Softwareentwicklungsprojekten zu verwenden:

- Der interne Zinssatz ist für viele Personen intuitiv besser verständlich als der Kapitalwert (vgl. [HaV99], 221f; [Bra01]) und damit auch besser vermittelbar. Dies gilt insbesondere für technische Manager in der Softwareentwicklung, die in der Regel keine finanzwissenschaftliche Ausbildung haben.
- Der interne Zinssatz erlaubt den Vergleich zwischen verschiedenen alternativen Projekten, auch wenn die Höhe der Investition unterschiedlich ist. Der Kapitalwert hingegen ist eine absolute Größe, was den Vergleich erschwert.

Risiko und Entscheidungen unter Unsicherheit

Entscheidungen über Softwareentwicklungsprojekte sind in der Regel mit großer Unsicherheit behaftet (vgl. [Boe84, PoC93]), da der Ausgang verschiedener Optionen nicht mit Sicherheit vorherzusagen ist. Die wesentliche Unsicherheit bei der Bewertung wiederverwendbarer Software besteht in der Vorhersage, wann und wie oft sie genutzt werden wird. Das Risiko, daß das erwünschte Ereignis (in unserem Fall die Nutzung wiederverwendbarer Software) als Ausgang nicht eintritt, muß in der ökonomischen Bewertung berücksichtigt werden.

Dafür gibt es eine Reihe von grundlegenden Methoden (vgl. [Boe84]):

1. Methoden zur Entscheidungsfindung unter kompletter Unsicherheit, wie die Maximax-Regel, die Maximin-Regel oder die Laplace-Regel. Sie sind generell nicht für Entscheidungen im Software Engineering geeignet [Boe84].
2. Ermittlung des Erwartungswerts für das Ergebnis. Dazu muß die Wahrscheinlichkeit jeder Ausgangsmöglichkeit geschätzt und der entsprechende Geldfluß damit gewichtet werden. Aus der Gesamtheit der Ausgänge kann der Erwartungswert als arithmetischer Durchschnitt berechnet werden.
3. Methoden, die die Unsicherheit reduzieren, indem Information gekauft wird. Dies kann zum Beispiel geschehen, indem man die Mittel dafür zur Verfügung stellt, einen Prototypen zu bauen. Die Unsicherheit bezüglich der Frage, ob und wie oft wiederverwendbare Software in Zukunft genutzt werden wird, läßt sich dadurch jedoch nicht reduzieren.

Bei der Bewertung wiederverwendbarer Software muß also mit Erwartungswerten gearbeitet werden, da dies die einzige der drei Methoden ist, die anwendbar ist. Dadurch wird jedoch nur der Tatsache Rechnung getragen, daß keine sichere Information über die Nutzung zur Verfügung steht. Die Höhe der Unsicherheit wird dabei jedoch nicht bewertet.

Daher ist es notwendig, eine Meßgröße für die Unsicherheit zu finden, mit der eine Investition behaftet ist. In der Finanzwissenschaft ist es üblich, die Streuung (in Form von Standardabweichung oder Varianz) der möglichen Ergebnisse als Meßgröße für das Risiko einer Investition zu verwenden ([Her68]; [BrM95], 131ff): je größer die Streuung, desto größer ist das mit der Investition verbundene Risiko. Wir wenden diese Methode bei der in Kapitel 6 beschriebenen Modellierung an.

Um eine Investition abgewogen zu beurteilen, muß sowohl deren erwartete Rendite als auch die damit verbundene Unsicherheit in Betracht gezogen werden, da eine Erhöhung der Rendite oft mit höherer Unsicherheit verbunden ist. Bei einer Investitionsentscheidung ist es daher nicht sinnvoll, die Rendite allein zu maximieren; statt dessen muß die Kombination aus Rendite und Unsicherheit optimiert werden [Her68].

Die übliche Methode, die Unsicherheit bei der Bewertung von Investitionen zu berücksichtigen, ist die Anwendung des Capital Asset Pricing Model ([BrM95], 161f). Es sieht vor, daß die Unsicherheit in die Berechnung des Diskontierungszinssatzes einfließt; dazu muß das Standardmaß für Risiko bei Geldanlagen β ermittelt werden. Das wirft zwei grundlegende Probleme auf, die besonders im Zusammenhang mit der Bewertung von Investitionen in die Entwicklung wiederverwendbarer Software auftreten:

- Da β mit Hilfe von Kapitalmarktdaten und aufgrund der Kapitalstruktur bestimmt wird ([BrM95], 183), ist es in der Regel nur für große, börsennotierte Unternehmen verfügbar, die entsprechend spezialisierte Fachleute beschäftigen. Nur wenige Unternehmen in der Softwareindustrie zählen dazu. Die Ermittlung für einzelne Unternehmensteile und kleinere, insbesondere nicht börssennotierte, Unternehmen ist äußerst schwierig (vgl. [CKM94],

330ff). Dieses Problem betrifft insbesondere EDV-Abteilungen von Unternehmen mit anderem Kerngeschäft, beispielsweise von Banken.

- Für jedes Projekt muß das jeweils spezifische β ermittelt werden ([BrM95], 181f)[32]. Voraussetzung dafür ist zunächst, daß das β des Unternehmens verfügbar ist, was mit den oben beschriebenen Schwierigkeiten verbunden ist. Auf dieser Basis schließlich muß das projektspezifische β geschätzt werden, was einschlägige Erfahrung verlangt; für das Vorgehen bei der Schätzung gibt es nur Anhaltspunkte ([BrM95], 205).

2.4.5 Ökonomische Besonderheiten der Entwicklung wiederverwendbarer Software

Bei der ökonomischen Analyse von Projekten zur Entwicklung wiederverwendbarer Software ergeben sich einige Besonderheiten, die wir in Tabelle 2-7 zusammengefaßt haben. Das grundlegende Vorgehen entspricht dem in 2.4.3 beschriebenen.

Gegenstand der Analyse	Besonderheit
Kosten	• Höhere Kosten in verschiedenen Phasen des Entwicklungsprozesses für – Adaptierbarkeit: alle Projektphasen – Nutzbarkeit für Entwickler: Programmierung (Testen, Dokumentation), Wartung – Portierbarkeit: Programmierung (Codierung, Testen), Wartung • Zusätzliche Kosten für Bereitstellung
Umsatz	• Umsatz setzt sich aus zwei Komponenten zusammen: – Unmittelbar: Verringerung der Kosten in Projekten, die die Software nutzen – Mittelbar: Erhöhung des Umsatzes (schwer zu quantifizieren) • Zu mehreren Zeitpunkten (i.d.R. mehrere über einen längeren Zeitraum verteilt) auftretende Nutzungsereignisse • Können nur mit Unsicherheit geschätzt werden, da keine sicheren Vorhersagen über Nutzung möglich sind

Tabelle 2-7: Ökonomische Besonderheiten der Entwicklung wiederverwendbarer Software

[32] In einem der Standardwerke zur Finanzierungstheorie wird dies am Beispiel der Firma DEC in drastischer Form so erläutert: "The true cost of capital depends on the use to which the capital is put. [...] It is clearly silly to suggest that DEC should demand the same rate of return from a very safe project as from a very risky one." ([BrM95], 182)

2.5 Betriebliche Informationssysteme

Der Schwerpunkt dieser Arbeit liegt auf der Entwicklung wiederverwendbarer Software für betriebliche Informationssysteme (vgl. 1.3). Die Anforderungen an die Software und die Schwierigkeiten bei der Entwicklung unterscheiden sich von denen anderer Arten von Softwaresystemen, weshalb auch die Bedingungen für Wiederverwendung unterschiedlich sind. Wir behandeln daher im Folgenden nach einer Begriffsbestimmung wesentliche Merkmale betrieblicher Informationssysteme, insbesondere die Schichtenarchitektur, und Besonderheiten und typische Schwierigkeiten bei ihrer Entwicklung.

2.5.1 Definition und Einordnung in den Softwaremarkt

Zunächst definieren wir, was wir unter betrieblichen Informationssytemen verstehen. Dabei greifen wir auf die Definition von Denert zurück:

> *Betriebliche Informationssysteme unterstützen und steuern die Abläufe in allen Bereichen eines Unternehmens: in Marketing und Vertrieb, in Produktion und Logistik ebenso wie im Rechnungswesen. Sie ermöglichen oder erleichtern die Arbeit der Sachbearbeiter, sie helfen dem mittleren Management bei seinen Dispositionen, und sie liefern der Unternehmensleitung Daten für ihre täglichen Entscheidungen und strategischen Planungen. ([Den91], 15)*

Im Folgenden wollen wir betriebliche Informationssysteme in den Software-Gesamtmarkt einordnen und sie von anderen Arten von Softwaresystemen abgrenzen. Eine Taxonomie des gesamten Marktes für Software gibt [HeB01] an. An ihr orientieren wir uns, da sie einen vielfach akzeptierten Industriestandard darstellt. Im Folgenden fassen wir die Taxonomie kurz zusammen und geben an, wie sich betriebliche Informationssysteme einordnen. Drei große Kategorien von Software werden am Markt unterschieden:[33]

Application Software (dt.: Anwendungssoftware). Die betrieblichen Informationssysteme umfassen die folgenden Unterkategorien:

- Anwendungen für Enterprise Resource Management: dazu gehört Buchhaltung und Controlling, Personalverwaltung, Materialwirtschaft, Logistik, Wartung und Projektmanagement
- Anwendungen für Service-Industrien, d.h. Banken und Versicherungen, Gesundheitswesen, Professional Services und Telekommunikation/Versorgung
- Anwendungen für Customer Relationship Management (CRM)
- Anwendungen für die Wertschöpfungskette in der Fertigungsindustrie sowie Groß- und Einzelhandel

[33] Wir übernehmen die englischen Termini, da sie nicht immer eindeutig zu übersetzen sind und vielfach auch im Deutschen unübersetzt verwendet werden. Einige wichtige Übersetzungen geben wir in Klammer an.

Daneben gibt es folgende Unterkategorien: Technische Anwendungen (CAD, CAE, CAM, Product Information Management PIM), Kollaborationsanwendungen und Verbraucher-Anwendungen (u.a. Software für Spiele, Unterhaltung, Aus- und Weiterbildung).

Application Development and Deployment Software (dt.: Software für Entwicklung und Bereitstellung von Anwendungen). Diese Kategorie umfaßt u.a. alle Arten von Software für das Management von Datenbanken, Werkzeuge für Erstellung und Lebenszyklusmanagement von Anwendungen, Anwendungsserver, Spreadsheets, Executive Information Systems und Data Mining-Software.

System Infrastructure Software (dt.: Systeminfrastruktursoftware). Diese Kategorie umfaßt u.a. die folgenden Unterkategorien: System-Level Software (u.a. Betriebssysteme), System-Management, Netzwerk-Management, Middleware.

Daneben gibt es eine weitere große Kategorie von Software, die in der Taxonomie nicht erwähnt wird, weil sie nicht frei am Markt verfügbar ist: *Embedded Software (dt.: eingebettete Software).* Sie liegt quer zu den oben genannten drei Kategorien, weil sie Elemente aus allen drei enthält. In der Regel ist sie sehr stark von der jeweiligen technischen Infrastruktur beeinflußt.

Hierbei ist zu beachten, daß betriebliche Informationssysteme viele andere Arten von Software nutzen, ohne die sie nicht arbeitsfähig wären, z.B. Datenbankmanagementsysteme, Middleware und nicht zuletzt Betriebssysteme und Messaging Software.

2.5.2 Wesentliche Merkmale betrieblicher Informationssysteme

Betriebliche Informationssysteme sind für die nutzenden Unternehmen in vielen Fällen geschäftskritisch: ein Ausfall führt zu einer erheblichen Beeinträchtigung, teilweise sogar bis zum Erliegen des Geschäfts. Beispiele hierfür sind das Kontokorrentsystem einer Bank, das Warenwirtschaftssystem einer Handelskette oder das Produktionsplanungs- und Steuerungssystem eines Automobilherstellers. Dementsprechend wird betrieblichen Informationssystemen in der Regel eine hohe Zuverlässigkeit abverlangt, die deutlich über Anwendungen für Verbraucher liegen, aber nicht so hoch sind wie bei extrem sicherheitskritischen Systemen, bei denen ein unverhältnismäßig hohes Risiko mit dem Ausfall verbunden ist z.B. bei Systemen für militärische Anwendungen oder Kernkraftwerke.

Die Bedeutung betrieblicher Informationssysteme für die nutzenden Unternehmen nimmt ständig zu, indem immer größere Bereiche von ihnen durchdrungen werden. In der Regel werden sie von vielen Nutzern gleichzeitig verwendet, weswegen die Anzahl der gleichzeitig damit arbeitenden Endbenutzer einige Tausend betragen kann. Das System muß genau auf die dadurch verursachte Transaktionslast vorbereitet sein, damit Zuverlässigkeit und Reaktionsgeschwindigkeit gesichert sind. Die erwarteten Reaktionszeiten liegen typischerweise im Bereich weniger Sekunden; harte Echtzeitanforderungen sind selten.

Betriebliche Informationssysteme sind aus drei wesentlichen Elementen aufgebaut: Datenverwaltung, Anwendungskern und Benutzerschnittstelle (vgl. [Den91], 222). Ihre Architektur ist häufig in Schichten strukturiert, die an diesen Elementen orientiert sind (vgl. [BMR00], 49); im folgenden Abschnitt gehen wir näher auf diese sog. *Schichtenarchitektur* ein.

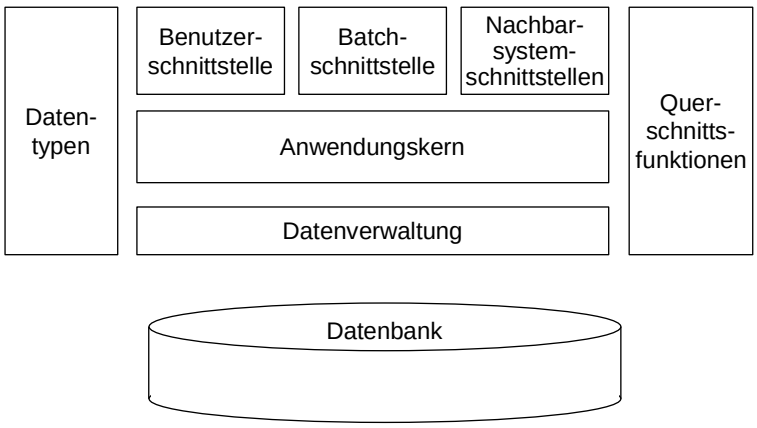

Abbildung 2-20: Betriebliches Informationssystem (nach [Sie00])

Das Modell eines betrieblichen Informationssystems (nach [Sie00]) ist in Abbildung 2-20 dargestellt. Die Datenverwaltung stellt typischerweise die Verbindung zu einer großen und komplexen Datenbank her, auf die die Anwender online zugreifen (vgl. [Den91], 15). Die Benutzerschnittstelle ermöglicht die Präsentation der Dialoge an der Benutzeroberfläche. Als solche wird heute meist ein Graphical User Interface (GUI) zur Datenein- und ausgabe genutzt, mit dem der Endbenutzer interaktiv arbeiten kann. Der entsprechende Modus des Systems heißt Dialogbetrieb. Daneben gibt es den sog. Batchbetrieb, der der massenhaften Verarbeitung von Daten ohne direkte interaktive Einwirkung des Endbenutzers dient (vgl. [Den91], 15f); dafür gibt es eine eigene Batchschnittstelle.

Die inhaltliche Verarbeitung der Daten geschieht im Anwendungskern. Seine Funktionen können als virtuelle Sachbearbeiter modelliert werden (vgl. [Den91], 222f), die den tatsächlichen Sachbearbeitern nachempfunden sind. Der Anwendungskern greift häufig auf Daten und Funktionen von unterschiedlichen Nachbarsystemen zu (Upstream-Systeme) bzw. stellt Daten oder Funktionen für sie zur Verfügung (Downstream-Systeme). Dafür sind Nachbarsystem-Schnittstellen erforderlich.

Für die Entwicklung des Anwendungskerns ist substantielle Expertise des entsprechenden Anwendungsgebiets notwendig. Im Gegensatz dazu ist der Schwerpunkt der geforderten Expertise bei Datenverwaltung und Benutzerschnittstelle im technischen Bereich.

Während betriebliche Informationssysteme anfangs ausschließlich als Host-Systeme zentral auf Großrechnern realisiert wurden, hat in jüngerer Zeit der Anteil verteilter Client-Server-Systeme stark zugenommen. Als Rechnerplattformen für Client/Server-Systeme werden in der Regel Personal Computer auf der Client-Seite und Workstations auf der Server-Seite eingesetzt, wobei die wesentlichen Betriebssysteme Microsoft Windows in verschiedenen Varianten

sowie Unix und Linux sind. Typische Großrechner-Betriebssysteme sind z.B. MVS und OS/390 von IBM sowie VMS von DEC. Ein weiteres wichtiges Element der Systemumgebung (vgl. 2.1.2) ist die verwendete Datenbank. Auch hier gibt es eine Reihe von Produktanbietern: Die wichtigsten sind Oracle, Informix und IBM mit DB2.

2.5.3 Die Schichtenarchitektur und ihre Elemente

Wie in 2.5.2 bereits erwähnt, kann man betriebliche Informationssysteme grob in Schichten strukturieren. Die Schichtenbildung hat eine Reihe von Vorteilen, die ihren Einsatz motivieren:

- *Trennung von Technik und Anwendung.* Die Trennung von Technik und Anwendung sorgt dafür, daß Änderungen sich im Sinne des Geheimnisprinzips jeweils nur auf einen Teil des Systems auswirken. Außerdem wird dadurch der Spezialisierung der Entwickler Rechnung getragen, deren Expertise sich meist auf einen dieser Bereiche konzentriert (vgl. auch 2.3.6).
- *Erleichterung der Verteilung*. Da betriebliche Informationssysteme heute oft als Client/Server-Systeme realisiert werden (siehe 2.5.2), muß das System zwischen Client- und Server-Rechnern verteilt werden. Diese Aufgabe, für die auch der Ausdruck Client-Server-Schnitt gebräuchlich ist, wird durch die Strukturierung in Schichten erleichtert.
- *Strukturierungshilfe*. Grundsätzlich ist es hilfreich, eine bewährte Strukturierung als Vorgabe zu nutzen, wenn man eine Entwurfsaufgabe angeht [Den01].

Ein häufig gewähltes Modell unterscheidet drei Schichten, die den drei wesentlichen Elementen betrieblicher Informationssysteme entsprechen: Datenverwaltung, Anwendungskern und Benutzerschnittstelle (siehe auch Abbildung 2-21). Die Schicht des Anwendungskerns wird auch als Domänenschicht bezeichnet ([Fow99], 258). Daneben sind auch andere Strukturierungen möglich, beispielsweise in zwei Schichten: Anwendung und Benutzerschnittstelle werden zusammengefaßt (vgl. [Fow99], 257f). Der Vorteil der Drei-Schichten-Architektur gegenüber der Zwei-Schichten-Architektur liegt darin, daß die enge Kopplung der Benutzerschnittstelle mit dem physikalischen Datenformat der Datenverwaltung vermieden wird, die viele Schwächen hat ([Fow99], 258).

Bei einer genaueren Analyse sind weitere Elemente des Systems zu erwähnen, die nicht einzelnen Schichten zugeordnet werden können: Querschnittsfunktionen und Datentypen. Typische Querschnittsfunktionen sind Fehlerbehandlung und Logging und Tracing. Sie werden in allen drei Schichten verwendet, ebenso wie die Datentypen.

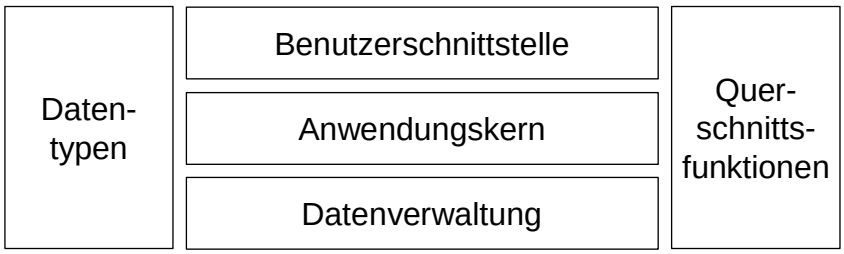

Abbildung 2-21: Schichtenarchitektur (drei Schichten)

Die Zerlegung eines betrieblichen Informationssystems in Schichten impliziert bereits ein nennenswertes Maß an Wiederverwendung innerhalb des Systems, denn die Komponenten des Anwendungskerns nutzen gemeinsam die anderen Elemente. Wir erläutern dies am Beispiel der Datenverwaltung: die Alternative zu einer gemeinsam genutzten Datenverwaltungsschicht ist, daß bei jeder Anwendungskomponente individuell Entwicklungsaufwand für die Regelung der Persistenz anfällt.

2.5.4 Besonderheiten und typische Schwierigkeiten bei der Entwicklung

Bei der Entwicklung betrieblicher Informationssysteme gibt es eine Reihe von Besonderheiten und typischen Schwierigkeiten, auf die wir näher eingehen wollen.

Zwischen Datenverwaltung und Anwendungskern tritt typischerweise ein Paradigmenbruch auf, der dadurch bedingt ist, daß eine relationale Datenbank genutzt wird, während der Anwendungskern in einer objektorientierten Programmiersprache ausgeführt wird. Die zugrundeliegenden relationalen bzw. objektorientierten Datenmodelle sind verschieden und müssen ineinander übersetzt werden. Diese Übersetzung, das sog. Objektrelationale Mapping (kurz O-R-Mapping) ist deswegen eine wichtige Aufgabe bei der Entwicklung betrieblicher Informationssysteme [Kel97].

Bei der Anbindung des Anwendungskerns an das GUI ist eine ganze Reihe anderer Probleme zu lösen, z.B. die Transformation der Daten in ihre graphische Darstellung, die Verwaltung des Dialoggedächtnisses oder die gleichzeitige Nutzung direkt und via Web angebundener Clients. Für viele dieser Probleme gibt es noch keine allgemein akzeptierten Lösungen [KRW01], so daß die GUI-Anbindung insgesamt weniger gut beherrscht ist als die Anbindung der Datenverwaltung.

Eine weitere typische Schwierigkeit beim Bau betrieblicher Informationssysteme liegt darin, ein komplexes Anwendungsumfeld auf ein Softwaresystem abzubilden − weniger in der Lösung anspruchsvoller algorithmischer Probleme, wie sie z.B. bei technischen Systemen insbesondere eingebetteter Software auftreten. Das wirkt sich vor allem auf den Anwendungskern aus. Dementsprechend werden Entwickler benötigt, die neben solidem softwaretechnischem Grundwissen auch über solide Expertise in einem Anwendungsgebiet, z.B. Buchhaltung/Controlling, Materialwirtschaft oder Personalverwaltung,

verfügen. Entwickler, die über diese Kombination an Wissen verfügen, sind selten.

Die Endbenutzer betrieblicher Informationssysteme stammen aus der jeweiligen Fachabteilung und haben dementsprechend in der Regel keine softwaretechnische Ausbildung. Sie sind lediglich Anwender, die bei technischen Problemen durch Fachleute unterstützt werden. Diese Fachleute gehören meist nicht zur jeweiligen Fachabteilung, sondern zur EDV-Abteilung der jeweiligen Firma. Diese Unterscheidung ist Grundlage für die in dieser Arbeit maßgebliche Definition des *Endbenutzers*: sie sind diejenigen Anwender, die das betriebliche Informationssystem zu Unterstützung der Abläufe bei ihrer jeweiligen fachlichen Arbeit nutzen, z.B. Buchhalter oder Sachbearbeiter für den Auftragseingang; sie sehen es also unter der fachlichen Perspektive. Die Administratoren des Systems, die es unter der technischen Perspektive sehen, sind nach unserer Definition keine Endbenutzer.

Betriebliche Informationssysteme werden häufig in ein Umfeld aus bereits bestehenden Systemen eingebettet, die sog. Systemlandschaft ([Hey00], 3). Diese Systemlandschaft verlangt umfangreiche Kommunikation mit unterschiedlichsten Nachbarsystemen, wodurch ein hoher Bedarf an externen Schnittstellen entsteht.

Das Ausmaß der Standardisierung ist von Schicht zu Schicht unterschiedlich. Am höchsten ist es bei der Datenverwaltung; da die von verschiedenen Datenbanken bereitgestellten Anwendungsschnittstellen (engl.: Application Programming Interfaces/APIs) sich sehr stark ähneln, sind die Unterschiede zwischen unterschiedlichen für die Benutzerschnittstelle einsetzbaren GUI-Bibliotheken, z.B. Swing und MFC, bereits deutlich größer, da die Art der Darstellung der Daten durch individuelle Endbenutzerwünsche beeinflußt wird. Der Anwendungskern schließlich ist hauptsächlich durch Anforderungen der Endbenutzer und von Nachbarsystemen beeinflußt, für die es keine produktübergreifenden Standards gibt.[34] Produkte, die vereinzelt De-Facto-Standards setzen, stellen einen sehr großen Funktionsumfang bei gleichzeitiger hoher Variabilität bereit. Sie setzen den Standard also nicht durch Auswahl einer Alternative, sondern durch Abdecken möglichst vieler Alternativen. Zusammenfassend läßt sich feststellen, daß ein typisches Problem betrieblicher Informationssysteme darin besteht, daß mit zunehmendem Einfluß von Endbenutzern und Nachbarsystemen auf die Anforderungen die Standardisierbarkeit dieser Anforderungen abnimmt und daß diese beim Anwendungskern am geringsten ist.

Eine weitverbreitete Gefahr bei der Entwicklung betrieblicher Informationssysteme besteht darin, daß ihre Struktur monolithisch wird (vgl. [Szy99], xiii), d.h. daß es nicht möglich ist, Teile des Systems unabhängig zu entwickeln und zu ändern. Der Grund dafür ist, daß zu große Abhängigkeiten bestehen, d.h. die einzelnen Teile des Systems nicht ausreichend entkoppelt sind. Dadurch wird insbesondere die Wiederverwendbarkeit des Systems oder von Teilen des Systems beeinträchtigt, weil Anpassungen, die Änderungen erfordern, erschwert werden (vgl. 2.3.4).

---

[34] Einzelne Produkte setzen teilweise De-Facto-Standards, z.B. SAP R/3 bei ERP-Anwendungen.

Kapitel 3

Umfeld und Positionierung der Arbeit

Dieses Kapitel faßt den Stand der Forschung zu den verschiedenen Aspekten der Entwicklung wiederverwendbarer Software zusammen. Ziel ist, den Forschungsbedarf zu bestimmen und die eigenständigen Beiträge der Arbeit herauszuarbeiten.

Zunächst stellen wir die Literatur zur Entwicklung wiederverwendbarer Software vor und fassen ausgewählte Beiträge zusammen. Dabei gehen wir nach folgenden Bereichen gegliedert vor: umfassende und spezifische Beiträge, empirische Untersuchungen sowie neuere Entwicklungen und Trends. Schließlich stellen wir den Forschungsbedarf zusammengefaßt dar, ordnen die Arbeit in das wissenschaftliche Umfeld ein, bestimmen ihren Beitrag und fassen sich aus der Literatur ergebende Forschungsfragen zu empirisch überprüfbaren Hypothesen zusammen.

Inhalt: **Seite**

3.1 Einleitung

Software-Wiederverwendung ist ein Gebiet, das viele Bereiche des Software Engineering berührt (vgl. 1.1). Entsprechend vielfältig ist die Literatur, auf die wir in unserer Arbeit zurückgreifen. Dies gilt umso mehr, als wir verschiedene Methoden wissenschaftlichen Arbeitens einsetzen – u.a. Fallstudien und Monte-Carlo-Simulation – und die Betrachtung über die softwaretechnischen auf die ökonomischen Aspekte ausdehnen. Dieses Kapitel gibt einen strukturierten Überblick über die Literatur, faßt ausgewählte Beiträge zusammen, stellt den Forschungsbedarf dar, ordnet die Arbeit in das wissenschaftliche Umfeld ein, bestimmt ihren Beitrag und formuliert empirisch zu überprüfende Hypothesen.

Die Literatur zur Entwicklung wiederverwendbarer Software läßt sich grob in Gruppen aufteilen. Dabei gehen wir zunächst auf die Veröffentlichungen ein, die wir in diesem Kapitel analysieren:

- Veröffentlichungen, die sich spezifisch und umfassend mit Wiederverwendung – insgesamt oder fokussiert auf die Herstellung – befassen und alle Aspekte abdecken, die sie jeweils für wesentlich halten [JGJ97, Kar95, Sam97, McC97, Tra95, Rei97, Cou97, Bas97, Lim98, CaE95, Krg92]. Die Einschätzung der Bedeutung der Aspekte ist naturgemäß von Autor zu Autor verschieden. Auf diese Gruppe von Beiträgen gehen wir in 3.2 näher ein.
- Veröffentlichungen, die Wiederverwendung empirisch untersuchen [Gri94, GrW95, Lim94, Joo94, MSG96, Ill99, FrF95, HaP97, MET99]. Wir besprechen sie in 3.3.
- Aktuelle Veröffentlichungen aus dem Umfeld der Entwicklung wiederverwendbarer Software, aus denen sich neuere Entwicklungen und Trends ablesen lassen [ShG96, BCK98, BrS02, GAO95, Ran99, Coh99, Gri98, GHJ94, BMR00, KeM99, NoK01, Szy99, Grf98, HeS99, DSW98, But99, FSJ99, Wol02, FHB00, Btl99, Cle99, WeL99, BFK99, AnG01, Lut99, KeM99, CHW98, GBS01, BBa01, TOH99, OsT01, SuR01, Ald00, Har01, AOS02, KIL96, KLM97, KHH01, BeA01, Beu99, HaU01, PoC93, Jon94, Wit96, FFF97, WiB98, BSa97, Wil99, MSR99, Sci99, MFG00, MeL95, Lim98a, FFa99]. Auf sie gehen wir in 3.4 ein.

Daneben baut diese Arbeit auf einer große Zahl anderer Veröffentlichungen auf, die an den jeweils relevanten Stellen referenziert, in diesem Kapitel jedoch nicht analysiert werden:

- Veröffentlichungen, die gezielt einzelne Aspekte der Entwicklung wiederverwendbarer Software untersuchen, u.a. [Par72, PCW83, Par76, Par85, Par79, Boe81, Boe84, GAO95, CHW98, CCo94, SGB01, Bai99, Cox87, Cox90].

- Veröffentlichungen, die wesentliche Aspekte der Entwicklung wiederverwendbarer Software in einem übergeordneten Kontext behandeln, u.a. [Bro87, Bro95, Gol95, McI68, BSi00, SiD00, Fow99].
- Beiträge, die Grundlagen des Software Engineering und der Entwicklung betrieblicher Informationssysteme behandeln und dabei teilweise und in unterschiedlicher Tiefe auf die Grundzüge der Wiederverwendung eingehen [Pre97, Som96, Pfl01, Bal96, Bal99, Den91]. Wir greifen auf sie immer wieder zurück, wenn es um grundlegende Fragen des Software Engineering geht, weniger in spezifischen Fragen der Entwicklung wiederverwendbarer Software.
- Beiträge aus unterschiedlichen Gebieten der Informatik, insbesondere des Software Engineering einschließlich der industriellen Praxis der Softwareentwicklung, u.a. [AHU74, Joh01, Sny86, Amb98, Tau01, Bab97, Box98, BeS97, BJR98, CuS95, End01].
- Beiträge aus anderen Gebieten, auf die wir zurückgreifen, weil sie relevante Grundlagen für unsere Arbeit enthalten, u.a. [BrM95, Att95, CKM94, Eis89, Bar99, Bra01, BSe01, FKP99, Hoc99, Kaf98].
- Zahlreiche Quellen, die wir in unseren Fallstudien verwenden, u.a. [SiF01, BeS01a, BeS01b, KrL98a, KrL98b, KHL98, HöS99, Fis98].

3.2 Umfassende und spezifische Beiträge zur Entwicklung wiederverwendbarer Software

3.2.1 Einführung

Wiederverwendung hat viele Facetten; die für diese Arbeit wesentlichen haben wir in Kapitel 2 näher beleuchtet. Zunächst untersuchen wir die bedeutendsten der Beiträge, die sich umfassend und spezifisch mit der Entwicklung wiederverwendbarer Software befassen.

Auswahlkriterien

Eine Reihe von umfangreichen Beiträgen befaßt sich mit Software-Wiederverwendung: [JGJ97, Kar95, Sam97, McC97, Tra95, Rei97, Cou97, Bas97, Lim98, CaE95, Krg92].[35] Um die wesentlichen Erkenntnisse in überschaubarer Weise aufzubereiten, haben wir die bedeutendsten ausgewählt, die dem Fokus unserer Arbeit entsprechen, und sie einer detaillierten Analyse unterzogen. Folgende Kriterien waren dafür ausschlaggebend:

1. Die Entwicklung wiederverwendbarer Software wird *umfassend* behandelt, d.h. der gesamte Entwicklungsprozess und alle wesentlichen Aspekte ge-

[35] Wir haben nur Werke berücksichtigt, deren Erscheinungsdatum nicht länger als zehn Jahre zurückliegt.

mäß Kapitel 2 werden abgedeckt und die Betrachtung ist nicht auf eine bestimmte Programmiersprache beschränkt.

2. Sie befassen sich *spezifisch* mit Software-Wiederverwendung, d.h. Wiederverwendung wird nicht eingebettet in einen größeren Kontext.
3. Es wird eine eigene *Methode* für die praktische Umsetzung entwickelt, nicht nur analysiert.
4. Der Beitrag hat einen hohen *Einfluß* auf die wissenschaftliche Diskussion. Dieses Kriterium ist notwendig, da wir uns wegen der großen Zahl auf die bedeutendsten Beiträge beschränken müssen. Zur Messung wird ausgewertet, wie oft das jeweilige Werk und die Autoren in anderen Veröffentlichungen zitiert werden; die Daten beziehen sich auf [NEC01].

Überblick über die Beiträge

Einen Überblick über die Bewertung der Beiträge nach diesen Kriterien gibt Tabelle 2-4. Drei davon wurden aufgrund der Bewertung für eine detaillierte Analyse ausgewählt: [JGJ97, Kar95, McC97].

Autor(en)	Beitrag	Jahr	Bewertung nach Kriterien				Fazit
			1 Spezifisch	2 Umfassend	3 Methode	4 Einfluß (Werk/ Autoren)	
Jacobson, Griss, Jonsson	[JGJ97]	1997	Ja	Ja	Ja	7/ 1637, 140, 73	Ausgewählt
Karlsson	[Kar95]	1995	Ja	Ja	Ja	21/63	Ausgewählt
McClure	[McC97]	1997	Ja	Ja	Ja	2/62	Ausgewählt
Tracz	[Tra95]	1995	Ja	Nein[36]	Ja	3/226	Nicht ausgewählt
Lim	[Lim98]	1998	Ja	Ja	Nein	0/4	Nicht ausgewählt
Sametinger	[Sam97]	1997	Ja	Ja	Nein	21/70	Nicht ausgewählt
Reifer	[Rei97]	1997	Ja	Nein[37]	Ja	3/17	Nicht ausgewählt

[36] Keine systematische Behandlung softwaretechnischer Aspekte (insbesondere Architektur)

[37] Geht nicht ausreichend auf technische Aspekte ein.

Coulange	[Cou97]	1997	Ja	Ja	Ja	1/1	Nicht ausgewählt[38]
Bassett	[Bas97]	1997	Ja	Nein[39]	Ja	7/14	Nicht ausgewählt
Carroll, Ellis	[CaE95]	1995	Ja	Nein[40]	Ja	8/21, 140	Nicht ausgewählt
Krueger	[Krg92]	1992	Ja	Nein[41]	Nein	2/11	Nicht ausgewählt

Tabelle 3-1: Überblick zur Auswahl stehender Beiträge zur Wiederverwendung

Vorgehen bei der Analyse

Wir untersuchen die genannten Beiträge jeweils auf ihre Grundlagen und auf die vorgeschlagene Methode. Die Grundlagen gliedern sich in empirische Basis und – explizit oder implizit gemachte – Annahmen. Um die Methode verstehen und bewerten zu können, ist es essentiell, diese Annahmen zu erheben.

Die *empirische Basis* untersuchen wir nach folgenden Kriterien:

- Untersuchte Projekte: untersuchte Firmen, entwickelte Software, operative Durchführung und Besonderheiten
- Ergebnisse der empirischen Untersuchungen: quantitative Ergebnisse, qualitative Ergebnisse und Schlußfolgerungen

Bezüglich der *Annahmen* werden die folgenden Punkte erhoben:

- Aktuelles Vorgehen: Umfang und Art der Wiederverwendung, Tool-Unterstützung, Quellen für wiederverwendbare Software
- Einschätzung der Wiederverwendung: Grundsätzliche Haltung der Entwickler, Vor- und Nachteile, Priorisierung und Abwägung der Vor- und Nachteile, Motivation der Wiederverwendung
- Hindernisse und notwendige Maßnahmen: Hindernisse, Defizite und Maßnahmen zu ihrer Überwindung, Bedeutung von Anreizen, Erfolgsfaktoren
- Ökonomische Aspekte: Kosten und Nutzen sowie deren Quantifizierung, andere Parameter der ökonomischen Bewertung, Implikationen für Organisation und Strategie

[38] Geringer Einfluss

[39] Ökonomische Aspekte nicht untersucht

[40] Beschränkt auf Programmiersprache C++; ökonomische Aspekte nicht untersucht

[41] Ökonomische Aspekte nicht untersucht

Im Hinblick auf die *vorgeschlagene Methode* betrachten wir folgende Punkte:

- Generelles Vorgehen: Entwicklungsmethode, empfohlene Art der Wiederverwendbarmachung, Kriterien für die Entscheidung über Projekte zur Entwicklung wiederverwendbarer Software
- Softwaretechnik: Vorgeschlagene Entwicklungsmethodik, Besonderheiten des Entwicklungsprozesses, Behandlung von Variabilität in Spezifikation und Konstruktion, Architektur wiederverwendbarer Software, Merkmale der Wiederverwendbarkeit und ihre Wechselwirkungen
- Ökonomische Bewertung: Verfahren, Zielgröße, Behandlung von Zeitwert des Geldes und Unsicherheit
- Management-Aspekte: Projektmanagement, Organisatorische Maßnahmen, Einordnung in die technische Strategie

In den folgenden Abschnitten fassen wir das Ergebnis der Analyse jeweils in einer kritischen Würdigung zusammen.

3.2.2 Kritische Würdigung von [JGJ97]

Aus unserer Sicht läßt [JGJ97] einige Lücken, die zu füllen sind. Zunächst bauen die Autoren auf der Erfahrung mit eingebetteter Software auf, so daß die Übertragbarkeit der Aussagen auf die Entwicklung betrieblicher Informationssysteme zu prüfen ist. Die empirische Basis ist schmal; die geschilderten Projekte werden nur oberflächlich beschrieben. Dabei werden wenig quantitative Daten zur Bewertung zur Verfügung gestellt und es wird nicht auf die Architektur eingegangen.

Die softwaretechnische Darstellung leidet unter der starken Betonung von UML. Die Frage des Vorgehens bei der Zerlegung bleibt offen, obwohl die große Bedeutung der Architektur herausgestellt wird. Eine unnötig hohe Hürde stellt die Definition eines eigenen Entwicklungsprozesses dar, weil sie die Optionen unnötig einschränkt. Auf die Merkmale wiederverwendbarer Software wird nicht systematisch eingegangen. Bei der Bewertung der Projekte schließlich gibt es mehrere Lücken: auf die Notwendigkeit einer gezielten Entscheidung wird ebensowenig eingegangen wie auf die softwaretechnische Bewertung. Das Modell zur ökonomischen Bewertung ist oberflächlich – der Einfluß der Wartung wird nicht explizit modelliert – und geht auf die Frage der Unsicherheit der Investition nicht ein. Zudem ist es nicht praktikabel in der Anwendung, da der notwendige Diskontierungsfaktor in vielen Fällen nicht sicher bestimmt werden kann (vgl. [BrM95], 183 und [CKM94], 330ff); der Kapitalwert als Meßgröße hat schließlich mehrere Nachteile (vgl. 2.4.4).

Ein großer Mangel des Buches sind unklar bestimmte Begriffe. Wir greifen zur Veranschaulichung den zentralen Begriff der Komponente heraus. Sie wird zunächst sehr weit definiert[42]; in der Folge wird der Begriff dann jedoch synonym

[42] "A component is a type, class or any other workproduct that has been specifically engineered to be reusable" ([JGJ97], 85). Zum Begriff des workproduct: " A workproduct is a unit of code, a document or piece of software model that can be independently managed within a software engineering organization. Individual types, classes and other model elements from the various models, diagrams, and related documents are typical workproducts. Complete

mit Klasse verwendet, z.B. indem Vererbung als ein Variabilitätsmechanismus für Komponenten angeführt wird ([JGJ97], 102). Irritierend ist darüber hinaus, daß ständig von einem "Business" die Rede ist, aber dafür übliche Vorgehensweisen, wie z.B. die Erstellung eines Business Case, nicht erwähnt werden.

3.2.3 Kritische Würdigung von [Kar95]

Eine Reihe wichtiger Punkte werden in [Kar95] nicht ausreichend behandelt. Zuerst ist das minimale empirische Fundament zu bemängeln: die beschriebenen Projekte werden bis auf eines äußerst kurz und unsystematisch wiedergegeben. Insgesamt werden weder ausreichende Daten zur ökonomischen Bewertung erhoben noch wird auf die Architektur eingegangen.

Die Entscheidungskriterien für Projektvorhaben lassen softwaretechnische Aspekte außer acht. Eine große Lücke im vorgeschlagenen softwaretechnischen Vorgehen besteht darin, daß das Problem der Zerlegung in Komponenten nicht behandelt wird. Positiv ist allerdings hervorzuheben, daß kein bestimmter Entwicklungsprozeß propagiert wird. Auch die Merkmale wiederverwendbarer Software werden in einiger Tiefe behandelt; eine Untersuchung der Wechselwirkungen und Auswirkungen auf andere Ziele als das der Wiederverwendbarkeit wird jedoch nicht vorgenommen. Im Hinblick auf die ökonomische Bewertung ist das umfassende Modell zu loben. Das Bewertungsverfahren ist jedoch wegen der Schwierigkeiten bei der Ermittlung des Diskontierungszinssatzes nicht praktikabel. Eine zusätzliche Schwäche ist, daß die Investitionsunsicherheit nicht explizit berücksichtigt wird.

Generell hat das Buch eine übersichtliche Struktur; insbesondere die Trennung nach Entwicklung für und Entwicklung mit Wiederverwendung ist hilfreich, auch die Begriffe sind größtenteils klar definiert. Leider bestehen jedoch Inkonsistenzen in der Aussage: u.a. ist der Entwicklungsprozeß nicht auf Komponenten eingerichtet.

3.2.4 Kritische Würdigung von [McC97]

Einige wichtige Probleme werden in [McC97] nicht behandelt. Zunächst ist die empirische Basis unzureichend: die Datenerhebung ist unsystematisch, es werden keine quantitativen Daten erhoben, Architektur und ökonomische Bewertung werden nicht behandelt. Ferner wird das Problem der Entscheidung über Projektvorhaben nicht angesprochen.

Die softwaretechnische Betrachtung läßt den Aspekt der Architektur völlig außer acht. Ebensowenig wird eine systematische Behandlung der Merkmale wiederverwendbarer Software durchgeführt. Hinsichtlich der ökonomischen Bewertung bestehen zwei Schwächen: Zum einen wird das Modell der Ent-

---

models, subsystems, and test models are also workproducts. Some workproducts are abstract, management-oriented entities and not always specific pieces of software. For example, a workproduct can be a configuration file, listing the names and versions of other software workproducts that are intended to be used together". ([JGJ97], 82)

wicklung wiederverwendbarer Software nur oberflächlich beschrieben, zum anderen berücksichtigt die Bewertungsmethode weder den Zeitwert des Geldes noch die Unsicherheit der Investition.

Positiv hervorzuheben ist die getrennte Behandlung von Herstellung und Nutzung wiederverwendbarer Software. Auch daß der bestehende Prozeß ergänzt wird, statt ihn im Ganzen zu verändern, ist aus unserer Sicht begrüßenswert. Negativ fällt hingegen die mangelnde Klarheit in der Strukturierung auf. Das Buch enthält viele Listen, deren Punkte sich teilweise überschneiden und unterschiedlichen Kategorien zuzurechnen sind, und deren Abdeckungsgrad im Dunkeln bleibt.[43]

3.3 Empirische Untersuchungen der Wiederverwendung

In einer Reihe von Veröffentlichungen werden empirische Untersuchungen unterschiedlicher Aspekte der Wiederverwendung dargestellt [Gri94, GrW95, Lim94, Joo94, MSG96, Ill99, FrF95, HaP97, MET99]. Wir stellen sie hier vor, fassen die wichtigsten Ergebnisse zusammen und würdigen sie kritisch.

3.3.1 Untersuchungen bei einzelnen Unternehmen

Hewlett-Packard

In drei Veröffentlichungen wird über die Erfahrungen mit Wiederverwendung bei der Firma Hewlett-Packard berichtet [Gri94, GrW95, Lim94]. Hewlett-Packard ist ein großes amerikanisches Unternehmen, das neben Hardware in Form von Computern (PCs und Server) und Peripheriegeräten (Drucker, Massenspeicher) auch Software herstellt.

Kritische Würdigung von [Gri94, GrW95]. Beide Veröffentlichungen machen im Wesentlichen Aussagen zu den Rahmenbedingungen und weniger zum konkreten Vorgehen. Insbesondere die Probleme der Architektur und ökonomischen Bewertung werden nicht behandelt. Daneben bestehen folgende Mängel: die Aussagen werden größtenteils nicht plausibel hergeleitet und begründet,

[43] Vgl. beispielsweise die Liste der Gründe dafür, warum Wiederverwendung scheitert: "Not-Invented-Here-Syndrome, Lack of understanding about why to practise reuse, Belief that reuse is counter-creative, Lack of long-term commitment and support from management, No reuse champion, No methodology support for reuse, Corporate culture and reward system discourage reuse, No reuse training or experience in practicing reuse, Unwilling to change current way of working, Management not convinced about the business value of reuse, View reuse as a high-risk technology, No tools to support the practice of reuse, Nothing to reuse; no software reuse library" ([McC97], 19)

sondern postuliert. Zudem sind die Aussagen ausschließlich qualitativer Natur.[44]

Kritische Würdigung von [Lim94]. Die in [Lim94] beschriebene Untersuchung ist von großem Wert, weil sie die einzige ökonomische Analyse in der Literatur ist, die eine solide quantitative Basis hat. Die ökonomische Analyse spricht für sich und läßt Wiederverwendung in einem positiven Licht erscheinen. Detaillierte Angaben zum Vorgehen und zu den Rahmenbedingungen fehlen jedoch, so daß nicht geklärt werden kann, ob die Ergebnisse unter anderen Bedingungen reproduzierbar wären. Beispielsweise wird nicht angegeben, welche Komponenten der jeweiligen Produkte wiederverwendet wurden; ebensowenig wird eine vergleichende Analyse der in den Kennzahlen wesentlich abweichenden Projekte durchgeführt, um die Ursache für die großen Unterschiede herauszuarbeiten. Schließlich beruht das Modell auf der Bestimmung des Kapitalwerts, der wesentlich eine Funktion des gewählten Diskontierungsfaktors ist. Die Bestimmung desselben ist in der Regel äußerst schwierig (vgl. 2.4.4); der gewählte Wert von 15 Prozent wird nicht motiviert.

Motorola

In [Joo94] wird über die Einführung von systematischer Wiederverwendung bei der Firma Motorola berichtet. Motorola ist ein großer amerikanischer Elektronikkonzern, der folgende Geschäftsfelder hat: Automobiltechnik, zellularer Mobilfunk, Funk, Halbleiter, Pager (Funkrufempfänger). Der Schwerpunkt der Softwareentwicklung liegt demzufolge auf eingebetteter Software und System-Level Software (vgl. 2.5); als Pilotprojekt wird explizit die Entwicklung von Compilern erwähnt.

Kritische Würdigung von [Joo94]. Ein generelles Manko von [Joo94] ist der Mangel an für Wiederverwendung spezifischen Aussagen. Konstruktion und Architektur wiederverwendbarer Software werden nicht behandelt. Darüber hinaus spielt der ökonomische Nutzen keine Rolle in der Betrachtung. Die subjektiven Einschätzungen der Autorin werden generell nicht untermauert, insbesondere nicht in quantitativer Form. Die postulierten Maßnahmen stehen deshalb in Frage.

Boeing

Zwei Veröffentlichungen berichten von Erfahrungen mit Software-Wiederverwendung bei der Firma Boeing [MSG96, Ill99].

Kritische Würdigung von [MSG96]. Die Aussagen in [MSG96] sind größtenteils sorgfältig belegt, gut nachzuvollziehen und differenziert; die Autoren gehen ausführlich auch auf Fehler ein, die gemacht wurden. Trotz der ausführlichen Begründung ihrer Bedeutung wird auf die gewählte Architektur leider nur oberflächlich eingegangen. Auch das Vorgehen bei der ökonomischen Analyse

[44] Beispiel: "We have learned that even with simple technology and only a few people focused on systematic reuse, you can incrementally build a reuse program that will produce significant improvements in quality, productivity, time to market, and competitive posture in just a few years." [GrW95]

wird nicht genau beschrieben; der Zeitwert des Geldes und die Unsicherheit scheinen nicht berücksichtigt worden zu sein.

Kritische Würdigung von [Ill99]. Das Vorgehen und die Architektur werden mit einiger Genauigkeit beschrieben. Ein Manko ist, daß sie ohne detaillierte Kenntnis der verwendeten Frames-basierten Entwicklungsmethode schwer zu verstehen sind. Eine Analyse des Nutzens aus ökonomischer Sicht wird außerdem nicht durchgeführt.

3.3.2 Untersuchungen bei mehreren Unternehmen

Über die Untersuchungen bei einzelnen Unternehmen hinaus wurden einige empirische Arbeiten veröffentlicht, welche die Verhältnisse bei mehreren Unternehmen vergleichen [FrF95, HaP97, MET99].

Kritische Würdigung von [FrF95]. Die Autoren leisten einen großen Beitrag, indem sie einige wichtige Fragen empirisch in quantitativer (nicht nur qualitativer) Form untersuchen. Dabei stellen sie z.B. den weitverbreiteten Glaubenssatz in Frage, das Not-invented-here-Syndrom sei ein wesentliches Hindernis für Wiederverwendung (Frage 4). Die Untersuchung hat jedoch zwei Schwächen: (1) Was die Menge der Probanden angeht, so ist nicht gesichert, daß die Gruppen pro Unternehmen repräsentativ sind, zumal auch Personen befragt werden, die nicht direkt der Softwareentwicklung angehören. (2) Die Ergebnisse werden nicht nach Probandengruppe getrennt ausgewiesen, obwohl Unterschiede in der Beantwortung denkbar wären. Darüber hinaus ist anzumerken, daß die befragten Organisationen keine am Markt operierenden Unternehmen sind und somit die industrielle Praxis nicht notwendigerweise wiedergeben. Außerdem sind alle Organisationen bis auf eine in USA ansässig. Eventuelle Unterschiede zu anderen Ländern sind zu prüfen.

Kritische Würdigung von [HaP97]. Das Fehlen einer durchgängigen Systematik und einer soliden quantitativen Basis macht die Untersuchungen bruchstückhaft und gibt vielen Aussagen einen eher anekdotischen als wissenschaftlichen Charakter[45]. Die abgeleiteten Empfehlungen sind vielfach eher vage[46]. Ein durchgängiges Eingehen auf die gewählte Architektur wäre wünschenswert.

Kritische Würdigung von [MET99]. In zweifacher Hinsicht leistet [MET99] einen Beitrag: (1) es werden europäische Unternehmen untersucht und so die größtenteils auf USA zentrierte Literatur ergänzt, (2) es findet ein empirisch fundierter Vergleich zwischen mehreren Unternehmen statt. Der Beitrag ist eingeschränkt dadurch, daß die Begriffe äußerst unscharf sind, beispielsweise wird nicht definiert, worin der Erfolg eines Projekts besteht. Auch die Behandlung der Frage nach dem richtigen Ansatz hinsichtlich der Architektur lei-

---

[45] Vgl. beispielsweise: "Unfortunately sufficient quantitative results to present a cost benefit analysis for these applications has not been made available to date, but all companies claim that their frameworks are very profitable" ([HaP97], 41)

[46] Exemplarisch sei angeführt: "Starting modestly with general purpose components normally breaks even after a few years" ([HaP97], 47)

det darunter, daß die Alternativen nicht sauber definiert sind. Neben der insgesamt geringen Anzahl erscheint aus unserer Sicht auch die Auswahl der Fragen verbesserungswürdig: daß sich z.B. für die Entwicklung wiederverwendbarer Software notwendige Investitionen nur schwer gegen das Top-Management durchsetzen lassen, erscheint selbstverständlich. Die Frage nach der ökonomischen Bewertung wird hingegen nicht behandelt.

3.4 Neuere Entwicklungen und Trends

Bei der Analyse aktueller Veröffentlichungen aus dem Umfeld der Software-Wiederverwendung zeigen sich zwei Gebiete mit interessanten neuen Entwicklungen (vgl. [SDD99, ZBB99]):

- Architektur wiederverwendbarer Software und
- Ökonomische und strategische Aspekte von Wiederverwendung.

Architektur ist insgesamt ein relativ junges akademisches Fach [Kar98] und sie ist von essentieller Bedeutung für die Wiederverwendbarkeit von Software (vgl. [Bro95], 224; [ShG96], 16; [Kar98]). Die Forschungstätigkeit in den letzten Jahren war äußerst rege: zu den verschiedenen Teilgebieten gibt es zahlreiche Veröffentlichungen, auf die wir in 3.4.1 eingehen.

Während sich die Forschung zur Wiederverwendung ursprünglich auf technische Aspekte konzentriert hat, wird die Bedeutung einer differenzierten ökonomischen und strategischen Bewertung zunehmend wahrgenommen (vgl. [Kar98, Gri98, ZAD97, ZBB99]). In den letzten Jahren gab es bedeutende Fortschritte, die wir in 3.4.2 näher betrachten.

3.4.1 Architektur

Softwarearchitektur hat große Bedeutung für die Entwicklung wiederverwendbarer Software (vgl. [Gri98, Coh99, Kar98], [ShG96], 7; siehe auch 2.3.2). Im Zentrum unseres Interesses steht die technische Architektur, insbesondere das Problem der Zerlegung des Systems in Komponenten bei möglichst guter Trennung der Zuständigkeiten zwischen den Komponenten (vgl. 2.3.2 bis 2.3.4, 2.3.6). Ziel ist dabei eine optimale Realisierung der Variabilität (vgl. 2.3.5). Folgende Teilgebiete, die im Kontext der Entwicklung wiederverwendbarer Software von besonderem Interesse sind, gehen wir im Folgenden näher ein:

- Entwurfsmuster
- Komponenten und Frameworks
- Variabilität und Produktlinien
- Trennung der Zuständigkeiten und Aspekt-orientierte Entwicklung

Daneben befassen sich einige neuere Veröffentlichungen allgemein mit Problemen des Entwurfs und der Architektur wiederverwendbarer Software. In [BrS02], einer aktuellen Veröffentlichung, welche die Objektorientierung kritisch analysiert, wird auch auf Wiederverwendung eingegangen und ein be-

deutender Mangel konstatiert: es wird festgestellt, daß die dafür häufig empfohlene Implementierungsvererbung kein geeignetes Mittel sei, um Wiederverwendbarkeit zu erreichen, insbesondere deshalb, weil sich damit große Systeme nicht strukturieren lassen. Generell stelle die Objektorientierung kein geeignetes Mittel zu Strukturierung großer Systeme zur Verfügung. Auf Probleme im Zusammenhang mit der Strukturierung großer Systeme geht auch [GAO95] ein. Als wesentliches Hindernis für die Nutzung wiederverwendbarer Komponenten wird herausgearbeitet, daß oft falsche Annahmen über die Struktur des Systems getroffen werden, in dem sie eingesetzt werden sollen.

Die populäre Lego-Metapher für aus Komponenten zusammengesetzte Systeme wird in [Ran99] in Frage gestellt: der durch die Architektur bestimmte Kontext der Nutzung müsse verstanden werden, wenn man Software-Komponenten erfolgreich wiederverwenden wolle. Dieser Kontext bestimme die Abhängigkeiten der Komponenten voneinander; dazu sei eine Zerlegung des Systems notwendig, die eine Trennung der Zuständigkeiten erreiche. Zur Lösung des Problems werden Schichten – sog. 'layers of functionality' – vorgeschlagen. Dabei bleiben jedoch die Fragen des Zusammenspiels der verschiedenen Schichten und der Priorisierung der Zuständigkeiten offen. Als Grundlage seiner Arbeiten gibt der Autor die Entwicklung eingebetteter Systeme an.

Eine Methode, durch Use Cases zur Architektur wiederverwendbarer Software zu gelangen, untersucht [Coh99]. Der Schwerpunkt der Betrachtung liegt auf Produktlinien. Die hohe Bedeutung einer sorgfältig entworfenen Architektur für die erfolgreiche Nutzung wiederverwendbarer Software streicht [Gri98] heraus; der beschriebene Ansatz besteht im Wesentlichen in objektorientiertem Vorgehen unter Verwendung von UML.

Entwurfsmuster

Entwurfsmuster (engl.: Design Patterns) sind eines der populärsten Forschungsgebiete der letzten Jahre. Gegenstand ist die abstrakte Beschreibung von guten Lösungen für typische, häufig auftretende Entwurfsprobleme. Diese Lösungen können beim Entwurf neuer Systeme genutzt werden, was Wiederverwendung auf der Ebene der Konstruktion darstellt. Das Standardwerk ist [GHJ94]; es leistet einen bahnbrechenden Beitrag zum Gebiet des Software Engineerings, beschränkt sich jedoch auf Muster einer niedrigen Granularitätsebene, so daß die Zuordnung zur Phase der Konstruktion und damit zur Softwarearchitektur keineswegs eindeutig ist; eine Zuordnung zur Programmierung ist genauso denkbar. Für die Strukturierung des gesamten Systems auf einer angemessenen Abstraktionsebene sind sie nicht geeignet.

Ein neueres Buch ähnlichen Zuschnitts ist [BMR00]. Das Thema wird in [BMR00] jedoch weiter gefaßt: neben Entwurfsmustern wird auch der Begriff der Architekturmuster eingeführt; die vorgestellten Muster bilden jedoch keine homogene Menge: sie bestehen einerseits aus den aus [ShG96] bekannten Architekturstilen, die eine Strukturierung des Systems nur auf der obersten Abstraktionsebene (z.B. in Schichten) ermöglichen, andererseits aus Mustern, die eher den Entwurfsmustern zuzuordnen sind (Model-View-Controller, Presen-

tation-Abstraction-Control) bzw. keine wirkliche Strukturierungsfunktion haben (Reflection).

Auf zwei weitere Veröffentlichungen zu einzelnen Aspekten von Entwurfsmustern [KeM99, NoK01] gehen wir unten ein.

Komponenten und Frameworks

Der Komponenten-Gedanke, mit Breitenwirkung 1968 zum ersten Mal formuliert und von Beginn an durch den Gedanken der Wiederverwendung motiviert [McI68], hat in den letzten Jahren einen rasanten Popularitätsanstieg erlebt, wie die große Zahl von Büchern zum Thema zeigt [Szy99, Grf98, HeS99, DSW98]. Von Interesse sind Komponenten für uns im Kontext des Systementwurfs als Einheiten der Zerlegung, die über Schnittstellen voneinander entkoppelt sind (vgl. 2.3.3); vor diesem Hintergrund untersuchen wir im Folgenden den Stand der Forschung.

Ein vielbeachtetes Buch zum Thema, das einen guten Überblick gibt, ist [Szy99]. Der Autor nennt Wiederverwendung als treibende Kraft der Beschäftigung mit Komponenten ([Szy99], 3). Neben einer soliden Begriffsbestimmung und der Behandlung von Schnittstellen wird ausführlich auf objektorientierte Techniken eingegangen. Große Teile des Buches befassen sich mit aktuellen technischen Standards für verteilte Komponenten wie Corba, DCOM und JavaBeans ([Szy99], 169ff). Der Gedanke eines kommerziellen Marktes für Komponenten wird formuliert, allerdings auf der Basis von schwer nachvollziehbaren Annahmen des Autors zur Funktionsweise solcher Märkte[47]. Wiederverwendbarkeit wird nicht nur auf Komponentenebene ([Szy99], 33f), sondern auch auf Architekturebene betrachtet ([Szy99], 133ff, 273ff). Dabei wird kurz auf Schnittstellen eingegangen, allerdings nur auf niedriger Granularitätsebene, ebenso auf Entwurfsmuster und Frameworks. Als Mittel zur Strukturierung ganzer Systeme werden Schichtenbildung und hierarchischer Aufbau in konzeptioneller Form angeführt, ohne das Vorgehen jedoch zu konkretisieren.

Ein weiteres Übersichtswerk ist [Grf98]. Ökonomische Aspekte von Komponenten werden hier gar nicht betrachtet, ebensowenig wie Architektur im Sinne einer Strukturierung des Gesamtsystems.

Mit Komponenten im Kontext betrieblicher Informationssysteme beschäftigt sich [HeS99], das als Vorgehen einen eigenen Ansatz namens 'Business Component Factory' propagiert, der im Wesentlichen auf bekannte Mittel zurückgreift. Keine Aussagen werden darin zur Strukturierung des Gesamtsystems gemacht. Auf Wiederverwendung gehen die Autoren nur insofern ein, als sie

[47] Zwei Zitate zur Veranschaulichung: "As with all mature markets, software component industries will eventually converge on a mixed income model" ([Szy99], 14) und "A new product can only create a market if its arrival is already awaited [...]" ([Szy99], 15). Die erste Behauptung, daß nämlich ein wesentlicher Teil der Einkünfte immer aus Services oder Werbung stammten, stimmt nicht einmal für alle Softwaremärkte: als Gegenbeispiel sei der Markt für Textverarbeitungssoftware angeführt; Marktführer Microsoft bezieht seine Einkünfte fast vollständig aus Lizenzgebühren. Als Gegenbeispiel zum zweiten Zitat führen wir den Markt für Supply Chain Management Software an, in dem führende Anbieter zu Beginn die Kunden durch Beratungsprojekte erst von der Nützlichkeit ihrer Produkte überzeugen mussten, z.B. die in Folge sehr erfolgreiche Firma i2 Technologies.

ihren Ansatz abgrenzen von sog. wiederverwendungsbasiertem Design (eng.: reuse-based design). Dabei wird behauptet, daß Kapselung ein wichtiges Merkmal komponentenbasierter Software, nicht aber wiederverwendbarer Software darstelle ([HeS99], 481f); dies trifft nicht zu (vgl. 2.3.4).

Als weiterer Ansatz zur Entwicklung komponentenbasierter Systeme und Frameworks wird in [DSW98] der sog. 'Catalysis Approach' beschrieben, der sich aus eingeführten Methoden zusammensetzt. Die Autoren nennen Wiederverwendbarkeit als wesentliche Motivation für komponentenbasiertes Entwickeln ([DSW98], 397); sie geben auch ein kurze, aber unspezifische Einführung in Wiederverwendung, wobei als wiederverwendbare Artefakte Frameworks propagiert werden ([DSW98], 453ff). Ein Kapitel über Architektur faßt bekannte Ideen kurz zusammen ([DSW98], 481ff). Trotz eines Kapitels zum Thema 'Business Model' werden ökonomische Aspekte nur beiläufig behandelt ([DSW98], 543ff). Ein Beitrag, der sich explizit mit Komponenten im Kontext von Wiederverwendung beschäftigt, ist [But99]. Der Autor gibt einen guten Überblick über das Thema.

In engem Zusammenhang mit Komponenten stehen *Frameworks*, die bereits in [Szy99] und vor allem [DSW98] eine wichtige Rolle spielten. Ein aktuelles Buch, das sich spezifisch mit dem Thema auseinandersetzt, ist [FSJ99]. Wiederverwendbarkeit ist zentrales Thema; Frameworks werden als 'object-oriented reuse technique' ([FSJ99], 3) gesehen und explizit über ihre Wiederverwendbarkeit definiert.[48] Als Beispiele für bestehende Frameworks führen die Autoren GUI Frameworks (MacApp, Microsoft Foundation Classes MFC) und Object Request Broker Frameworks an; sie postulieren, daß Frameworks auch komplexere Anwendungsdomänen erobern werden, können dafür aber kein Beispiel geben ([FSJ99], 9f). Insofern steht das Buch auf einem brüchigen Fundament. Ein weiterer Mangel ist, daß der Begriff Framework nicht konsistent verwendet wird (vgl. z.B. [FSJ99], 219). Das Kapitel über die Strukturierung großer Frameworks beschränkt sich im Wesentlichen auf eine Adaption der Schichten-Idee ([FSJ99], 395ff). Ein eigenes Kapitel beschäftigt sich ausführlich mit der strategischen Analyse von Investitionen in Frameworks ([FSJ99], 567ff). Wir gehen darauf in 3.4.2 ein.

Auch durch [Wol02] wird der Mangel an Beispielen für erfolgreiche Frameworks im Bereich betrieblicher Informationssysteme untermauert: hier wird das IBM SanFrancisco-Framework analysiert und als nicht erfolgreich eingestuft.

Eine neuerer Überblicksartikel [FHB00] geht insbesondere auf die Trennung der Zuständigkeiten als Kernziel der Frameworkentwicklung ein und verweist dazu auf Aspekt-Orientierte Programmierung.

Ein hybrider Entwicklungsprozeß, der Top-Down- und Bottom-Up-Elemente kombiniert, wird in [Btl99] für die Framework-Entwicklung vorgestellt. Außerdem wird vorgeschlagen, die Einsatzart von Frameworks mit zunehmender Reife anzupassen (Übergang von White-box zu Black-box).

---

[48] "A framework is a reusable, semi-complete application that can be specialized to produce custom applications". ([FSJ99], 4)

Variabilität und Produktlinien

In [Par76] wird der Begriff der *Produktfamilie* begründet. Er umschreibt eine Menge von Programmen, deren gemeinsame Eigenschaften die Unterschiede überwiegen und bei deren Entwicklung es von Vorteil ist, zunächst die gemeinsamen Eigenschaften zu studieren [Par76]. Variabilität der gemeinsamen Basis ist die zentrale Eigenschaft einer Produktfamilie (vgl. [Par76, Joh01, SGB01]), so daß die beiden Begriffe in engem Zusammenhang zueinander und auch zur Wiederverwendbarkeit stehen ([Cle99]; vgl. auch 2.1.2). Im Verlauf der Zeit änderte sich die Bezeichnung: statt Produktfamilien wird heute üblicherweise von *Produktlinien* (engl.: *Product-Line Software*) gesprochen [Joh01].

Einen guten aktuellen Überblick über das Gebiet der Produktlinien gibt [Cle99]; die Bedeutung der Architektur und Komponenten-basierter Entwicklung wird betont.

Ein weiterer aktueller Beitrag ist [WeL99]. Im Zentrum steht der sog. 'FAST'-(Family-Oriented Abstraction, Specification and Translation) Prozeß, der sehr detailliert beschrieben und durch ausführliche Beispiele illustriert wird. Er greift im Wesentlichen auf bekannte Konzepte zurück und setzt einen Schwerpunkt auf die Spezifikationsphase; die Frage der Architektur wird nicht behandelt, sie wird aber als ein wesentlicher Faktor für die Wiederverwendbarkeit genannt ([WeL99], 393f). Die ökonomische Analyse ([WeL99], 45ff) ist stark vereinfacht; weder der Zeitwert des Geldes noch die Unsicherheit der Investition werden berücksichtigt.

Ein weiterer Entwicklungsprozeß für Produktlinien namens 'PuLSE' wird in [BFK99] vorgestellt.

Eine Reihe von Veröffentlichungen befaßt sich jeweils mit einzelnen Aspekten von Produktlinien: In [AnG01] werden verschiedene Möglichkeiten der Implementierung von Variabilität bei der Programmierung systematisch verglichen. Mit der Sicherheit der Wiederverwendung von Produktlinien-Spezifikationen im Kontext der Entwicklung von Software für die Raumfahrt beschäftigt sich [Lut99]. In [KeM99] werden Muster für die Modellierung von Variabilität in Produktfamilien vorgeschlagen. Diese bauen allerdings größtenteils auf Implementierungsvererbung auf, die wegen der Verletzung des Geheimnisprinzips sehr kritisch zu bewerten ist [Sny86, BrS02]. Erfahrungen mit Produktlinien in der industriellen Praxis werden schließlich in [MSG96] berichtet (siehe 3.3).

Mehrere Veröffentlichungen behandeln als Schwerpunkt *Variabilität*: In [CHW98] beschäftigen sich die Autoren damit, wie Variabilität spezifiziert werden sollte, und schlagen dafür eine Methode namens 'SCV (Scope, Commonality, and Variability) Analysis' vor; ein großer Mangel ist, daß kein einziges Anwendungsbeispiel angeführt wird, das die sehr abstrakt beschriebene Methode illustriert. Die ökonomische Argumentation gleicht der in [WeL99] (s.o.) und hat dieselben Schwächen. Auf die Bedenklichkeit der auch hier als Variabilitätsmechanismus vorgeschlagenen Implementierungsvererbung sind wir bereits oben eingegangen.

Eine Bestandsaufnahme zur Variabilität mit einem Vorschlag zur Terminologie macht [GBS01]; die ökonomische Bewertung wird nicht behandelt. Einen ähn-

lichen Überblick gibt [BBa01]; auch hier wird die ökonomische Bewertung nicht angesprochen.

Trennung der Zuständigkeiten und Aspekt-orientierte Entwicklung

Trennung der Zuständigkeiten (engl.: separation of concerns) ist eines der zentralen Prinzipien des Softwareentwurfs und von eminenter Bedeutung für die Entwicklung wiederverwendbarer Software (siehe 2.3.4). In der Forschung der letzten Jahre hat sie große Beachtung erfahren, insbesondere unter dem Stichwort 'advanced separation of concerns' (vgl. z.B. [NoK01]).

In [TOH99] wird dazu ein Verfahren vorgestellt, das eine Zerlegung des Systems in sog. Hyperslices vorsieht und diese zu Hypermodules komponiert. Als Anwendungsbeispiel wird eine Software Engineering-Umgebung verwendet. Eine Schwäche ist, daß die Implementierung des Konzepts unzureichend behandelt wird. Das empfohlene package-Konstrukt in der Programmiersprache Java erfüllt z.B. die Anforderung nicht, daß ein Modul (in diesem Fall eine Klasse) in mehreren Hyperslices (packages) enthalten sein kann. Auf die zitierten Methoden des Aspekt-orientierten Programmierens gehen wir unten ein. Eine Umsetzung ihrer Konzepte unter Verwendung des Werkzeugs Hyper/J stellen dieselben Autoren in [OsT01] vor. Der Schwerpunkt hier liegt jedoch nicht so sehr auf der Trennung im Sinne des Geheimnisprinzips, sondern auf unterschiedlichen Möglichkeiten der Komposition, was die Gefahr des ripple effects heraufbeschwört (vgl. [BeA01]).

Die Forschung wird in verschiedene Richtungen vorangetrieben: Eine Anwendung der kognitionswissenschaftlichen prototypischen Kategorien-Lehre auf den Softwareentwurf im Sinne der multi-dimensional separation of concerns wird in [SuR01] vorgestellt. Die Autoren konzedieren jedoch, daß dabei einige bisher ungelöste Probleme bestehen. In [NoK01] wird vorgeschlagen, Entwurfsmuster als concerns zu betrachten und so austauschbar zu machen. Wir bezweifeln jedoch, daß die von den Autoren dafür postulierte Notwendigkeit tatsächlich besteht, da Entwurfsmuster normalerweise nach der Eignung für ein Problem gewählt werden und daher gerade nicht ausgetauscht werden sollten.

Eine Reihe von Testproblemen[49] für verschiedene Typen oder Dimensionen von Aufgaben[50] werden in [Ald00] beschrieben. Sieben Dimensionen werden vorgeschlagen: (1) Functional concerns, (2) Program organization (d.h. Struktur des Gesamtsystems), (3) Global properties, (4) Concerns leading to repetitive code, (5) System performance, (6) Special purpose concerns best specified in a declarative or graphical way, (7) Context dependent behavior. Eine weitere Überarbeitung dieser Dimensionen erachtet der Autor als notwendig, wobei er die Testprobleme zur Überprüfung des Fortschritts empfiehlt. Insgesamt trägt die Veröffentlichung sehr zur Systematisierung des Gebiets bei.

In [Har01] wird Mehrfachvererbung als Methode für Komposition empfohlen und Fragen im Zusammenhang mit der Generierung von Java-Code aus UML-

[49] engl.: challenge problems

[50] engl.: types or dimensions of concerns

Diagrammen untersucht. Sehr fragwürdig erscheint, daß Implementierungsvererbung als Weg zur Wiederverwendung beschrieben wird (vgl. 2.3.4).

Aspekt-orientierte Softwareentwicklung (engl.: Aspect-Oriented Software Development AOSD) ist eine bedeutende neue Methode, um Trennung der Zuständigkeiten zu erreichen; die Methoden von AOSD ermöglichen, nicht lokalisierbare Aspekte eines Systems zu modularisieren [AOS02].

In [KIL96] wurde die Methode unter dem Namen *Aspect-Oriented Programming* erstmalig publiziert. Im Kern besteht sie darin, jeden der relevanten Aspekte eines Systems separat ausdrücken zu können und sie anschließend automatisch mit Hilfe eines sog. Aspect Weavers zu kombinieren. Eine Reihe von offenen Fragen bestehen: u.a. gehören dazu die Fragen nach Entwurfsprinzipien für die Zerlegung in Aspekte und nach der Interaktion von Aspekten und Komponenten [KLM97]. In [KHH01] wird die Sprache AspectJ als Erweiterung der Programmiersprache Java vorgestellt, mit der Aspekt-orientiert entwickelt werden kann.

Mit der Frage, ob bei Aspekt-orientierter Entwicklung das Geheimnisprinzip gewahrt werden kann, beschäftigt sich [BeA01]. Eine Verletzung desselben konstatieren die Autoren für die Sprachen AspectJ und Hyper/J. Als mögliche Lösung wird der sog. 'Composition Filters'-Ansatz vorgeschlagen. Offen ist dabei jedoch, wie die Reihenfolge der Filter gehandhabt werden muß; die Frage, welche Aspekte separiert werden sollen bleibt ebenfalls offen.

Zwei aktuelle Veröffentlichungen beschäftigen sich mit der Wiederverwendung von Aspekten [Beu99, HaU01]. Dabei ist unklar, ob die Komplexität der entstehenden Konstrukte beherrschbar bleibt. Intuitiv scheint es, daß man sich vom eigentlichen Ziel, nämlich wiederverwendbare Software zu entwickeln, relativ weit entfernt. Darüber hinaus steht die grundlegende Annahme von [HaU01], daß nämlich Implementierungsvererbung die Basis von Wiederverwendbarkeit sein sollte, für uns in Frage (s.o.).

3.4.2 Ökonomische und strategische Aspekte von Wiederverwendung

Ökonomische und strategische Aspekte der Softwareentwicklung rücken zunehmend ins Blickfeld des wissenschaftlichen Software Engineering. Diese Aspekte lassen sich nicht streng trennen, da sie sich überlappen. Wir klassifizieren die Literatur deshalb danach, welchen Schwerpunkt sie setzt.

Es gibt eine erhebliche Zahl aktueller Veröffentlichungen zum Thema, die sich nicht auf den Kontext der Wiederverwendung konzentriert: einen Schwerpunkt auf Ökonomie setzen [Kry97, Pad99, AKK00, AsK01, FFi01, Han01, OhH01], auf Strategie [Erd99, BCJ99, BoB01, SuC01].

Die Literatur zu den *ökonomischen Aspekten der Software-Wiederverwendung* ist umfangreich. Einen guten Überblick über die Möglichkeiten der ökonomischen Bewertung wiederverwendbarer Software gibt [PoC93]. Neben einer einfachen Berechnung der Rendite (engl.: Return on Investment ROI) auf Projektebene werden für die Bewertung auf Firmenebene als Alternativen der Kapitalwert (vgl. 2.4.4) und der davon abgeleitete interne Zinssatz (vgl. 2.4.4) ange-

geben, wobei explizit darauf hingewiesen wird, daß das Risiko der Investition in Betracht gezogen werden muß, ohne jedoch ein Vorgehen dafür anzugeben. Unverständlich erscheint, daß bei der Bewertung auf Projektebene der Zeitwert des Geldes nicht berücksichtigt wird. Eine empirische Untersuchung, auf die wir in 3.3 ausführlich eingegangen sind, ist [Lim94]. [Jon94] gibt einen kurzen Überblick von Erfahrungswerten zur Rendite für verschiedene wiederverwendbare Artefakte.

Eine aktuell häufig zitierte Referenz für die Investitionsanalyse bei Produktlinien ist [Wit96]. Neben der Frage, wie eine Produktlinie spezifiziert werden sollte, behandelt der Autor Bewertungsmethoden für Investitionen. Zur Behandlung von Unsicherheit werden hier neben der Kapitalwertmethode Entscheidungsbäume propagiert. Das Problem der Bestimmung des Diskontierungszinssatzes wird nicht behandelt.

Eine sehr umfassende – ebenfalls häufig referenzierte – Untersuchung der Methoden zur Bewertung von Investitionen in wiederverwendbare Software ist [FFF97]. Zwei bekannte Methoden werden ausführlich referiert: Kapitalwertbasierte Methoden einschließlich Entscheidungsbäume und Optionstheorie inklusive Realoptionen (engl.: real options). Das vorgeschlagene Verfahren für die Bestimmung des Diskontierungszinssatzes setzt allerdings voraus, daß die notwendigen Parameter[51] für das jeweils zu bewertende Projekt zur Verfügung stehen. Davon kann jedoch in der Regel nicht ausgegangen werden ([BrM95], 183; siehe auch 2.4.4). Das zugrundeliegende Modell des Entwicklungsprozesses hat darüber hinaus folgende Mängel: (1) die Wartung wird nicht modelliert; (2) es werden externe Geldflüsse vorausgesetzt, die jedoch bei interner Wiederverwendung nicht entstehen; (3) es können keine Systeme abgebildet werden, die aus neuentwickelten und wiederverwendeten Komponenten zusammengesetzt sind.

Ein Vorgehen bei der Erstellung eines ökonomischen Modells wird in [WiB98] vorgeschlagen. Basierend auf der Annahme, daß die Entwicklung wiederverwendbarer Software als Investition zu betrachten sei, werden detailliert mögliche Schritte bei der Kosten-Nutzen-Analyse beschrieben.

Drei Veröffentlichungen [BSa97, Lim98, Wil99] analysieren im Überblick verschiedene ältere ökonomische Modelle, wobei die Auswahl der Modelle im Wesentlichen übereinstimmt.

Mit den ökonomischen Aspekten der Entwicklung von Produktlinien setzt sich [Sci99] auseinander. Das nur oberflächlich beschriebene empfohlene Vorgehen berücksichtigt weder den Zeitwert des Geldes noch die Unsicherheit der Investition.

Von einem Experiment mit der Entwicklung eines Frameworks für Telekommunikationsanwendungen berichtet [MSR99]. Auch hier werden bei der ökonomischen Analyse weder der Zeitwert des Geldes noch die Unsicherheit der Investition berücksichtigt.

---

[51] Unter anderem das Standardmaß für Risiko bei Geldanlagen β, das im verwendeten Capital Asset Pricing Model benötigt wird.

Die Autoren von [MFG00] präsentieren ein Modell für die ökonomische Bewertung von Software-Wiederverwendung, für das sie beanspruchen, daß es alle bisherigen Modelle integriert. Trotz der eher oberflächlichen Beschreibung werden zumindest zwei Mängel offensichtlich: der Diskontierungsfaktor wird als bekannt vorausgesetzt, was in vielen Fällen nicht zulässig ist, und die Unsicherheit der Investition wird nicht explizit berücksichtigt.

Einige jüngere Veröffentlichungen konzentrieren sich auf die *strategischen Aspekte der Software-Wiederverwendung*. Zunächst wird in [MeL95] die strategische Ausrichtung einer Produktfamilie anhand einer detailliert ausgeführten Fallstudie bei einer Softwareproduktfirma untersucht. Im Zentrum steht die Analyse der Produktfamilie mit Hilfe sog. 'Product Family Maps'. Wesentliches Ergebnis ist die zentrale Bedeutung einer ausgewogenen technischen Strategie für den wirtschaftlichen Erfolg des Unternehmens, wofür eine genaue Planung der Evolution der Architektur notwendig ist. Die Autoren präsentieren eine erhebliche Menge technischer Detailinformation, gehen aber auf die Architektur des Systems nur oberflächlich ein, obwohl ihre große strategische Bedeutung betont wird.

Die Bedeutung einer Einbeziehung von Software-Wiederverwendung in die Überlegungen zur Unternehmensstrategie streicht [Lim98a] heraus. Der Autor gibt einen allgemeinen Überblick der Probleme, die bei der Entwicklung einer Strategie gelöst werden müssen. Die Darstellung ist jedoch wenig spezifisch für die Entwicklung wiederverwendbarer Software.

In [FFa99] geben die Autoren einen Überblick über Standardinstrumente der strategischen und ökonomischen Analyse. Trotz des Anspruchs ist die Betrachtung wenig spezifisch für die Entwicklung von Frameworks; so wird z.B. die strategische Bedeutung der Architektur nicht untersucht. Bei der vorgeschlagenen Methode der Bewertung bleibt die Frage der richtigen, an die Unsicherheit angepaßten Wahl des Diskontierungssatzes unbeantwortet (siehe auch Anmerkungen zu [FFF97]).

3.5 Positionierung der Arbeit

3.5.1 Forschungsbedarf

Aus dem in diesem Kapitel erhobenen Stand der Forschung auf dem Gebiet der Software-Wiederverwendung ergibt sich der Forschungsbedarf, den wir im Folgenden zusammenfassen.

Die bisherigen Arbeiten konzentrieren sich größtenteils nicht auf ***betriebliche Informationssysteme***. Zu anderen Arten von Software bestehen jedoch große Unterschiede, insbesondere zu den häufig im Zentrum stehenden eingebetteten Systemen (vgl. 2.5). Daher ist es notwendig, die Entwicklung wiederverwendbarer Software in diesem Kontext spezifisch zu untersuchen, um Besonderheiten und eventuelle Unterschiede zu anderen Softwarearten herauszufinden.

Das ***empirische Fundament*** der Forschung ist schwach; vorhandene empirische Daten sind größtenteils qualitativer Art. Die grundlegenden Annahmen sind unzureichend, vielfach gar nicht, untermauert. Besonders bei der zentralen Annahme des großen ökonomischen Nutzens ist dies ein bedeutendes Manko. Auch die Gewichtung der Vor- und Nachteile sowie der Hindernisse leidet darunter. Das aktuelle Vorgehen ist ebenfalls unzureichend untersucht; dies gilt in besonderem Maß für die Architektur wiederverwendbarer Software. Empirische Untersuchungen, die diese Defizite beseitigen, sind notwendig.

Auf dem Gebiet der Softwaretechnik ist zunächst die technische Architektur ein zentrales Thema der Entwicklung wiederverwendbarer Software. Das zeigt sich an der regen Forschungstätigkeit auf diesem Gebiet und in den entsprechenden Veröffentlichungen. Diese Einschätzung wird von verschiedenen Beiträgen zum Zustand des Gebiets bestätigt [Bro95, ZBB99, SDD99, Lat99]. Eine Lücke besteht bei den ***Verfahren zur Zerlegung in Komponenten***, die entweder über eine grobe Gliederung des Systems in Schichten nicht hinausgehen oder auf einer zu feinen Granularitätsebene ansetzen. In engem Zusammenhang mit Komponenten stehen die zahlreichen Methoden zur Trennung der Zuständigkeiten. Sie konzentrieren sich auf – oftmals aufwendige – Verfahren zu deren technischer Realisierung und lassen die Frage der Gewichtung der Aufgaben[52] außer acht. Zudem verletzen sie teilweise das Geheimnisprinzip. Generell wird die Frage vernachlässigt, wie das Zusammenspiel der Komponenten bei gleichzeitiger Trennung der Zuständigkeiten realisiert werden soll. Ein durchgängiges Verfahren für die systematische Bestimmung der technisch und ökonomisch sinnvollen ***Variabilität*** in Spezifikation und Konstruktion steht darüber hinaus nicht zur Verfügung. Forschungsbedarf besteht auch hinsichtlich des ***Wiederverwendbarkeitsbegriffs:*** Es gibt keine systematische Analyse des Zusammenhangs zwischen der Ausprägung der Wiederverwendbarkeit und der Investition. Zudem sind die Wechselwirkungen mit allgemeinen Entwicklungszielen unzureichend untersucht.

Insgesamt existiert kein umfassendes Verfahren für die ***Bewertung von Projektvorhaben***, das softwaretechnische und ökonomische Kriterien in angemessener Weise verbindet. Die Methoden zur ***ökonomischen Bewertung*** der Entwicklung wiederverwendbarer Software wurden in den letzten Jahren immer mehr verfeinert. Allerdings hat man dabei Konzepte übertragen, die in anderem Kontext sinnvoll, hier jedoch nicht praktikabel sind. Dazu zählt insbesondere das Capital Asset Pricing Model, das bei der Anwendung auf einzelne Projekte große Schwierigkeiten aufwirft, besonders, wenn das jeweilige Unternehmen nicht selbst börsennotiert ist (vgl. [BrM95], 181ff, 205 und [CKM94], 330ff; siehe auch 2.4.4). Darüber hinaus ist das zugrundeliegende ***ökonomische Modell*** der Entwicklung wiederverwendbarer Software oft unzureichend, weil wichtige Aspekte nicht berücksichtigt werden. Daher besteht der Bedarf nach einem praktikablen, Zeitwert des Geldes und Investitionsunsicherheit berücksichtigenden Bewertungsverfahren auf Basis einer spezifischen und umfassenden Modellierung.

[52] Die Zuständigkeiten beziehen sich auf Aufgaben (vgl. 2.3.4)

Zunehmend wird die Notwendigkeit einer Einbindung der Wiederverwendung in die Strategie erkannt [ZAD97, Kar98, Zan99]. Die Veröffentlichungen zur ***strategischen Relevanz*** der Wiederverwendung sind jedoch unsystematisch und wenig spezifisch: es fehlt an einer systematischen Analyse des Nutzens und seiner Relevanz für die Erreichung unterschiedlicher strategischer Ziele. Außerdem werden die Unterschiede zwischen Produkt- und Projektgeschäft nicht herausgearbeitet.

3.5.2 Einordnung und Beitrag der Arbeit

Diese Arbeit macht wissenschaftlich Beiträge zu verschiedenen Teilgebieten der Forschung im Umfeld der Software-Wiederverwendung.

Die Fokussierung auf ***betriebliche Informationssysteme*** hilft, die dort bestehende Lücke zu schließen. Unsere Ergebnisse sind daher zunächst im Kontext betrieblicher Informationssysteme relevant; einige grundlegendere Ergebnisse können allerdings auf andere Bereiche übertragen werden, insbesondere das ReValue-Modell zur ökonomischen Bewertung der Investition in wiederverwendbare Software (s.u.).

Wir stellen unterschiedliche empirische Untersuchungen in der industriellen Praxis an, um das ***empirische Fundament*** der Forschung zu verbessern. Quantitative Daten erheben wir zum aktuellen Vorgehen, zur Einschätzung der Wiederverwendung, notwendigen Maßnahmen und ökonomischen Aspekten. So ist es möglich, grundlegende Annahmen zu überprüfen, Vor- und – die bisher vernachlässigten – Nachteile zu gewichten und gegeneinander abzuwägen. Auch die Hindernisse und notwendigen Maßnahmen priorisieren wir auf Basis der erhobenen Daten. Im Rahmen der Möglichkeiten sammeln wir Daten zum ökonomischen Nutzen. Aus den erhobenen Daten leiten wir allgemeine Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software ab. In Fallstudien erheben wir – größtenteils qualitative – Daten über Projekte zur Entwicklung wiederverwendbarer Software in der industriellen Praxis; besondere Aufmerksamkeit gilt hier der Architektur. Davon leiten wir in detaillierter Form softwaretechnische Erfolgsfaktoren für solche Projekte ab.

Da ausreichend detaillierte und aussagekräftige Daten zum ökonomischen Nutzen nicht verfügbar sind, entwickeln wir ein detailliertes und umfassendes ***ökonomisches Modell*** der Entwicklung wiederverwendbarer Software: das ReValue-Modell. Dieses Modell ist die Basis für die ökonomische Bewertung. Mit Hilfe der Monte-Carlo-Methode simulieren wir auf Basis des ReValue-Modells die Nutzung wiederverwendbarer Software und ermitteln so für unterschiedliche Szenarien den ökonomischen Nutzen. Wir bewerten ihn mit der internen Rendite und berücksichtigen die Unsicherheit der Investition durch Bestimmung eines einseitigen Konfidenzintervalls. So stellen wir ein praktikables Verfahren für die ***ökonomische Bewertung*** zur Verfügung, das die Unsicherheit der Investition berücksichtigt und dabei ohne schwer zu ermittelnde Parameter auskommt.

Auf Basis dieser Ergebnisse entwickeln wir Leitlinien für die Entwicklung wiederverwendbarer Software, deren Anwendung dem Erfolg in softwaretechnischer und ökonomischer Hinsicht dient.

Zunächst stellen wir einen verfeinerten ***Wiederverwendbarkeitsbegriff*** vor, der die bestehenden Defizite beseitigt: Ausgehend von einer systematischen Analyse der Merkmale der Wiederverwendbarkeit untersuchen wir den Zusammenhang zwischen Grad der Wiederverwendbarkeit und notwendiger Investition. Anschließend betrachten wir Wiederverwendbarkeit im Kontext des Entwicklungsprozesses und bestimmen ihre Wechselwirkungen mit allgemeinen Entwicklungszielen.

Daraufhin bestimmen wir mit Hilfe einer systematischen Nutzenanalyse die bisher unzureichend untersuchte ***strategische Relevanz*** der Wiederverwendung. Wir unterscheiden dabei zwischen operativ und strategisch relevantem Nutzen. Für die verschiedenen Ausprägungen des strategisch relevanten Nutzens überprüfen wir – getrennt nach Produkt- und Projektgeschäft – den Beitrag zur Erreichung strategischer Ziele.

Aus der Analyse mit Hilfe des ReValue-Modells leiten wir ökonomische Kriterien für die ***Bewertung von Projektvorhaben*** ab, die wir um softwaretechnische Kriterien ergänzen, die sich aus den oben genannten softwaretechnischen Erfolgsfaktoren ergeben. Dadurch stellen wir ein im Sinne des Forschungsbedarfs umfassendes Verfahren bereit. Die ökonomischen Kriterien unterscheiden zwischen operativ und strategisch relevanten Auswirkungen wägen diese gegeneinander ab.

Wir stellen ein neuartiges ***Verfahren zur Zerlegung in Komponenten*** vor, das auf einer mittleren Granularitätsebene eines Systems ansetzt und die Trennung der Zuständigkeiten mit Hilfe der Software-Kategorien realisiert. Dies trägt zur Schließung der bestehenden Lücke bei.

3.5.3 Empirisch zu überprüfende Hypothesen

Eine Reihe der sich aus der Literatur ergebenden Forschungsfragen fassen wir zu Hypothesen zusammen, die wir in Kapitel 4 bis Kapitel 6 überprüfen. Wir geben sie im Folgenden nach Themengebieten geordnet und mit einem Kürzel versehen wieder:

Hypothesen zum aktuellen Vorgehen

(1) Wiederverwendung findet heute überwiegend in der Programmierung statt [Code].
(2) Der Schwerpunkt liegt auf Ad-Hoc-Wiederverwendung (Entwickler selbst oder innerhalb der Gruppe) [AdHoc].
(3) Eigene Entwicklung ist für die Entwickler die wichtigste Quelle für wiederverwendbare Software [EigEntw].
(4) Es besteht ein Mangel an systematisch wiederverwendbarer Software [MangelSW].

Hypothesen zur Einschätzung der Wiederverwendung

(5) Die Entwickler sind grundsätzlich negativ zur Wiederverwendung eingestellt [NegEinst].

(6) Die Erhöhung der Entwicklungsproduktivität ist der wichtigste Vorteil von Wiederverwendung [ErhProd].
(7) Verkürzung der Entwicklungszeit ist ein wichtiger Vorteil der Wiederverwendung [VerkZeit].
(8) Senkung der Wartungslast ist ein wichtiger Vorteil der Wiederverwendung [SenkWart].
(9) Erhöhung der Qualität ist ein wichtiger Vorteil der Wiederverwendung [QualiErh].
(10) Wiederverwendung ist grundsätzlich möglich; es gibt keine inhärenten, grundsätzlich nicht lösbaren Probleme (z.B. schlechtere Performance, suboptimale Lösungen) [GrundsProb].
(11) Wichtigster Nachteil ist der Zusatzaufwand für die Herstellung wiederverwendbarer Software [ZusatzAuf].

Hypothesen zu Hindernissen und notwendigen Maßnahmen

(12) Es werden keine ausreichenden Ressourcen für die Herstellung wiederverwendbarer Software zur Verfügung gestellt [Ress].
(13) Das Not-invented-here-Syndrom spielt eine große Rolle [NIH].
(14) Der Zusatzaufwand für Dokumentation und Wartung schreckt von der Entwicklung wiederverwendbarer Software ab (gegenläufiger Anreiz) [Abschreck].
(15) Mangelnde Information und mangelnde Tool-Unterstützung sind wichtige Hindernisse [InfoTool].
(16) Wiederverwendung ist momentan im Prozess nicht ausreichend verankert [Prozess].
(17) Die Architektur hat großen Einfluß auf den Erfolg der Wiederverwendung [Archi].
(18) Objektorientierte Entwicklung ist keine Voraussetzung für den Erfolg [OO].
(19) Abbau von Hindernissen ist wichtiger als die Schaffung positiver Anreize [HindAbb].
(20) Finanzielle Anreize haben hohe Bedeutung [FinAnr].
(21) Die Unterstützung der Wiederverwendung durch eine zentrale Organisationseinheit ist notwendig [ZentrOrg].

Hypothesen zu ökonomischen Aspekten

(22) Wiederverwendung hat generell großes ökonomisches Potential aufgrund von Produktivitätssteigerungen [GroßesPot].
(23) Die Kosten der Herstellung wiederverwendbarer Software sollten als Investition betrachtet und bewertet werden [Invest].
(24) Es gibt eine große Varianz in der erzielbaren Rendite [Varianz].

Eine tabellarische Übersicht über die Hypothesen und die Referenzen, die jeweils dafür oder dagegen sprechen bzw. die Hypothese unterstützen, gibt Tabelle 3-2. Unterstützende Referenzen sind solche, die indirekt die Aussage stützen, z.B. im Falle der Hypothese [Varianz] mehrere unterschiedliche Aussagen zur Rendite der Entwicklung wiederverwendbarer Software. Neben vielen Hypothesen, die übereinstimmend von mehreren Quellen gestützt werden,

sind darunter auch einige ([NegEinst], [NIH], [OO], [FinAnr], [GroßesPot], [Varianz]), zu denen es widersprüchliche Positionen gibt.

Nr.	Hypothese	Dafür	Dagegen	Unterstützende Referenz
1	[Code]	[ZBB99]		[BaB91]
2	[AdHoc]	[Kar95], 5 [ZAD97] [Shm99]		
3	[EigEntw]	[Kar95], 5f, 60 [CaE95], 2		
4	[MangelSW]	[You93], 181 [Som96], 396 [GAO95] [Tra95], 137 [JGJ97], 8		[McC97], xix
5	[NegEinst]	[McC97], 19	[JGJ97], 8f	
6	[ErhProd]	[Kar95], 10 [Sam97], 12 [McC97], xxi; [Lim98], 102 [Den91], 17		
7	[VerkZeit]	[JGJ97], 5 [Kar95], 10 [McC97], xxi [Lim98], 102 [Sam97], 13		
8	[SenkWart]	[Kar95], 5 [Sam97], 13		
9	[QualiErh]	([Kar95], 11) ([Tra95], xi) [Sam97], 11ff		
10	[GrundsProb]	[JGJ97], 6f [Kar95], 4 [McC97], 19 [GrW95]		
11	[ZusatzAuf]	[BBo91] [Tra95], xi [Joo94] [CaE95], 4ff		[JGJ97], 10
12	[Ress]	[BBo91] [Tra95], xi [Joo94] [CaE95], 4ff		[Sam97], 16

13	[NIH]	[Kar95], 6 [Sam97], 15 [Tra95], 126 [McC97], 19 [GrW95] [Shm99] [KeM99]	[FrF95]	
14	[Abschreck]	[CaE95], 5		
15	[InfoTool]	[McC97], 19		
16	[Prozess]	([McC97], xxi		
17	[Archi]	[Bro95], 224 [JGJ97], 10 [Lat99] [SDD99]		
18	[OO]	[KRi88], 67 [Sam97], 17 [FrF95]	[SoS99], 5f [SBF96]	
19	[HindAbb]	[Kar95], 5		
20	[FinAnr]	[Fra94]	[JGJ97], 404 [JGJ97a] [Gol95], 265	
21	[ZentrOrg]	[JGJ97], 371		
22	[GroßesPot]	[Gol95], 204 [Den91], 17 [Kar95], 8, 11 [Tra95], 132f	[MaR92]	[Cox90] [GrW95] [McC97], 3 [JGJ97], 4ff [SoS99], 5
23	[Invest]	[ZAD97] [Shm99] [BaB91] [JGJ97], 23		[Kar95], 5 [Sam97], 12
24	[Varianz]	[Tra95], 132f		[Gol95], 204 [Kar95], 11 [JGJ97], 23 [MSG96]

Tabelle 3-2: Zusammenfassung empirisch zu überprüfender Hypothesen

3.5.4 Beiträge der einzelnen Kapitel

Der Stärkung des empirischen Fundaments dienen die Fallstudien in Kapitel 4 und Kapitel 5. In *Kapitel 4* liegt der Schwerpunkt auf den allgemeinen Aspekten: aktuelles Vorgehen, Rahmenbedingungen und Handlungsbedarf. Auf Basis der erhobenen Daten wird der Großteil der Hypothesen überprüft. Außerdem

leiten wir allgemeine Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software ab. In *Kapitel 5* stehen die softwaretechnischen Aspekte im Zentrum. Anhand der Daten ergänzen wir die Hypothesenprüfung und formulieren softwaretechnische Erfolgsfaktoren.

Die ökonomischen Aspekte werden in *Kapitel 6* behandelt. Wir entwickeln ein ökonomisches Modell der Entwicklung wiederverwendbarer Software und führen mit Hilfe von Monte-Carlo-Simulationen eine ökonomische Bewertung verschiedener Szenarien durch. Die Ergebnisse sind die Grundlage für die Überprüfung der ökonomischen Hypothesen.

In *Kapitel 7* schließlich werden die Ergebnisse der empirischen und simulativen Untersuchungen synthetisiert. Zunächst stellen wir die Rahmenbedingungen für die Entwicklung wiederverwendbarer Software auf Basis einer Zusammenfassung der überprüften Hypothesen dar. Dann entwickeln wir einen verfeinerten Wiederverwendbarkeitsbegriff, indem wir die Merkmale wiederverwendbarer Software im Zusammenhang analysieren. Weiterhin geben wir Leitlinien für die softwaretechnische und ökonomische Bewertung von Projektvorhaben an, die auf den Ergebnissen der vorangegangenen Kapitel basieren. Darauf formulieren wir softwaretechnische Leitlinien. Diese umfassen im Wesentlichen das Konzept der kontrollierten Variabilität, eine neues Verfahren für die Zerlegung in Komponenten und eine Formalisierung und Verallgemeinerung der Software-Kategorien. Schließlich geben wir ökonomische und strategische Leitlinien an, indem wir Investition und Nutzen analysieren und strategische Optionen für unterschiedliche Rahmenbedingungen ermitteln.

Kapitel 4

Empirische Untersuchung zum Status quo der Wiederverwendung

Ziel dieses Kapitels ist, den Status quo der Wiederverwendung in der industriellen Praxis empirisch zu erheben und Schlüsse für das Vorgehen zu ziehen.

Das Kapitel stellt die Ergebnisse einer Befragung von 55 Entwicklern zum Status quo der Wiederverwendung bei einem internationalen Softwareproduktunternehmen vor und analysiert sie. Anhand der Ergebnisse überprüfen wir einen großen Teil der in Kapitel 3.5.3 formulierten Hypothesen. Wir interpretieren daraufhin die Ergebnisse im Zusammenhang und leiten verallgemeinernd Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software ab.

Inhalt: **Seite**

4.1 Einleitung

In diesem Kapitel wird eine bei einem internationalen Softwareproduktherstel-ler durchgeführte empirische Untersuchung zu verschiedenen Aspekten der Software-Wiederverwendung beschrieben. Sie dient dazu, das empirische Fundament hinsichtlich der Rahmenbedingungen und grundlegenden Annahmen in der industriellen Praxis zu stärken.

In der Einleitung stellen zunächst wir das Unternehmen und seinen Markt vor und beschreiben das Umfeld für Wiederverwendung, das methodische Vorgehen bei der Untersuchung und den Stand der Forschung sowie unseren eigenen Beitrag.

In den folgenden Unterkapiteln stellen wir die Ergebnisse der Untersuchung in verschiedenen Bereichen vor: Aktuelles Vorgehen, Einschätzung der Wiederverwendung, Hindernisse, notwendige Maßnahmen und ökonomische Aspekte. Im Rahmen der Analyse der Ergebnisse überprüfen wir die in 3.5.3 formulierten Hypothesen, interpretieren die Ergebnisse im Zusammenhang und leiten allgemeine Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software ab. Schließlich fassen wir das Kapitel kurz zusammen.

4.1.1 Das Unternehmen und sein Markt

Das betrachtete Unternehmen stellt Produkte her, die der Domäne betrieblicher Informationssysteme zuzuordnen sind. Sie werden in großen Firmen aus Industrie und Handel eingesetzt. Die Produkte werden an mehreren Standorten in verschiedenen Ländern entwickelt und weltweit vertrieben. Der Wettbewerb ist hart; in unterschiedlichen Teilmärkten hat das Unternehmen jeweils mindestens zwei ernstzunehmende Konkurrenten um die Marktführerschaft.

In den vergangenen Jahren gingen mehrere Innovationswellen über den von dem untersuchten Unternehmen bedienten Markt für betriebliche Informationssysteme hinweg. Die letzte davon wurde durch den Siegeszug des Internets ausgelöst, der sich durch das world wide web auch auf die Anforderungen der Unterstützung durch betriebliche Informationssysteme auswirkte. Elektronische Abwicklung des Handels über das Internet (e-commerce) und die Nutzung des Internets für alle betrieblichen Funktionen (e-business) waren die Ziele und Schlagworte der dadurch in Gang gesetzten Revolution. Viele neue Softwareanbieter drängten mit unterschiedlichen Produkten auf den Markt für Informationssysteme.

Das untersuchte Unternehmen reagierte auf die durch die Innovationswellen ausgelösten Marktentwicklungen, indem es so schnell wie möglich seinerseits Produkte auf den Markt brachte, die die neu entstandenen Anforderungen abdeckten. Dabei handelte es sich zum Teil um neue Produkte, zum Teil um spezialisierte Varianten bestehender Produkte, die dadurch zunehmend ausdifferenziert wurden. Da das untersuchte Unternehmen jeweils nicht der Vorreiter

am Markt war, sondern auf die dortigen Entwicklungen reagierte (fast follower-Prinzip), war Geschwindigkeit in der Entwicklung besonders wichtig.

Die Produkte haben die typischen, in 2.5 aufgeführten Merkmale betrieblicher Informationssysteme. Die wichtigsten seien hier nochmals kurz erwähnt: sie bauen auf einer Datenbank auf, haben einen Anwendungskern und eine graphische Benutzeroberfläche für den Endbenutzer; die Architektur ist eine Schichtenarchitektur. Große Anzahlen von Benutzern arbeiten in der Regel gleichzeitig mit den Systemen. Die Systeme stellen zahlreiche Schnittstellen (sog. Application Programming Interfaces oder APIs) für Nachbarsysteme zur Verfügung. Mehrere Betriebssysteme und Datenbanken stehen für die Kunden als Systemplattform zur Auswahl.

Die Entwicklung ist in drei Hierarchieebenen organisiert: Entwickler, Gruppenleiter und Abteilungsleiter. Daneben gibt es unterschiedliche Bereiche für die verschiedenen Produkte und Produktteile.

4.1.2 Umfeld der Wiederverwendung

Für die Interpretation der Ergebnisse der empirischen Untersuchung ist das Verständnis des Zusammenhangs wichtig, in dem sie erhoben wurden. Daher fassen wir im Folgenden zusammen, was das Umfeld der Wiederverwendung kennzeichnet.

Aus der oben beschriebenen Marktentwicklung folgt zunächst, daß die Komplexität der Produktpalette wegen der neuen Produkte und der Ausdifferenzierung der alten Produkte zunimmt. Durch die entstehenden Produktfamilien (vgl. 3.4.1) vergrößert sich das Potential für Wiederverwendung.

In Vorgesprächen ergaben sich weitere Erkenntnisse zum Umfeld, ebenso wie in der empirischen Untersuchung, da unter anderem einige Fragen zum Hintergrund der befragten Personen gestellt wurden:

- Die Entwickler verbringen wesentliche Anteile ihrer Arbeitszeit mit der Wartung und Weiterentwicklung vorhandener Produkte. Dadurch wird die Innovationskraft des Unternehmens verringert, denn diese Zeit geht der Entwicklung neuer Produkte verloren. Im Durchschnitt machen Wartung und Weiterentwicklung insgesamt etwa 40 Prozent der Arbeitszeit der befragten Entwickler aus. Der Grund dafür ist, daß einige Produkte bereits eine große Kundenbasis haben, die ständig neue Wünsche äußert.
- Geschwindigkeit der Entwicklung ist ein sehr wichtiges Ziel, da die Zeitspanne, die jeweils bis zur Markteinführung eines Produkts vergeht (time to market), möglichst gering sein soll.
- Es wird überwiegend prozedural programmiert; der Anteil objektorientierter Entwicklung nimmt jedoch zu: die Hälfte der Entwickler verwendet auch objektorientierte Programmiersprachen; bei diesen Entwicklern liegt der durchschnittliche Anteil objektorientierter Programmierung bei 42 Prozent.

Die – im Gegensatz zu vielen anderen Software-Produktherstellern – äußerst konsequent eingehaltene Schichtenarchitektur bedeutet ein erhebliches Maß an

dadurch bedingter systematischer Wiederverwendung (vgl. 2.5.3). Diese Art der Wiederverwendung weist zwei Besonderheiten auf:

- Vorrangiges Ziel ist die Trennung der Zuständigkeiten und die dadurch erreichbare Beherrschbarkeit des Systems. Ohne Zerlegung in Schichten wäre das Systems aufgrund der Mehrfachentwicklung nicht nur erheblich teurer, sondern im Grenzfall ein nicht mehr wartbarer Monolith. Insofern spielen die Qualitätsziele im Vergleich zu den reinen Kostenzielen eine wichtige Rolle.
- Im allgemeinen Verständnis handelt es sich deshalb bei der Zerlegung in Schichten primär um eine Maßnahme, die eine gute Architektur und damit die Beherrschbarkeit des Systems sicherstellt. Der Gedanke der Wiederverwendung tritt in den Hintergrund.

Dementsprechend umfaßt der im untersuchten Unternehmen gebräuchliche Begriff der Wiederverwendung nicht die sich aus der Schichtenarchitektur ergebende Wiederverwendung von Software, die Grundfunktionen für Datenverwaltung und graphische Benutzeroberfläche, Querschnittsfunktionen und Datentypen bereitstellt. Dies muß bei der Interpretation der Ergebnisse berücksichtigt werden. Diese Definition liegt der Untersuchung zugrunde, deren Ergebnisse wir hier präsentieren.

Wiederverwendung im soeben beschriebenen Sinne findet bei dem untersuchten Unternehmen bereits in nennenswertem Umfang statt. Systematische Wiederverwendung ist dabei beschränkt auf die Kategorien Null-Software und T-Software. Die Wiederverwendbarmachung erfolgt sowohl a priori als auch a posteriori. Die Bereitstellung der wiederverwendbaren Software erfolgt in Form einer Programmbibliothek, der sog. *Reuse Library*, auf die über die vernetzte Entwicklungsumgebung zugegriffen werden kann. Die Reuse Library wird von einer Gruppe von Entwicklern betreut, die folgende Aufgaben versehen:

- Identifizierung von Software-Bausteinen, die sich zur Wiederverwendung eignen
- Aufbau, Verwaltung und Wartung der Reuse Library
- Unterstützung von größeren Projekten zur Entwicklung wiederverwendbarer Software, um eine Überlastung der ursprünglichen Entwickler zu vermeiden.

Auf der Ebene des Anwendungskerns für die verschiedenen Anwendungen findet in begrenztem Umfang Ad-Hoc-Wiederverwendung, jedoch keine systematische Wiederverwendung statt.

4.1.3 Methodisches Vorgehen bei der Untersuchung

Die Untersuchung wurde als Fallstudie durchgeführt. Daher gehen wir zunächst auf die Ziele und theoretischen Grundlagen dieser Methode ein. Anschließend beschreiben wir, wie die Untersuchung im einzelnen durchgeführt wurde und behandeln die Frage, inwiefern die Ergebnisse repräsentativ sind.

Fallstudie: Ziele und theoretische Grundlagen

Die Fallstudie ist eine bewährte Methode zur empirischen Untersuchung komplexer Phänomene. Zur Bestimmung des Begriffs stützen wir uns auf folgende Definition ([Yin90], 23):

> *Eine Fallstudie ist eine empirische Untersuchung, die ein zeitgenössisches Phänomen innerhalb seines realen Kontexts untersucht, bei dem die Begrenzung zum Kontext nicht klar bestimmbar ist. Die Untersuchung verwendet vielfältige Quellen.*

Fallstudien werden als Forschungsstrategie im allgemeinen gewählt, um Fragen nach dem "Wie" oder "Warum" zu klären, wenn der Forscher nur in geringem Umfang Kontrolle über die Ereignisse ausüben kann, oder wenn ein zeitgenössisches Phänomen in einem realen Kontext untersucht wird ([Yin90], 13). Fallstudien werden in vielen Umgebungen als Forschungsmethode angewendet, unter anderem in Organisations- und Management-Untersuchungen ([Yin90], 13). In der Regel werden – gemäß o.g. Definition – unterschiedliche Arten der Datenerhebung eingesetzt, darunter Archivrecherchen, Interviews mit und ohne Fragebögen und Beobachtungen [Eis89]. Eine besonders wichtige Methode der Datenerhebung ist das Fragebogen-basierte Interview (vgl. [Att95], 160ff). Bei der Gewichtung der Antworten ist der Unterschied zwischen offenen und geschlossenen Fragen zu berücksichtigen: während geschlossene Fragen vom Befragten lediglich verlangen, etwas wiederzuerkennen, muß er sich bei offenen Fragen an etwas erinnern. Letzteres ist schwieriger; man erhält daher weniger Antworten, die entsprechend höher zu gewichten sind (vgl. [Att95], 183).

Fallstudien gelten als besonders geeignet dafür, in frühen Stadien der Erforschung eingesetzt zu werden oder um eine frische Perspektive auf ein bereits erforschtes Gebiet zu ermöglichen [Eis89]. Letzterer Grund war bei der vorliegenden Arbeit ausschlaggebend für die Wahl der Fallstudie. Auf Basis von Fallstudien kann induktive Theoriebildung erfolgen [Eis89][53]; dies gilt insbesondere für den von uns verfolgten Einzelfall-Ansatz [Hei95]. In der Literatur zur Software-Wiederverwendung wird er regelmäßig angewendet. Das zeigt sich daran, daß es eine Reihe von Veröffentlichungen gibt, die Fallstudien bei einer einzigen Firma behandeln: [Joo94, Gri94, GrW95, Lim94, MeL95, MCG96, Ill99].[54]

Durchführung der Untersuchung

Die Untersuchung wurde, vom Vorstand des Unternehmens unterstützt, im Herbst des Jahres 1999 durchgeführt. Ihren Kern bildeten Fragebogen-basierte Interviews mit Entwicklern. Um sie vorzubereiten, wurden zunächst Hypothesen durch Studium der Literatur (siehe 3.5.3) gebildet. Auf deren Basis erarbeiteten wir eine erste Arbeitsversion des Fragebogens. In einer ersten Serie von Interviews mit sechs verschiedenen Gesprächspartnern aus verschiede-

[53] Eisenhardt führt eine Reihe von Beispielen für dieses Vorgehen an.

[54] Siehe auch 3.3.

nen Bereichen und organisatorischen Ebenen der Entwicklung wurde diese erste Version des Fragebogens diskutiert. Dabei wurden sowohl Inhalt und Struktur als auch Terminologie besprochen. Letzteres war wichtig, weil die im Unternehmen gebräuchlichen Begriffe auf die in dieser Arbeit verwendeten abgebildet werden mußten. Zudem dienten die Interviews dem Verständnis der in 4.1.2 beschriebenen Rahmenbedingungen.

Diese vorbereitenden Interviews wurden auf der Basis eines Gesprächsleitfadens durchgeführt. Methodisch sind sie daher als teilstrukturierte, qualitative Befragung einzuordnen ([Att95], 162); als Vorbereitung für die sich anschließenden Interviews ist ein solches Vorgehen unabdingbar ([Att95], 163). In ihrem Verlauf wurde die Arbeitsversion des Fragebogens mehrfach iterativ überarbeitet; mit jedem der Gesprächspartner wurden jeweils mindestens zwei Gespräche geführt. Am Ende der ersten Interviewserie lag der endgültige Fragebogen (siehe Anhang) vor; außerdem waren die wesentlichen Rahmenbedingungen erhoben (s.o.).

Daran schloß sich eine Serie von Interviews mit 55 Entwicklern aus allen Bereichen der Entwicklung und von allen drei organisatorischen Ebenen an. Diese Interviews basierten auf dem endgültigen Fragebogen, der in drei an die jeweilige organisatorische Ebene angepaßten Versionen verwendet wurde. Die Interviews wurden über einen Zeitraum von ca. acht Wochen zwischen Ende Oktober und Ende Dezember 1999 vom Autor persönlich vor Ort geführt. Sie konzentrierten sich auf den Standort der Zentrale, an dem alle Bereiche der Entwicklung vertreten waren; dies half, die Kosten der Befragung in vertretbarem Rahmen zu halten. Methodisch ist sie als stark strukturierte Befragung einzuordnen ([Att95], 163), die quantitative und qualitative Elemente kombiniert.

Die Verteilung der Stichprobe auf die drei organisatorischen Ebenen (vgl. 4.1.1) ist wie folgt: neben neun Abteilungsleitern wurden vierzehn Gruppenleiter und 32 Entwickler befragt. Die Abteilungsleiter wurden alle einzeln befragt; für die Gruppenleiter und Entwickler wurde teilweise als Vorgehen die Gruppenbefragung (vgl. [Att95], 174) gewählt: mehrere Befragte beantworteten den Fragebogen schriftlich unter Anwesenheit des Autors, der für Erläuterungen zur Verfügung stand. Der Ablauf war jeweils wie folgt: zunächst erläuterte der Autor die Ziele der Befragung und den Aufbau des Fragebogens und erhob die individuellen Tätigkeiten und Erfahrungen mit Wiederverwendung. Dann wurde die Befragung anhand des Fragebogens durchgeführt. Schließlich wurden je nach den Befragten zur Verfügung stehender Zeit Fragen von jeweils besonderem Interesse vertieft.

Aufbau des Fragebogens

Der Fragebogen ist in der Fassung für Abteilungsleiter im Anhang abgedruckt. Einen Überblick über den Aufbau gibt Abbildung 4-1. Die Formulierungen wurden an den im Unternehmen üblichen Sprachgebrauch angepaßt; daher stimmen sie teilweise nicht mit den in dieser Arbeit bevorzugten überein. Für wiederverwendbare Artefakte wird beispielsweise durchgängig der kürzere englische Begriff *Reusables* verwendet. Auch Wiederverwendung wird meist in der englischen Übersetzung als *Reuse* bezeichnet.

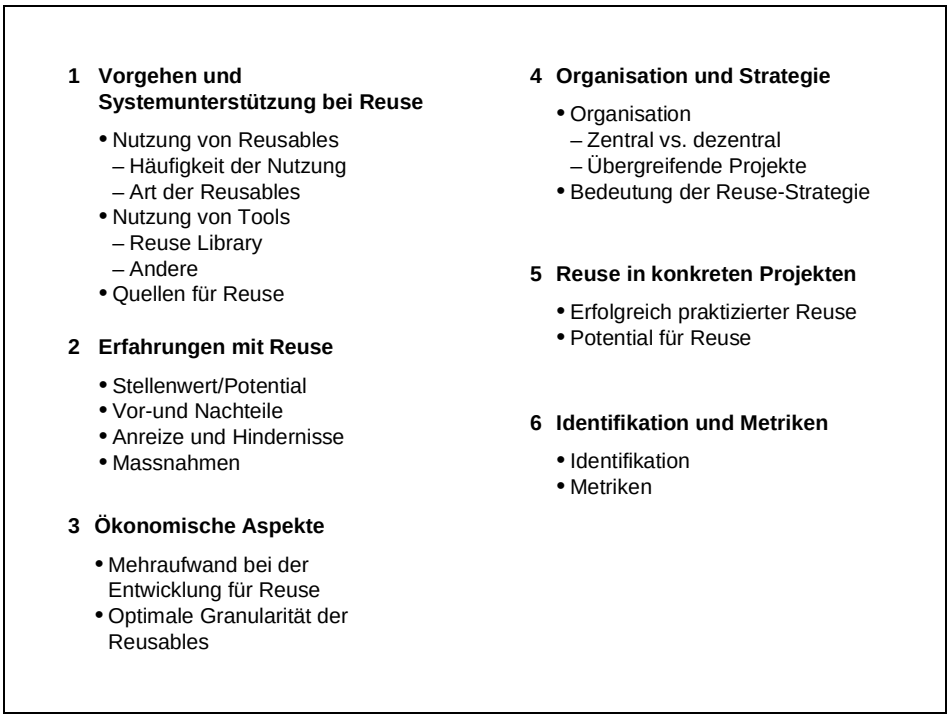

Abbildung 4-1: Aufbau des Fragebogens

Auswertung der Daten und Repräsentativität der Ergebnisse

Die Auswertung der Fragebögen erfolgte mit Hilfe einer Datenbank, in die die verschiedenen Ergebnisse eingetragen wurden. Der Fragebogen erhebt ca. 200 Datenpunkte, gemessen als Datenbankfelder. Die in die Datenbank eingegebenen Rohdaten wurden statistisch ausgewertet. Wo möglich und sinnvoll, wurden die Ergebnisse durch inferenzstatistische Berechnungen abgesichert. Dabei wurde das Konfidenzintervall für ein Konfidenzniveau von 95 Prozent bestimmt.

Bei der Analyse der Repräsenativität gehen wir in zwei Schritten vor. Zunächst erörtern wir die Frage, ob die Ergebnisse für die untersuchte Firma repräsentativ sind. Dafür sind die Ergebnisse der inferenzstatistischen Auswertungen ausschlaggebend. Es zeigt sich, daß die Anzahl der Befragten in den meisten Fällen hoch genug ist, um robuste Aussagen zu garantieren. Daher sind die Ergebnisse für die untersuchte Firma repräsentativ. Dies gilt für die in der Literatur verfügbaren Untersuchungen bei anderen Firmen nicht (vgl. 3.3).

Nun wenden wir uns der Frage zu, inwiefern die Ergebnisse auch für andere Firmen relevant sind. Dabei sind folgende Tatsachen von Bedeutung:

- Das untersuchte Unternehmen ist sehr erfolgreich, so daß es grundsätzlich als Vorbild für andere Firmen dienen kann. Dies wird dadurch unterstrichen, daß es sich um ein großes Unternehmen handelt, das dementsprechend einen großen Markt bedient und hartem Wettbewerb ausgesetzt ist.
- Das untersuchte Unternehmen operiert international, so daß ein zu großer Einfluß lokaler kultureller Besonderheiten ausgeschlossen ist. Da der Markt für Softwareprodukte ein globaler Markt ist (vgl. [Hoc99]), ist davon auszugehen, daß sich die Unterschiede zu anderen Ländern in Grenzen halten.

- Es bestehen bereits Erfahrungen mit systematischer Wiederverwendung, so daß differenzierte und kompetente Aussagen zu erwarten sind.

Zusätzlich deuten die eigene Erfahrung des Autors aus der Mitwirkung an dem in [Hoc99] beschriebenen Survey und Interviews mit verschiedenen anderen Softwarefirmen im Rahmen dieser Arbeit darauf hin, daß die hier dargestellten Ergebnisse im Rahmen unserer Interpretation repräsentativ sind. Wir halten daher eine Verallgemeinerung der Aussagen über die untersuchte Firma hinaus auf andere große Unternehmen grundsätzlich für vertretbar, wenn dabei die spezifischen Rahmenbedingungen berücksichtigt werden.

Bei der Verallgemeinerung wenden wir folgende Techniken an:

- Wir führen die beobachteten Phänomene argumentativ auf ein allgemeines zugrundeliegendes Prinzip zurück. In vielen Fällen stützen wir die Überlegungen durch Querbezüge zwischen unterschiedlichen erhobenen Daten, um die Konsistenz der Aussagen sicherzustellen.
- Wir widerlegen bestehende allgemeine Annahmen, indem wir ihnen widersprechende empirische Erkenntnisse für den untersuchten konkreten Einzelfall in der Art eines Widerspruchsbeweises werten.

4.1.4 Stand der Forschung und eigener Beitrag

Forschungsbedarf besteht nach der Analyse der Literatur zu Wiederverwendung in Kapitel 3 insbesondere hinsichtlich des empirischen Fundaments: die grundlegenden Annahmen sind unzureichend untermauert. Vielfach stehen gar keine empirischen Daten zur Verfügung; wenn Daten vorhandenen sind, so sind sie überwiegend qualitativer Natur. Das Wissen über das aktuelle Vorgehen ist deshalb überwiegend nicht gesichert; auch die Annahmen zu Vor- und Nachteilen und Hindernissen der Wiederverwendung sowie notwendigen Maßnahmen können deshalb nicht verifiziert und gewichtet werden. Darüber hinaus gibt es keine ausreichenden Daten zu den ökonomischen Aspekten und der Architektur wiederverwendbarer Software.

Diese Aussagen gelten insbesondere für die Fallstudien bei einzelnen Unternehmen [Joo94, Gri94, GrW95, Lim94, MeL95, MCG96, Ill99], die wir in 3.3 im Detail analysiert haben. Sie beschränken sich auf die USA und setzen sich größtenteils nicht mit der Entwicklung von betrieblichen Informationssystemen auseinander. Darüber hinaus haben sämtliche empirische Untersuchungen, also auch [FrF95, HaP97, MET99], keine breite Basis von Befragten, sondern beschränken sich auf einige wenige Personen, teilweise die Autoren selbst (vgl. 3.3).

Wir leisten in mehrerlei Hinsicht einen wissenschaftlichen Beitrag. Zunächst untersuchen wir die Wiederverwendung bei einer europäischen Firma, die betriebliche Informationssysteme herstellt. Unsere Ergebnisse beruhen auf der Befragung einer Gruppe von 55 Mitarbeitern aller Ebenen der Entwicklung. Was die Ergebnisse angeht, so stellen wir detaillierte quantitative Daten aus folgenden Bereichen vor und gewichten sie: aktuelles Vorgehen, Einschätzung der Wiederverwendung (Vor- und Nachteile), Hindernisse, notwendige Maßnahmen und ökonomisches Potential. Wir fassen die Ergebnisse in Form der Überprüfung der in 3.5.3 formulierten Hypothesen zusammen, interpretieren

sie im Zusammenhang und leiten allgemeine Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software ab.

Nicht ausreichend sind aufgrund unzureichender Daten die Ergebnisse im Bereich der Ökonomie. Wir ergänzen sie durch die in Kapitel 6 beschriebenen Ergebnisse von Monte-Carlo-Simulationen. Keine Daten erheben wir in diesem Kapitel zur Architektur; sie wird in Kapitel 5 behandelt.

4.2 Ergebnisse der Untersuchung

Dieses Kapitel verfolgt zwei Ziele. Zunächst wollen wir die Ergebnisse der Untersuchung darstellen und analysieren; dabei mußten wir wegen der großen Menge an Daten eine an den Zielen der Arbeit orientierte Auswahl der wichtigsten Ergebnisse treffen. Außerdem überprüfen wir anhand der Ergebnisse nach Möglichkeit die Hypothesen aus 3.5.3. Als Kriterium untersuchen wir dabei, welcher Aussage sich die Mehrheit der Entwickler anschließt.

Die Auswertungen beziehen sich auf den Fragebogen und übernehmen deshalb dessen Formulierungen, die teilweise von den in dieser Arbeit bevorzugten abweichen (vgl. 4.1.3).

4.2.1 Aktuelles Vorgehen

Zunächst erheben wir, welche wiederverwendbaren Artefakte (im Fragebogen Reusables genannt) auf den verschiedenen Stufen des Entwicklungsprozesses verwendet werden. Eine Übersicht über den Grad der Nutzung insgesamt, d.h. ohne Unterscheidung nach der Art der Artefakte, zeigt Tabelle 4-1.

Grad der Nutzung wiederverwendbarer Artefakte	Spezifikation	Konstruktion	Programmierung				Systemintegration
			Codierung	Entwicklertest	Techn. Doku.	Benutzer-Doku.	Integrationstest
sehr hoch	3%	0%	9%	0%	0%	0%	0%
hoch	34%	19%	22%	13%	3%	0%	9%
mittel	16%	19%	34%	9%	9%	6%	3%
niedrig	16%	19%	25%	10%	6%	13%	6%
sehr niedrig	3%	6%	3%	9%	3%	0%	6%
keine	28%	38%	7%	59%	79%	81%	76%

Tabelle 4-1: Grad der Nutzung wiederverwendbarer Artefakte insgesamt

Es zeigt sich, daß wiederverwendbare Artefakte von der Mehrheit der Entwickler (53 Prozent respektive 65 Prozent) in den Phasen der Spezifikation und Codierung mit zumindest mittlerer Intensität genutzt werden. Wir analysieren im Folgenden, welche Arten von Artefakten in diesen Phasen genutzt werden und klassifizieren sie gemäß der Einteilung in 2.1.3.

Templates	Modellhafte Beschreibungen	Geschäftsprozeßbeschreibungen	UML Use Cases	Andere
69%	16%	6%	0%	6%

Tabelle 4-2: In der Spezifikation genutzte wiederverwendbare Artefakte

In der Spezifikation werden überwiegend Templates genutzt, die unmittelbar in stark abstrahierter Form ins Endprodukt eingehen (siehe Tabelle 4-2). Demgegenüber kommen in der Codierungsphase fast immer Unterprogramme und in knapp der Hälfte der Fälle objektorientierte Klassen zum Einsatz; sie gehen unmittelbar in konkreter Form ins Endprodukt ein. Die an zweiter Stelle liegenden Demonstrationsprogramme dienen der Dokumentation und gehen mittelbar ins Endprodukt ein, ebenso wie die an fünfter Stelle liegenden Code-Generatoren (siehe Tabelle 4-3).

Unterprogramme	Demo-Progr.	OO-Klassen	Templates	Code-Generatoren	Andere
94%	59%	47%	38%	25%	25%

Tabelle 4-3: In der Codierung genutzte wiederverwendbare Artefakte

Somit bestätigt sich Hypothese [Code] lediglich für unmittelbar konkret sowie mittelbar genutzte Artefakte; für Templates, die unmittelbar in abstrahierter Form in das Enprodukt eingehen, ist sie widerlegt.

Daneben findet Wiederverwendung in nennenswertem Umfang auch in der Kostruktionsphase statt: insgesamt 38 Prozent der Befragten geben den Grad der Nutzung mit hoch oder mittel an (vgl. Tabelle 4-1). Auch hier handelt es sich bei den genutzten Artefakten vorwiegend um Templates (siehe Tabelle 4-4).

Templates	Modellhafte Beschreibungen	Andere
53%	16%	19%

Tabelle 4-4: In der Konstruktion genutzte wiederverwendbare Artefakte

Weiterhin untersuchen wir die Herkunft genutzter wiederverwendbarer Artefakte. Für die Entwickler insgesamt ergibt sich, daß die eigene Entwicklung die wichtigste Quelle ist (siehe Abbildung 4-2), gefolgt von informellen Kontakten innerhalb der eigenen Abteilung und solchen außerhalb der Abteilung; alles dies repräsentiert Ad-Hoc-Wiederverwendung. Die für die systematische Wiederverwendung eingerichtete Reuse Library steht in der Bedeutung für die Mehrheit der Entwickler an letzter Stelle.

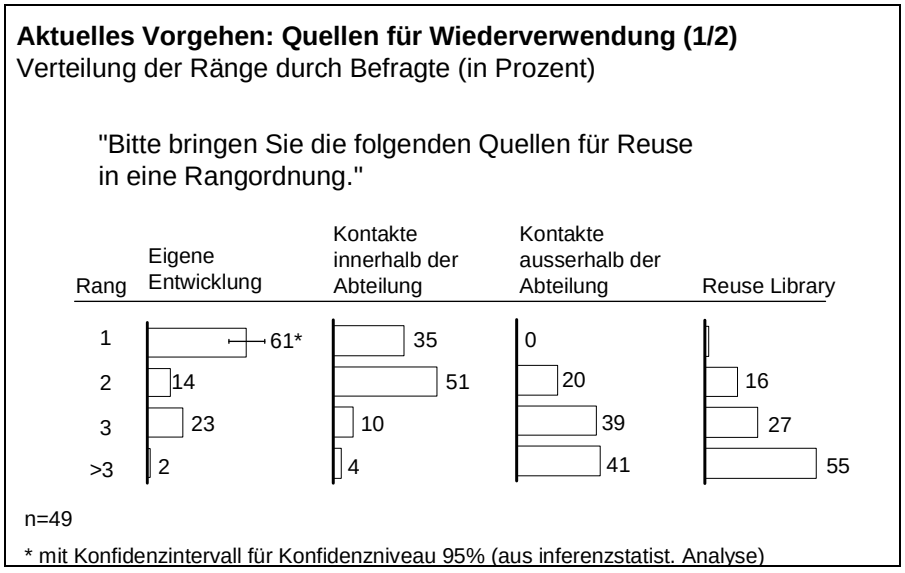

Abbildung 4-2: Herkunft wiederverwendbarer Artefakte (insgesamt)

Eine getrennte Auswertung nach Dienstalter zeigt allerdings ein differenzierteres Bild (siehe Abbildung 4-3): die Aussage trifft nur zu für Entwickler mit höherem Dienstalter. Für jüngere Entwickler hingegen ist die Reihenfolge der beiden wichtigsten Quellen umgekehrt: Kontakte innerhalb der Abteilung stehen in der Bedeutung vor der eigenen Entwicklung.[55]

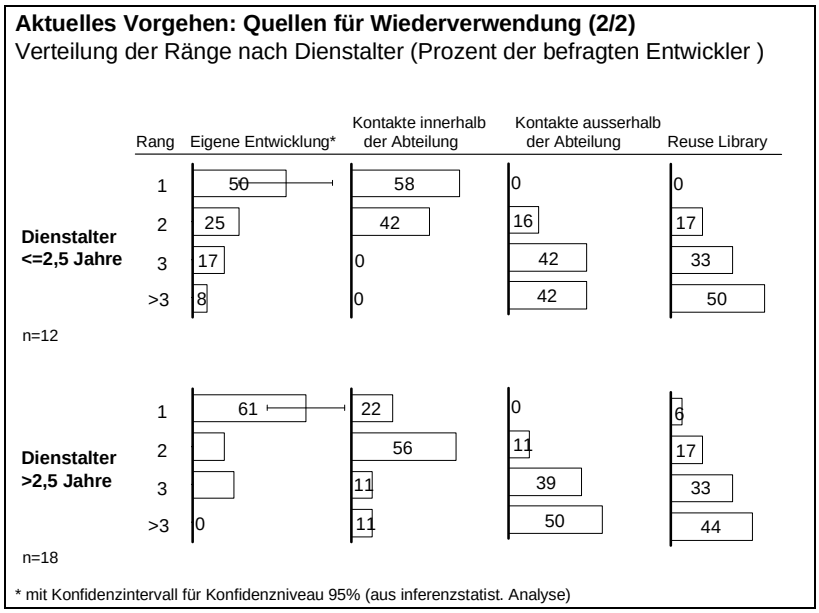

Abbildung 4-3: Herkunft wiederverwendbarer Artefakte (nach Dienstalter)

Eine mögliche Erklärung ist, daß wenig erfahrene Entwickler keinen ausreichend großen eigenen Fundus haben. Hypothese [AdHoc] bestätigt sich daher: der Schwerpunkt liegt auf Ad-Hoc-Wiederverwendung. Hypothese [EigEntw]

[55] Die Repräsentativität der Aussage ist eingeschränkt durch die kleine Stichprobe, die zu einem großen Konfidenzintervall führt.

bestätigt sich nur für Entwickler mit einem Dienstalter von zweieinhalb Jahren und mehr; für Entwickler mit einem geringeren Dienstalter ist sie widerlegt.

Um herauszufinden, weshalb nicht mehr systematische Wiederverwendung stattfindet, untersuchen wir das aktuelle Vorgehen bezüglich der Reuse Library. Abbildung 4-4 zeigt, daß fast alle Entwickler sie kennen und eine große Mehrheit von 78 Prozent auch weiß, wie sie genutzt wird; weniger als ein Viertel der Entwickler jedoch weiß, wie sie bestückt wird. Letzteres bedeutet, daß mehr als drei Viertel der Entwickler nichts direkt oder indirekt in die Reuse Library einstellen können; eine mögliche Folge ist ein Mangel an zur Verfügung stehender wiederverwendbarer Software.

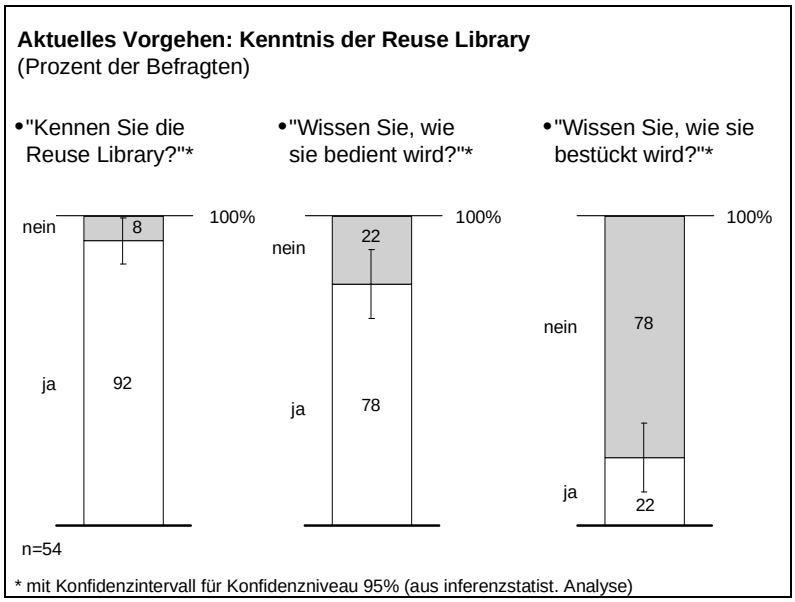

Abbildung 4-4: Kenntnis der Reuse Library

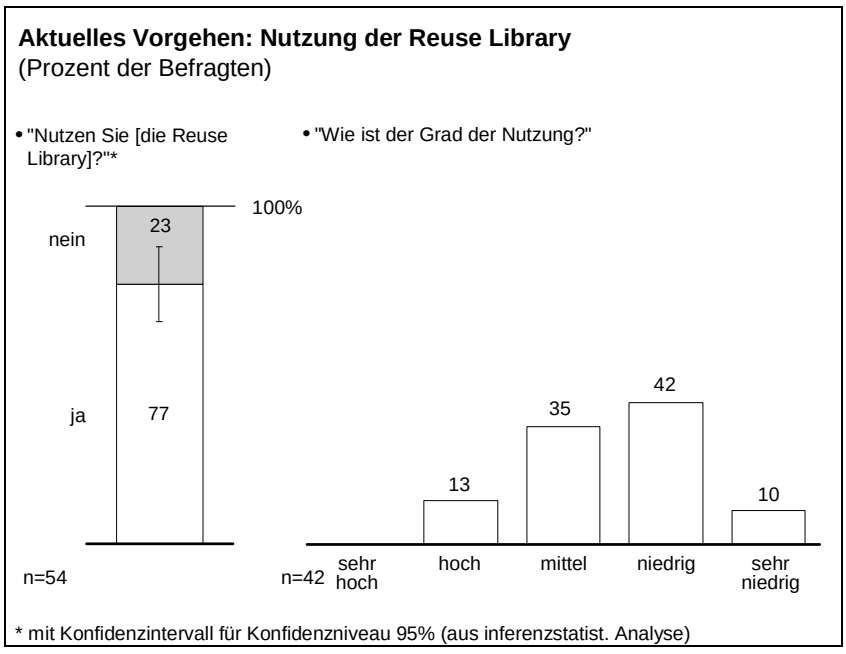

Abbildung 4-5: Nutzung der Reuse Library

Gut drei Viertel der Entwickler nutzen grundsätzlich die Reuse Library (siehe Abbildung 4-5); allerdings wird der Nutzungsgrad von der Mehrheit der Befragten als niedrig oder sehr niedrig eingestuft. Das deutet darauf hin, daß möglicherweise entsprechend Hypothese [MangelSW] nicht genügend wiederverwendbare Software zur Verfügung steht. Ein weiteres Indiz in dieser Richtung ist die Festellung von oben, daß weniger als ein Viertel der Entwickler potentiell als Hersteller in Frage kommen. Wir greifen dies in 4.2.4 nochmals abschließend auf.

4.2.2 Einschätzung der Wiederverwendung

Die grundsätzliche Haltung zur Wiederverwendung ist – wie Abbildung 4-6 zeigt – äußerst positiv: 95 Prozent der Befragten finden, daß in Zukunft mehr oder viel mehr Wiederverwendung betrieben werden sollte.

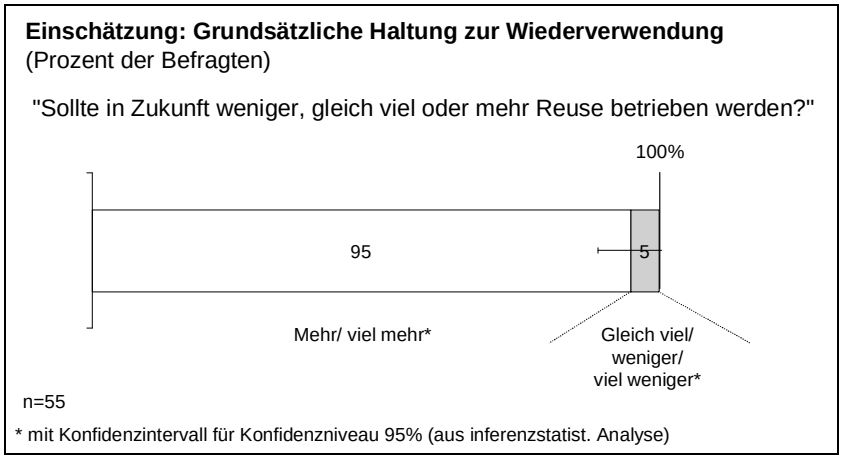

Abbildung 4-6: Grundsätzliche Haltung zur Wiederverwendung

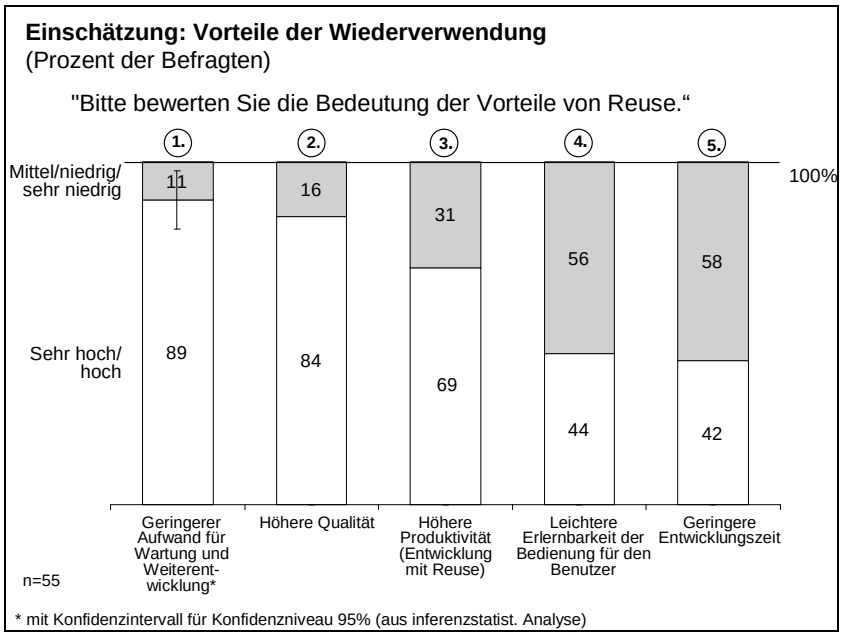

Abbildung 4-7: Vorteile der Wiederverwendung

Die Einschätzung der Vorteile der Wiederverwendung ist Abbildung 4-7 zu entnehmen. Die höchste Bedeutung hat der geringere Aufwand für Wartung und Weiterentwicklung[56]; 89 Prozent der Befragten stufen ihn als hoch oder sehr hoch ein. Dies bestätigt Hypothese [SenkWart]. An zweiter Stelle folgt höhere Qualität mit 84 Prozent; dadurch bestätigt sich Hypothese [QualiErh]. Erst danach kommt die höhere Produktivität bei der Nutzung wiederverwendbarer Software; Hypothese [ErhProd] ist also widerlegt. Weniger als die Hälfte der Befragten stuft die Bedeutung der geringeren Entwicklungszeit als hoch oder sehr hoch ein, Hypothese [VerkZeit] ist somit widerlegt.

Neben der Bedeutung der Vorteile ist auch die Frage von Interesse, wann die Vorteile wirksam werden. Eine entsprechende Analyse ergibt, daß nur die höhere Qualität schon bei der Herstellung relevant ist, denn auch die ursprünglich entwickelte Software kann davon profitieren, vor allem, wenn der Rückfluß von Qualitätsverbesserungen aus den Nutzungsprojekten zum ursprünglichen Projekt sichergestellt ist. Alle anderen Vorteile werden erst wirksam, wenn die Software genutzt (wiederverwendet) wird. Dies gilt insbesondere für höhere Produktivität und geringere Entwicklungszeit.

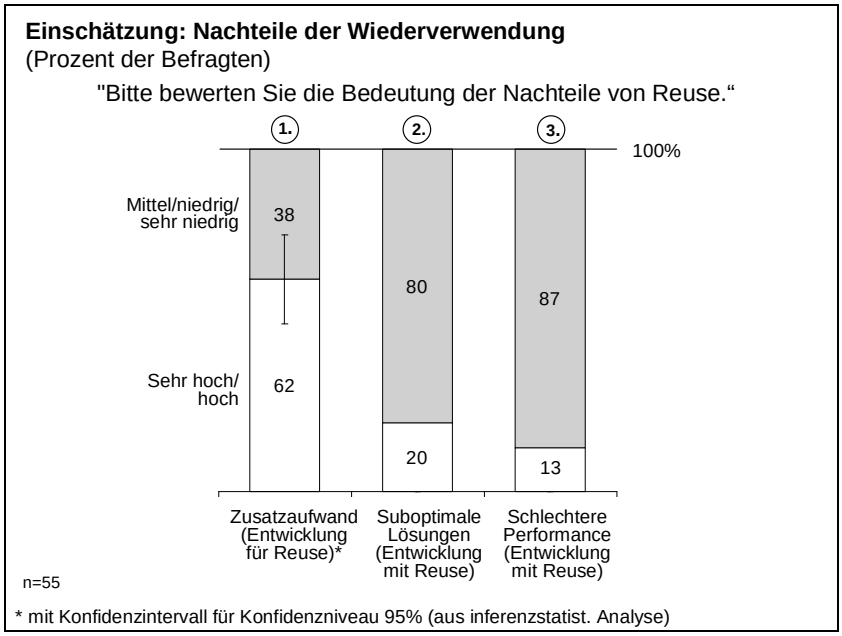

Abbildung 4-8: Nachteile der Wiederverwendung

Als wichtigster Nachteil der Wiederverwendung wird der Zusatzaufwand bei der Herstellung wiederverwendbarer Software bewertet: 62 Prozent der Befragten stufen seine Bedeutung als hoch oder sehr hoch ein (vgl. Abbildung 4-8/Nr. 1). Das bestätigt Hypothese [ZusatzAuf]. Die Nachteile, die bei der Nutzung entstehen können, haben hingegen geringe Bedeutung: 20 Prozent der Befragten stufen die Bedeutung der Gefahr suboptimaler Lösungen als hoch oder sehr hoch ein; nur 13 Prozent werten schlechtere Performance entsprechend (vgl. Abbildung 4-8/Nr. 2, 3). Dies ist ein Indiz, das für die Gültigkeit

[56] Dabei spielt der hohe Aufwand für Wartung und Weiterentwicklung (vgl. 4.1.2) eine Rolle, der allerdings eine typische Folge länger andauernden Erfolgs bei Produktunternehmen ist.

von Hypothese [GrundsProb] spricht; wir greifen diese Hypothese in 4.2.3 nochmals auf.

Bei der Abwägung der Vor- und Nachteile der Wiederverwendung sind 95 Prozent der Befragten der Meinung, die Vorteile überwögen (vgl. Abbildung 4-9). Gemeinsam mit der grundsätzlich positiven Einstellung widerlegt dies Hypothese [NegEinst].

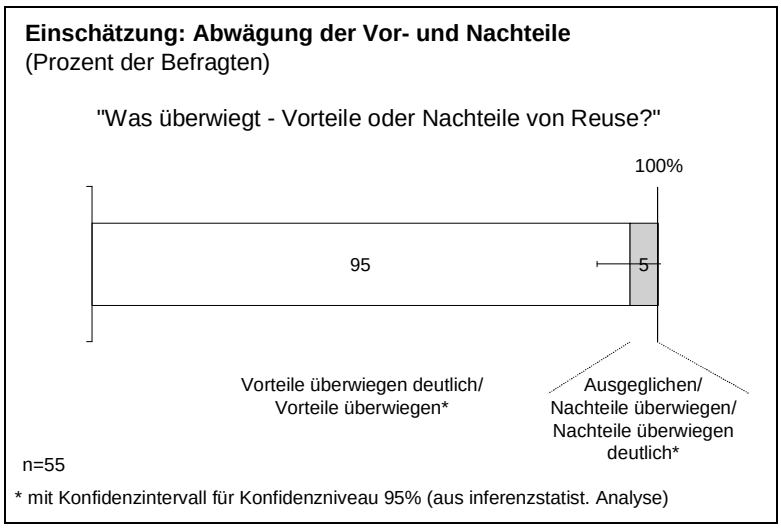

Abbildung 4-9: Abwägung der Vor- und Nachteile

4.2.3 Hindernisse

Die Befragten bewerteten auch die Hindernisse, also die Faktoren, die Wiederverwendung verhindern. Dabei ergeben sich naturgemäß Überschneidungen mit den Nachteilen, die als immanente Hindernisse wirksam werden, beispielsweise der Zusatzaufwand für die Entwicklung wiederverwendbarer Software. Die Mehrzahl der Hindernisse besteht jedoch in nicht-immanenten Nachteilen der Wiederverwendung, beispielsweise in mangelnder Information der Entwickler über Wiederverwendung. Die Ergebnisse der Untersuchung bezüglich der Hindernisse sind in Abbildung 4-10 bis Abbildung 4-12 zu sehen. Die verschiedenen Hindernisse sind dabei nach der Summe der Befragten sortiert, die die Bedeutung jeweils mit 'sehr hoch', 'hoch' oder 'mittel' einstufen.

Zunächst betrachten wir Abbildung 4-10. Zwei der bedeutendsten Hindernisse entstehen aus unzureichenden Ressourcen für die Entwicklung wiederverwendbarer Software: Zeitmangel und Mangel an personellen Ressourcen. Ihre Bedeutung wird von 67 respektive 56 Prozent der Befragten zumindest als mittel eingestuft (vgl. Abbildung 4-10/Nr. 1, 3) . Dadurch wird Hypothese [Ress] bestätigt; außerdem ist dies ein Indiz für die Gültigkeit der Hypothese [Invest], die wir in Kapitel 6 abschließend behandeln. Die Bedeutung der abschreckenden Wirkung des Dokumentations- und Wartungsaufwands für die Herstellung wiederverwendbarer Software wird von 53 Prozent der Befragten zumindest als mittel eingestuft (vgl. Abbildung 4-10/Nr. 4); das bestätigt Hypothese [Abschreck]. Hypothese [InfoTool] wird bezüglich mangelnder Information dadurch bestätigt, daß 62 Prozent der Entwickler die Bedeutung un-

zureichender Information zumindest als mittel bewerten (vgl. Abbildung 4-10/Nr. 2); wir gehen auf sie unten nochmals bezüglich der Tool-Unterstützung ein.

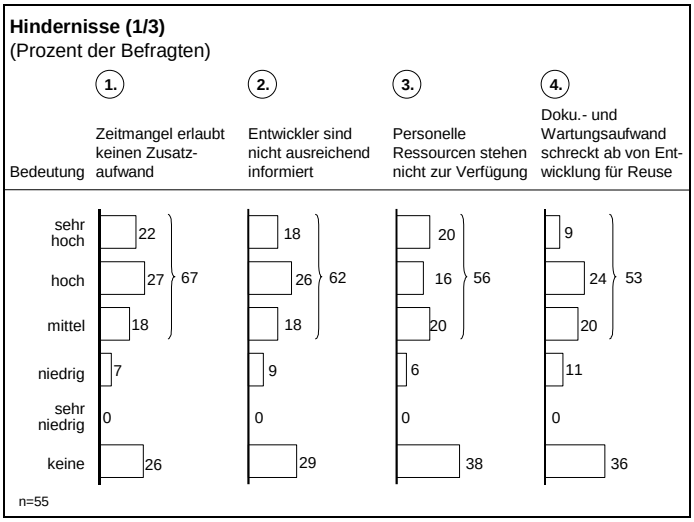

Abbildung 4-10: Hindernisse (1)

Die Bedeutung von Anreizen wird insgesamt als gering eingestuft: für 55 Prozent der Befragten spielen sie keine Rolle bei der Herstellung wiederverwendbarer Software (vgl. Abbildung 4-11/Nr. 8); bei der Nutzung sind es sogar 78 Prozent (vgl. Abbildung 4-12/Nr. 13). Dies ist ein bestätigendes Indiz für Hypothese [HindAbb] und ein Indiz für die Widerlegung von Hypothese [FinAnr], auf die wir unten noch einmal zurückkommen. Bemerkenswert ist darüber hinaus der deutliche Unterschied zwischen der Beurteilung der Herstellung und derjenigen der Nutzung: sie ist vermutlich auf die nicht ausreichenden Ressourcen für die Herstellung (s.o.) zurückzuführen, die als gegenläufiger Anreiz empfunden werden.

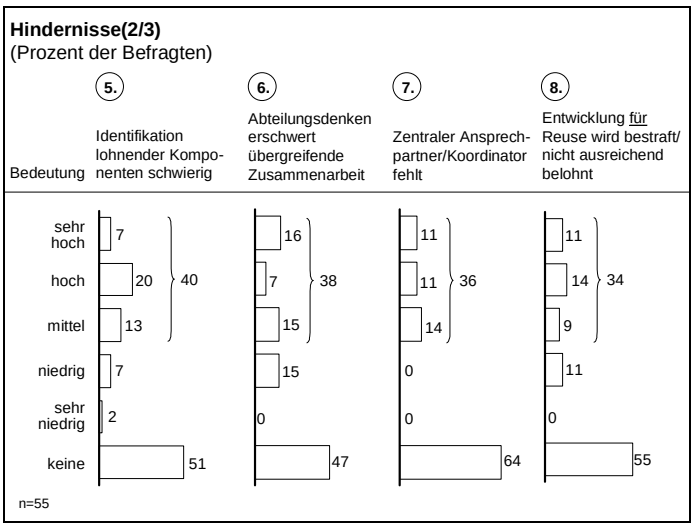

Abbildung 4-11: Hindernisse (2)

Aus der Betrachtung der in Abbildung 4-12 dargestellten Ergebnisse ergibt sich Folgendes: 62 Prozent der Befragten sind der Meinung, daß das Ausmaß der objektorientierten Entwicklung keine Rolle für den Erfolg der Wiederverwen-

dung spielt (vgl. Abbildung 4-12/Nr. 9); dies bestätigt Hypothese [OO]. Was die Abnahme der Kontrolle über das eigene Produkt bei der Nutzung wiederverwendbarer Software angeht, so hat sie für 78 Prozent der Befragten keine Bedeutung (vgl. Abbildung 4-12/Nr. 12). Zusätzlich sind 84 Prozent der Meinung, Wiederverwendung führe nicht zu schlechteren Produkten (vgl. Abbildung 4-12/Nr. 14). Diese beiden Ergebnisse sprechen eindeutig dagegen, daß der Not-invented-here-Effekt besteht: Hypothese [NIH] ist also widerlegt. Zusätzlich folgt aus letzterem Ergebnis im Zusammenspiel mit den Indizien aus 4.2.2 (Abbildung 4-8/Nr. 2, 3), daß Hypothese [GrundsProb] bestätigt ist.

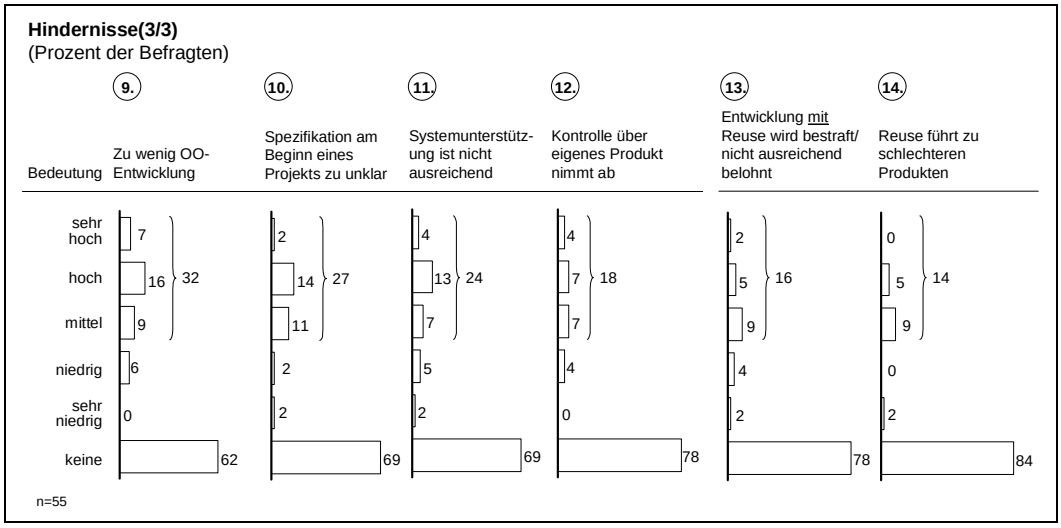

Abbildung 4-12: Hindernisse (3)

Eine Mehrheit von 69 Prozent der Befragten stuft das Hindernis nicht ausreichender Systemunterstützung als unbedeutend ein (vgl. Abbildung 4-12/Nr. 11), was Hypothese [InfoTool] bezüglich der Tool-Unterstützung widerlegt. Dies ist aufgrund des Vorhandenseins der Wiederverwendungs-Bibliothek nicht verwunderlich, zeigt aber dennoch, daß es grundsätzlich möglich ist, eine befriedigende Bibliothek zu erstellen.

Wir haben die Hindernisse nach der Relevanz für die in dieser Arbeit im Fokus befindliche Herstellung und die Nutzung wiederverwendbarer Software klassifiziert; eine Übersicht gibt Tabelle 4-5.

Wir analysieren das Ergebnis der Klassifikation daraufhin, welche Maßnahmen zur Steigerung der Wiederverwendung notwendig sind. Die Bedeutung der ersten vier Hindernisse wird von der Mehrheit der Befragten jeweils mindestens mit mittel angegeben (siehe Abbildung 4-10). Drei dieser vier Hindernisse betreffen vor allem die Herstellung wiederverwendbarer Software. Zudem gehören die Hindernisse, die ausschließlich für die Nutzung relevant sind, zu den am wenigsten bedeutenden. Daraus folgt, daß Maßnahmen zur Steigerung der Wiederverwendung zuvorderst bei der Herstellung ansetzen müssen, was die Wahl des Fokus dieser Arbeit unterstreicht.

Rang	Hindernis (Formulierung im Fragebogen)	Relevanz	
		Herstellung	**Nutzung**
1	Mangel an Zeit erlaubt keinen Zusatzaufwand (insbesondere für Entwicklg. für Reuse)	Schwerpunkt	
2	Entwickler sind nicht ausreichend informiert	Relevant	Relevant
3	Personelle Ressourcen für Zusatzaufwand (insbesondere für Entwicklg. für Reuse) stehen nicht zur Verfügung	Schwerpunkt	
4	Höherer Dokumentierungs-, Wartungs- und Weiterentwicklungsaufwand schreckt von Entwicklung für Reuse ab	Ausschließlich	
5	Identifikation lohnender Komponenten ist schwierig	Relevant	Relevant
6	Abteilungsdenken erschwert übergreifende Zusammenarbeit	Relevant	Relevant
7	Es fehlt ein zentraler Ansprechpartner/Koordinator	Relevant	Relevant
8	Entwicklung für Reuse wird bestraft/nicht ausreichend belohnt	Ausschließlich	
9	Es wird zu wenig objektorientiert entwickelt	Relevant	Relevant
10	Spezifikation ist am Anfang eines Entwicklungsprojekts noch zu unklar	Relevant	Relevant
11	Systemunterstützung ist nicht ausreichend	Relevant	Relevant
12	Bei Entwicklung mit Reuse nimmt die Kontrolle über das eigene Produkt ab		Ausschließlich
13	Entwicklung mit Reuse wird bestraft/nicht ausreichend belohnt		Ausschließlich
14	Reuse führt zu schlechteren Produkten (schlechtere Performance und/oder suboptimale, nicht maßgeschneiderte Lösungen)	Relevant	Schwerpunkt

Tabelle 4-5: Relevanz der Hindernisse für Herstellung und Nutzung

4.2.4 Notwendige Maßnahmen

Wir betrachten nun die aus Sicht der Befragten notwendigen Maßnahmen. Zunächst zeigt sich, daß es Potential für einen Ausbau systematischer Wiederverwendung gibt: zwei Drittel der Befragten geben an, daß sie die Reuse Library mehr nutzen würden, wenn etwas daran verbessert würde (vgl. Abbildung 4-13). Die Mehrheit der genannten Maßnahmen zielt auf eine Erhöhung der Menge an verfügbarer Software ab: 36 Prozent der Befragten fordern mehr Dokumentation, 42 Prozent lokale Bausteine. Darüber hinaus bezieht sich ein Teil der individuellen Antworten ('Andere') darauf: 20 Prozent zielen auf mehr Applikationsbausteine ab, 14 Prozent auf mehr Beispiele. Daraus folgt, daß offensichtlich nicht genügend Software vorhanden ist, was in Verbindung mit den Indizien aus 4.2.1 (Abbildung 4-5) Hypothese [MangelSW] bestätigt.

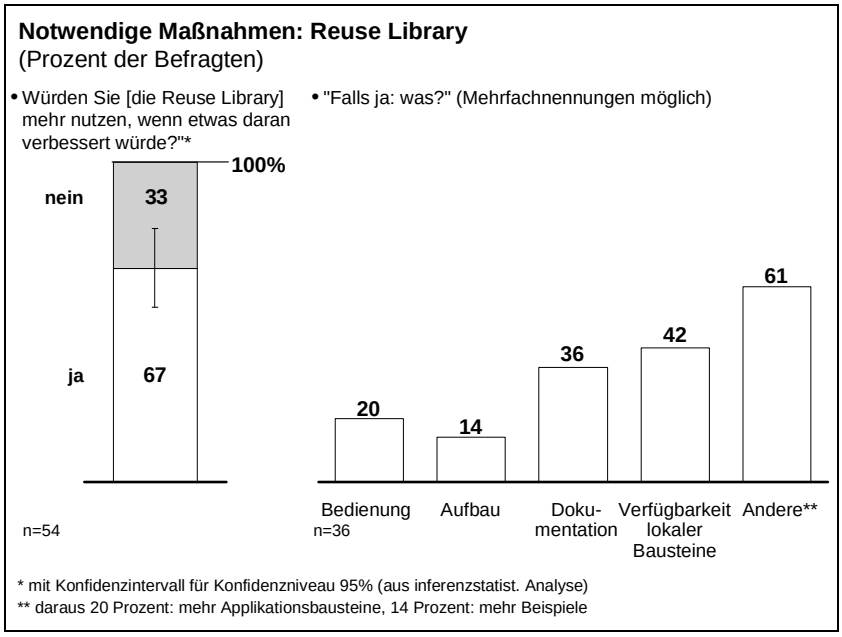

Abbildung 4-13: Notwendige Maßnahmen (Reuse Library)

Die Einschätzung der notwendigen Maßnahmen hinsichtlich der Anreize zeigt Abbildung 4-14, getrennt nach Anreizen für Entwickler und Anreizen für Gruppenleiter; trotz leicht abweichender Zahlen ergeben sich in der Aussage keine Unterschiede. In beiden Fällen mißt die Mehrheit der Befragten dem Abbau der Hindernisse im Gegensatz zu Anreizen hohe bis überwiegend sehr hohe Bedeutung bei. Gleichzeitig werden alle Anreizformen (u.a. sichtbares Honorieren, interessante Aufgaben, direkte finanzielle Anreize) von der Mehrheit der Befragten als unbedeutend eingestuft. Das bestätigt Hypothese [HindAbb]. Speziell der Fall direkter finanzieller Anreize ist bemerkenswert: ihnen werden von 92 respektive 88 Prozent keine Bedeutung zugemessen. Das widerlegt Hypothese [FinAnr].

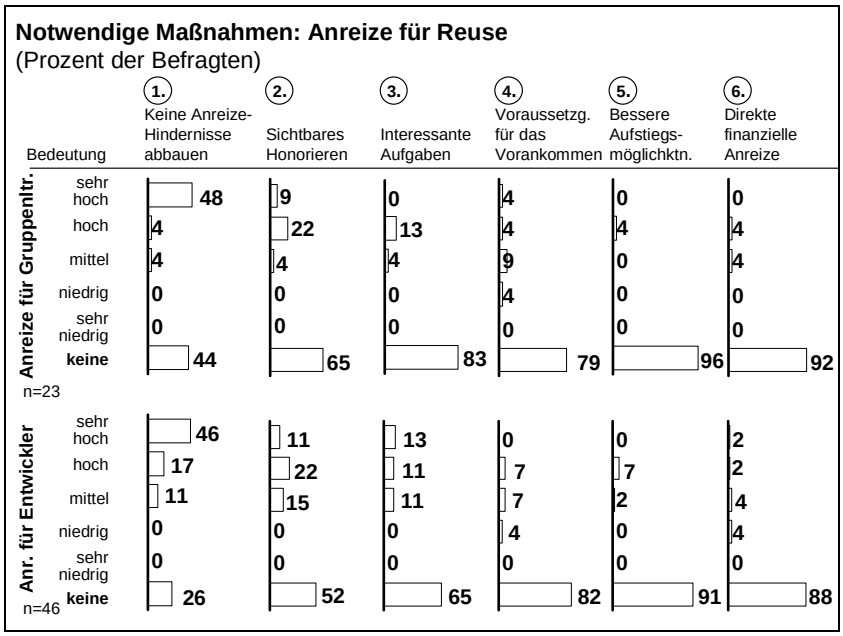

Abbildung 4-14: Notwendige Maßnahmen (Anreize für Reuse)

Eine Auswertung der freien Antworten auf die offene Frage nach notwendigen Maßnahmen zur Förderung der Wiederverwendung zeigt Abbildung 4-15. Die Antworten wurden nach Kategorien klassifiziert; die Liste beschränkt sich auf die sechs meistgenannten Kategorien. Da es sich hier um Antworten auf offene Fragen handelt, kann nicht derselbe Maßstab angelegt werden wie an die bisherigen Fragen, die in geschlossener Form gestellt wurden. Vielmehr sind die Antworten höher zu gewichten (vgl. 4.1.3). Daher setzen wir die höchste Anzahl von Antworten (30 Prozent der Befragten) gleich 100 Prozent und beziehen die Anteile der anderen Antworten darauf, indem wir sie proportional berechnen.

Dreißig Prozent der Befragten regten direkt oder indirekt an, zusätzliche Ressourcen für wiederverwendbare Software bereitzustellen; das ist die am häufigsten genannte Maßnahme. Das Ziel ist dabei, eine größere Menge bereitzustellen oder die Wiederverwendbarkeit zu erhöhen. Dreizehn Prozent der Befragten regten explizit bessere Dokumentation und mehr Beispiele an; das entspricht letzterem Ziel.

Am zweithäufigsten wurden (von 27 Prozent der Befragten, entsprechend 90 Prozent der maximal erzielten Antworten) Maßnahmen genannt, die der Information und der Schaffung von Bewußtsein dienen; wir haben sie unter dem Punkt 'Internes Marketing' klassifiziert. Dies korrespondiert damit, daß mangelnde Information der Entwickler das zweitwichtigste Hindernis ist (vgl. Abbildung 4-10/Nr. 2). In unserer Definition der Wiederverwendbarkeit (siehe 2.1.2) entspricht das der Verfügbarkeit, die einen Bestandteil der Nutzbarkeit für Entwickler darstellt: es genügt nicht, daß wiederverwendbare Software vorhanden ist – sie muß auch für die Entwickler verfügbar sein.

An dritter Stelle steht die Modifikation des Entwicklungsprozesses. Sie wird von 20 Prozent der Befragten (67 Prozent der maximal erzielten Antworten) als notwendig erachtet. Dies bestätigt Hypothese [Prozess].

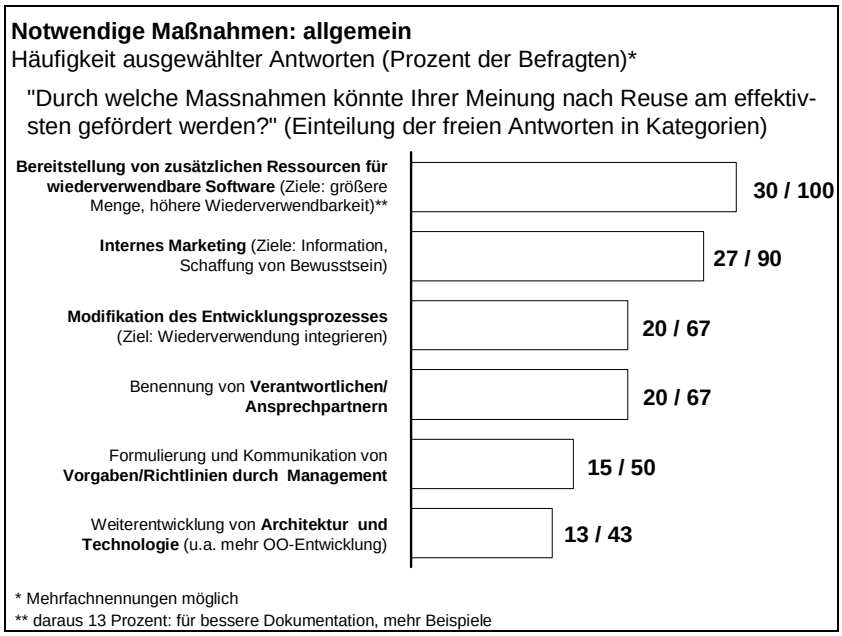

Abbildung 4-15: Notwendige Maßnahmen (freie Antworten)

Die Einschätzung hinsichtlich notwendiger organisatorischer Maßnahmen zeigen Abbildung 4-16 und Tabelle 4-6. Generell zeigt sich ein sehr ausgeglichenes Bild hinsichtlich der Frage, ob die Aufgaben im Zusammenhang mit Wiederverwendung von einer zentralen, speziell dafür eingerichteten Einheit oder dezentral als zusätzliche Aufgabe wahrgenommen werden sollten (vgl. Abbildung 4-16).[57]

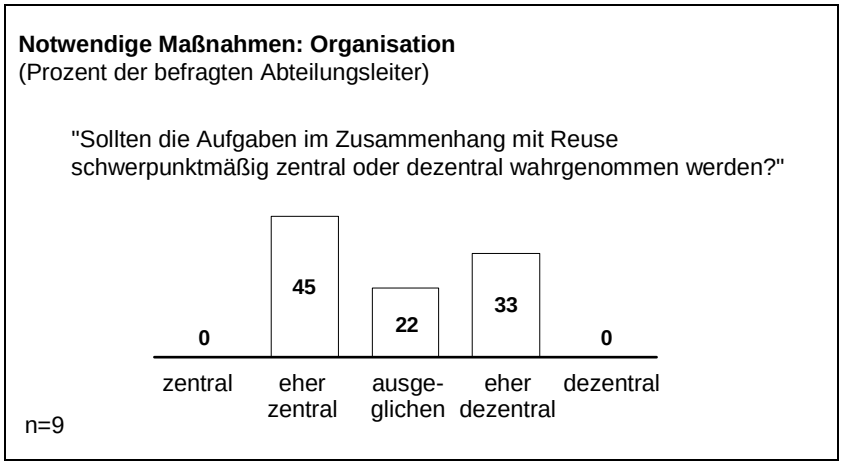

Abbildung 4-16: Organisatorische Maßnahmen

Bezüglich der einzelnen Aufgaben ergibt sich ein klareres Bild: lediglich das Sichten und Sammeln wiederverwendbarer Bausteine, die Verwaltung der Infrastruktur und die Verwaltung der für Wiederverwendung spezifischen Prozesse sollten nach Meinung der Mehrheit der Befragten von einer zentralen Einheit vorgenommen werden. Bei allen Entwicklungstätigkeiten (Entwicklung, Generalisierung, Dokumentation, Erstellung von Testfällen) zieht die

[57] Da es sich um eine Management-Frage handelt, wurden nur die Abteilungsleiter befragt.

Mehrheit eine dezentrale Lösung vor. Insofern bestätigt sich Hypothese [ZentrOrg] nur für die drei oben genannten Aufgaben. Dies entspricht im Wesentlichen der tatsächlichen Lösung.

	Entwicklung	Dokumentation	Erstellung von Testfällen	Generalisierung	Sammeln, Screening	Verwaltung der Infrastruktur	Verwaltung der Reuse-Prozesse
zentral	**0%**	**33%**	**33%**	**11%**	**78%**	**100%**	**100%**
dezentral	**100%**	**67%**	**67%**	**89%**	**22%**	**0%**	**0%**

Tabelle 4-6: Organisation einzelner Aufgaben

4.2.5 Ökonomische Aspekte

Die Befragten gaben Einschätzungen zu den ökonomischen Aspekten ab. Es wurden jeweils drei Schätzwerte erhoben: durchschnittlicher Wert, minimaler und maximaler Wert. Für diese drei Werte berechneten wir jeweils den Mittelwert und die Standardabweichung der Schätzungen aller Befragten.

Auf die Frage nach dem Zusatzaufwand für die Entwicklung wiederverwendbarer Software wird im Mittelwert ein Durchschnittsfaktor von 2,2 angegeben; das Minimum liegt bei 1,4, das Maximum bei 6,6 (siehe Tabelle 4-7). Bemerkenswert ist die hohe Schwankungsbreite beim Maximum, das eine Standardabweichung von 15,2 aufweist. Diese Schätzungen deuten darauf hin, daß große Unterschiede in den zu erwartenden Kosten bestehen. Das ist ein erstes Indiz für die Gültigkeit der Hypothese [Varianz], die wir in 6.3.5 nochmals aufgreifen. Der ermittelte Durchschnittswert entspricht den in der Literatur angegebenen Werten, die zwischen 1,5 und 3 liegen; allerdings ist dort von statistischen Schwankungen keine Rede (vgl. [Tra95], xi; [JGJ97], 23).

	Durchschnitt	Minimum	Maximum
Mittelwert	2,2	1,4	6,6
Standardabweichung	1,5	0,6	15,2

Tabelle 4-7: Geschätzte Kosten der Wiederverwendbarmachung (bezogen auf ursprüngliche Entwicklungskosten)

Um auch die Kosten der Nutzung in die Betrachtung zu integrieren, wurde außerdem die Anzahl der für den Break-Even, d.h. das Erreichen der Gewinnschwelle, notwendigen Nutzungen erhoben (siehe Tabelle 4-8). Der Durchschnitt liegt mit 3,9 bereits über dem in der Literatur zitierten Wert von zwei bis drei Nutzungen (vgl. [JGJ97], 23). Zudem ist hier die statistische Standardabweichung noch weit größer. Diese Werte sind ein weiteres Indiz für die Gültigkeit der Hypothese [Varianz].

	Durchschnitt	Minimum	Maximum
Mittelwert	3,9	2,0	41,2
Standardabweichung	2,6	1,4	159,7

Tabelle 4-8: Geschätzte Anzahl der für den Break-Even notwendigen Nutzungen

4.3 Analyse der Ergebnisse und Schlußfolgerungen

4.3.1 Überblick der geprüften Hypothesen

Einen Überblick über das Ergebnise der Hypothesenprüfung gibt Tabelle 4-9. Insgesamt ergibt sich ein differenziertes Bild: zwölf Hypothesen bestätigen sich, fünf sind widerlegt, vier Hypothesen sind lediglich teilweise bestätigt. Für drei Hypothesen steht das Ergebnis noch aus ([Archi], [GroßesPot], [Invest]). Sie verlangen detailliertere Untersuchungen, die in Kapitel 5 ([Archi]) und Kapitel 6 ([GroßesPot], [Invest]) durchgeführt werden.

Nr	Hypothese	Ergebnis	Kapitel
1	[Code] Wiederverwendung findet heute überwiegend in der Programmierung statt.	• Bestätigt für unmittelbar konkret und mittelbar ins Endprodukt eingehende Artefakte. • Widerlegt für abstrahiert eingehende Artefakte (Spezifikation/Templates).	4.2.1
2	[AdHoc] Der Schwerpunkt liegt auf Ad-Hoc-Wiederverwendung (Entwickler selbst oder innerhalb der Gruppe).	Bestätigt.	4.2.1
3	[EigEntw] Eigene Entwicklung ist für die Entwickler die wichtigste Quelle für wiederverwendbare Software.	• Bestätigt für Entwickler mit einem Dienstalter von zweieinhalb Jahren und mehr. • Widerlegt für Entwickler mit einem geringeren Dienstalter.	4.2.1
4	[MangelSW] Es besteht ein Mangel an systematisch wiederverwendbarer Software.	Bestätigt.	4.2.4, 4.2.1

5	[NegEinst] Die Entwickler sind grundsätzlich negativ zur Wiederverwendung eingestellt.	Widerlegt.	4.2.2
6	[ErhProd] Die Erhöhung der Entwicklungsproduktivität ist der wichtigste Vorteil von Wiederverwendung.	Widerlegt.	4.2.2
7	[VerkZeit] Verkürzung der Entwicklungszeit ist ein wichtiger Vorteil der Wiederverwendung.	Widerlegt.	4.2.2
8	[SenkWart] Senkung der Wartungslast ist ein wichtiger Vorteil der Wiederverwendung.	Bestätigt.	4.2.2
9	[QualiErh] Erhöhung der Qualität ist ein wichtiger Vorteil der Wiederverwendung.	Bestätigt.	4.2.2
10	[GrundsProb] Wiederverwendung ist grundsätzlich möglich; es gibt keine inhärenten, grundsätzlich nicht lösbaren Probleme (z.B. schlechtere Performance, suboptimale Lösungen).	Bestätigt.	4.2.2, 4.2.3
11	[ZusatzAuf] Wichtigster Nachteil ist der Zusatzaufwand für die Herstellung wiederverwendbarer Software.	Bestätigt.	4.2.2
12	[Ress] Es werden keine ausreichenden Ressourcen für die Herstellung wiederverwendbarer Software zur Verfügung gestellt.	Bestätigt.	4.2.3
13	[NIH] Das Not-invented-here-Syndrom spielt eine große Rolle.	Widerlegt.	4.2.3

14	[Abschreck] Der Zusatzaufwand für Dokumentation und Wartung schreckt von der Entwicklung wiederverwendbarer Software ab (gegenläufiger Anreiz).	Bestätigt.	4.2.3
15	[InfoTool] Mangelnde Information und mangelnde Tool-Unterstützung sind wichtige Hindernisse.	• Bestätigt für Information. • Widerlegt für Tools.	4.2.3
16	[Prozess] Wiederverwendung ist momentan im Prozess nicht ausreichend verankert.	Bestätigt.	4.2.4
17	[Archi] Die Architektur hat großen Einfluß auf den Erfolg der Wiederverwendung.	(Noch kein Ergebnis)	
18	[OO] Objektorientierte Entwicklung ist keine Voraussetzung für den Erfolg.	Bestätigt.	4.2.3
19	[HindAbb] Abbau von Hindernissen ist wichtiger als die Schaffung positiver Anreize.	Bestätigt.	4.2.4
20	[FinAnr] Finanzielle Anreize haben hohe Bedeutung.	Widerlegt.	4.2.4
21	[ZentrOrg] Die Unterstützung der Wiederverwendung durch eine zentrale Organisationseinheit ist notwendig.	• Bestätigt für Sammeln & Screening, Verwaltung der Library, Verwaltung der Reuse-Prozesse. • Widerlegt für Tätigkeiten der Entwicklung (Entwicklung, Generalisierung, Dokumentation, Erstellung von Testfällen).	4.2.4
22	[GroßesPot] Wiederverwendung hat generell großes ökonomisches Potential aufgrund von Produktivitätssteigerungen.	(Noch kein Ergebnis)	

23	[Invest] Die Kosten der Herstellung wiederverwendbarer Software sollten als Investition betrachtet und bewertet werden.	Indizien für Bestätigung	4.2.3
24	[Varianz] Es gibt eine große Varianz in der erzielbaren Rendite.	Indizien für Bestätigung	4.2.5

Tabelle 4-9: Ergebnis der Hypothesenprüfung

4.3.2 Interpretation der Ergebnisse im Zusammenhang

Der Status quo stellt sich wie folgt dar (vgl. 4.2.1): Wiederverwendung findet bereits in nennenswertem Umfang in den Phasen der Codierung und der Spezifikation statt. In konkreter Form werden wiederverwendbare Artefakte nur bei der Codierung genutzt. Der Schwerpunkt liegt eindeutig auf Ad-Hoc-Wiederverwendung, systematische Wiederverwendung hingegen hat nachrangige Bedeutung.[58] Potential für einen Ausbau ist somit grundsätzlich im Bereich systematischer Wiederverwendung zu erwarten, zumal die Reichweite ad hoc wiederverwendbarer Software gering ist: Sie beschränkt sich auf das unmittelbare Umfeld eines Entwicklers.

Die Rahmenbedingungen sind günstig (vgl. 4.2.2, 4.2.3): Die Einstellung der Entwickler zur Wiederverwendung ist äußerst positiv, insbesondere spielt der häufig angeführte Not-invented-here-Effekt keine wesentliche Rolle. Außerdem sehen die Entwickler großes Potential für einen Ausbau der Wiederverwendung. Zudem bestehen nach Einschätzung der Entwickler keine grundsätzlichen, immanenten Probleme, die die Nutzung wiederverwendbarer Artefakte verhindern, sei es in Form schlechterer Performance oder mangelnder Qualität der zu erzielenden Lösungen. Aus dem bestehenden Potential leiten wir ab, daß es offensichtlich andere Faktoren gibt, die den Ausbau verhindern.

Zwei wesentliche Erkenntnisse lassen sich aus der Einschätzung der Vorteile gewinnen (vgl. 4.2.2, Abbildung 4-7): Zum einen ist offensichtlich die gängige Definition von Produktivität kurzfristig angelegt, weil sie die Wartung nicht einschließt. Gerade die Reduzierung des Aufwands für Wartung und Weiterentwicklung wird jedoch als größter Vorteil der Wiederverwendung eingestuft. In diesem Zusammenhang ist die Tatsache zu berücksichtigen, daß dieser Aufwand im Durchschnitt mit 39 Prozent einen hohen Anteil der Arbeitszeit einnimmt. Dabei handelt es sich jedoch um ein typisches Merkmal erfolgreicher Produktunternehmen, die sich in ihrem Markt längerfristig etabliert haben

[58] Dabei weisen wir nochmals explizit auf die im untersuchten Unternehmen geläufige Definition der Wiederverwendung hin, die den präsentierten Ergebnissen zugrundeliegt. Diese Definition umfaßt nicht die durch die äußerst konsequente Einhaltung der Schichtenarchitektur implizierte Wiederverwendung, die ein erhebliches Ausmaß hat (vgl. 4.1.2; siehe auch 2.5.3). Wenn also im folgenden von Potential für den Ausbau systematischer Wiederverwendung die Rede ist, ist damit Wiederverwendung gemeint, die darüber hinausgeht.

(vgl. 4.2.2), so daß wir das untersuchte Unternehmen in dieser Hinsicht durchaus für repräsentativ halten. Zum anderen liegt in der Einschätzung der Entwickler die Erhöhung der Qualität an zweiter Stelle der Vorteile. Dabei ist zu beachten, daß der aus ihr entstehende ökonomische Nutzen nur mittelbar finanziell wirksam wird (vgl. 2.2.3). Daraus folgt, daß der Nutzen umfassend betrachtet werden muß und sich nicht auf die unmittelbar finanziell wirksamen Anteile beschränken darf. Dies muß bei der Bewertung von Projekten und bei der strategischen Ausrichtung berücksichtigt werden.

Die Untersuchung der Hindernisse und notwendigen Maßnahmen zeigt, daß der wesentliche Hemmschuh die mangelnde Bereitschaft zur Investition in die Herstellung systematisch wiederverwendbarer Software ist (vgl. 4.2.3, Abbildung 4-10): Zeit und Ressourcen für den Zusatzaufwand stehen nicht in ausreichendem Maß zur Verfügung. Entwickler, die aus Eigeninitiative Mehraufwand für die Entwicklung wiederverwendbarer Software erbringen, werden also in der Regel indirekt dafür bestraft, weil sie das zur Verfügung stehende Budget überschreiten. Das erklärt, warum der Schwerpunkt bisher so eindeutig auf Ad-Hoc-Wiederverwendung liegt: sie verlangt im Gegensatz zu systematischer Wiederverwendung keine Investition (vgl. 2.1.2). Die mangelnde Bereitschaft zur Investition ist nicht auf die spezifische Situation des untersuchten Unternehmens, sondern hauptsächlich auf die allgemeinen Rahmenbedingungen für Softwareunternehmen zurückzuführen (vgl. 2.2.2): Gewinnrückgänge werden vom Kapitalmarkt hart bestraft, selbst wenn sie kurzfristiger Natur sind. Dies fördert Kurzsichtigkeit bei der Investitionsplanung: Investitionen unter Unsicherheit, die sich nicht direkt positiv auf die Marktposition auswirken, erscheinen unattraktiv.

Der Aufwand, der für die systematische Wiederverwendbarmachung – insbesondere für Dokumentation und Wartung – anfällt, hat neben den kurzfristigen finanziellen Auswirkungen einen weiteren negativen Effekt: er schreckt Entwickler von der Herstellung wiederverwendbarer Software ab (vgl. 4.2.3, Abbildung 4-10). Das ist ein Warnsignal auch insofern, als wegen des hohen Schwierigkeitsgrades oft die besten Entwickler dafür benötigt werden (vgl. [MSG96]); diese jedoch für Dokumentation und Wartung einzusetzen – also für Tätigkeiten, die ein anderes Qualifikationsprofil erfordern als die Neuentwicklung – ist ineffizient.

Insgesamt gesehen wird die Entwicklung wiederverwendbarer Software momentan aus Sicht der Entwickler effektiv bestraft. Was die notwendigen Maßnahmen angeht, räumen die Befragten folgerichtig mit großer Mehrheit dem Abbau von Hindernissen Vorrang vor der Einführung von Anreizen – gleich, welcher Art – ein. Eine wesentliche notwendige Maßnahme ist daher, mehr Ressourcen für die Herstellung wiederverwendbarer Software zur Verfügung zu stellen. Diese Analyse wird durch drei weitere Ergebnisse gestützt: (1) die am häufigsten genannte Maßnahme zur Förderung der Wiederverwendung ist die Bereitstellung von zusätzlichen Ressourcen für wiederverwendbare Software (vgl. 4.2.4, Abbildung 4-15); (2) sie ist die direkte Konsequenz aus dem Mangel an systematisch wiederverwendbarer Software (vgl. 4.2.1); (3) als wesentliche Verbesserung an der Reuse Library wird gefordert, mehr Software zur Verfügung zu stellen (vgl. 4.2.4, Abbildung 4-13).

Daneben gibt es deutliche Hinweise darauf, daß eine Erhöhung der Nutzbarkeit für Entwickler notwendig ist: Zum einen wird die mangelnde Information der Entwickler als zweitwichtigstes Hindernis eingestuft (vgl. 4.2.3, Abbildung 4-10), zum anderen wird explizit die Bereitstellung von Ressourcen für bessere Dokumentation und mehr Beispiele gefordert (vgl. 4.2.4, Abbildung 4-15). Mangelhafte Dokumentation ist ein Phänomen, unter dem Softwareentwicklung allgemein leidet (vgl. z.B. [Bro95], 165f); insofern scheint es uns plausibel, daß diese Aussagen über das untersuchte Unternehmen hinaus Gültigkeit haben.

Was die Organisation angeht, zieht eine deutliche Mehrheit der Befragten eine dezentrale Zuordnung der Entwicklung vor; Entwicklung umfaßt hier Generalisierung, Dokumentation und Erstellung von Testfällen. Dies schließt eine Unterstützung durch eine zentrale Gruppe, z.B. bei der Wartung, nicht aus, um die oben diskutierte abschreckende Wirkung der Entwicklung wiederverwendbarer Software abzumildern; die Verantwortung soll jedoch dezentral wahrgenommen werden. Eine zentrale Gruppe soll hingegen die Verantwortung dafür haben, wiederverwendbare Bausteine zu sammeln und für die Biliothek auszuwählen, die Infrastruktur zu pflegen und die spezifischen Prozesse zu verwalten.

4.3.3 Allgemeine Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software

Aus den Ergebnissen leiten wir verallgemeinernd die folgenden allgemeinen Erfolgsfaktoren ab:

1. Investition in wiederverwendbare Software tätigen (insbesondere, um ursprüngliche Entwickler bei der Wiederverwendbarmachung zu unterstützen, wenn notwendig)
2. Alle Merkmale der Wiederverwendbarkeit berücksichtigen
3. Nutzen umfassend bewerten

Ad 1: Das hauptsächliche Hindernis für Wiederverwendung sind unzureichende Mittel für die Herstellung wiederverwendbarer Software. Die wichtigste Voraussetzung für den Erfolg systematischer Wiederverwendung ist daher, daß dafür eine bewußte Investition getätigt wird. Nur so läßt sich auch der Mangel an systematisch wiederverwendbarer Software beseitigen. Die investierten Mittel sollten gegebenenfalls auch dafür verwendet werden, den oder die ursprünglichen Entwickler bei der Wiederverwendbarmachung zu unterstützen. Dadurch kann verhindert werden, daß gute Entwickler von der Entwicklung wiederverwendbarer Software abgeschreckt werden bzw. ineffizient für Wartungs- und Weiterentwicklungsaufgaben eingesetzt werden.

Ad 2: Die Merkmale der Wiederverwendbarkeit umfassen neben Adaptierbarkeit und Portierbarkeit auch die Nutzbarkeit für Entwickler, die sich in Verständlichkeit, Änderungseffizienz und Verfügbarkeit untergliedert (vgl. 2.1.2). Alle diese Merkmale sind bei der Entwicklung zu berücksichtigen; insbesondere sind ausreichende Dokumentation und Beispiele im Sinne der Verständlichkeit und Information der Entwickler im Sinne der Verfügbarkeit sicherzu-

stellen. Dies ist eine notwendige Bedingung für den Erfolg der Wiederverwendung, denn eine Komponente, die zwar aufgrund ihrer Variabilität grundsätzlich an die Anforderungen eines speziellen Kontexts der Wiederverwendung angepaßt werden kann, aber nicht auffindbar ist, einen zu hohen Einarbeitungsaufwand erfordert oder für den Entwickler nicht verständlich ist, wird nicht genutzt.

Ad 3: Zunächst ist bei der Bewertung des unmittelbar finanziell wirksamen Nutzens eine vollständige Definition der Produktivität zugrundezulegen, die die Wartung berücksichtigt. Außerdem ist auch der nur mittelbar finanziell wirksame Nutzen durch erhöhte Qualität der entwickelten Software in die Bewertung des Nutzens einzubeziehen. Diese Bewertung ist eine wichtige Grundlage für die Entscheidung über die Investition in wiederverwendbare Software.

Daneben sind zwei Faktoren zu erwähnen, die entgegen weitverbreiteter Annahmen nur nachrangige Bedeutung haben:

- *Anreize*. Sie treten in der Bedeutung hinter den Abbau bestehender Hindernisse zurück.
- *Technische Details der Vorgehensweise*. Dies umfaßt insbesondere das nicht als notwendig erachtete objektorientierte Vorgehen, und die genaue Art Systemunterstützung mit Hilfe einer Wiederverwendungsbibliothek, die im Vergleich zum Inhalt der Bibliothek als unwichtig eingestuft wird.

4.4 Zusammenfassung

In diesem Kapitel haben wir die Ergebnisse einer empirischen Untersuchung des Status quo der Wiederverwendung bei einem internationalen Softwareprodukthersteller vorgestellt. Sie umfassen aktuelles Vorgehen, Einschätzung der Vor- und Nachteile, Hindernisse, notwendige Maßnahmen und ökonomische Aspekte. Wir haben auf Basis der Ergebnisse die Hypothesen aus 3.5.3 überprüft; zwölf davon wurden bestätigt, fünf widerlegt, vier teilweise bestätigt; drei Hypothesen bleiben offen und werden in den folgenden Kapiteln überprüft. Außerdem haben wir die Ergebnisse im Zusammenhang interpretiert und verallgemeinernd folgende drei allgemeine Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software abgeleitet: (1) Investition in wiederverwendbare Software tätigen, (2) alle Merkmale der Wiederverwendbarkeit berücksichtigen und (3) Nutzen umfassend bewerten.

Kapitel 5

Fallstudien zur Softwaretechnik

Ziel dieses Kapitels ist, empirisch fundierte softwaretechnische Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software zu bestimmen.

Das Kapitel stellt die Ergebnisse von drei Fallstudien zur Softwaretechnik in Individualentwicklungsprojekten der sd&m AG vor. Dabei wird sowohl die Herstellung als auch die Nutzung untersucht. Ein besonderer Schwerpunkt liegt auf der Architektur. Die Ergebnisse werden im Vergleich analysiert und Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software abgeleitet.

Inhalt: **Seite**

5.1 Einleitung

In diesem Kapitel wird anhand von drei Fallstudien eine empirische Untersuchung der Wiederverwendung bei der sd&m AG beschrieben. Sie dient dazu, das schwache empirische Fundament hinsichtlich der softwaretechnischen Aspekte der industriellen Praxis, besonders der Architektur, zu verbessern.

In der Einleitung stellen wir kurz das Unternehmen sd&m AG vor. Darauf geben wir einen Überblick über die drei Fallstudien, erläutern das methodische Vorgehen und fassen schließlich den Stand der Forschung und den eigenständigen Beitrag zusammen.

In weiteren Unterkapiteln geben wir zunächst detailliert die Ergebnisse der Fallstudien wieder. Dann analysieren wir sie vergleichend, ziehen Schlußfolgerungen für die zu überprüfenden Hypothesen und leiten softwaretechnische Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software ab. Abschließend fassen wir das Kapitel zusammen.

5.1.1 Das Unternehmen und sein Markt

Die sd&m AG [sww] ist ein Softwarehaus mit Sitz in München und acht weiteren Niederlassungen in Deutschland (Berlin, Bonn, Düsseldorf, Frankfurt, Hamburg, Hannover, Köln, Stuttgart) sowie einer in der Schweiz (Zürich); sie hat ca. 800 Mitarbeiter.

Geschäftsfeld der sd&m AG ist die Entwicklung individueller betrieblicher Informationssysteme. Kunden sind überwiegend große Unternehmen aus verschiedenen Industrien, u.a. Banken und Versicherungen, Automobilhersteller, Telekommunikationsunternehmen, Logistikunternehmen und Reiseveranstalter.

5.1.2 Übersicht über die Fallstudien

Bei der sd&m AG führten wir drei Fallstudien zu Projekten durch, in denen systematisch wiederverwendbare Software (Definition siehe 2.1.2) hergestellt wurde. Die Projekte wählten wir aus, weil in substantiellem Umfang Software entwickelt wurde, für die systematische Wiederverwendung angestrebt war; als substantiell wurde der Umfang dann eingestuft, wenn der Aufwand für die ursprüngliche Entwicklung mehr als 3 Bearbeitermonate betrug. Die erfolgreiche Wiederverwendung der hergestellten Software war kein Kriterium für die Auswahl: zwei erfolgreiche Projekte stehen einem gegenüber, bei dem die Wiederverwendung scheiterte. Einen Überblick über die Fallstudien gibt Tabelle 5-1.

Das Projekt Quasar Database Interface erreichte eine erfolgreiche Wiederverwendung der a priori wiederverwendbaren Datenbankzugriffsschicht in fünf Fällen. Im Projekt Data Warehouse Loader wurde ein Data Warehouse-Ladewerkzeug a posteriori wiederverwendbar gemacht, das zweimal erfolgreich ge-

nutzt wurde. Keine erfolgreiche Nutzung gab es bei dem im Projekt GO a priori wiederverwendbar entwickelten Framework für die Client-Seite eines Hostsystems (Client-Framework).

Name des Projekts	Entwickelte wiederverwendbare Software	Wiederverwendbarkeit	Erfolgreiche Nutzungen
Quasar Database Interface (QDI)	Datenbankzugriffsschicht	A priori	Fünf
Data Warehouse Loader (DWL)	Data Warehouse-Ladewerkzeug	A posteriori	Zwei
Global Ordering (GO)	Client-Framework	A priori	Keine

Tabelle 5-1: Übersicht über Fallstudien bei sd&m AG

5.1.3 Methodisches Vorgehen

Als wissenschaftliche Untersuchungsmethode haben wir die Form der Fallstudie gewählt, deren Ziele und theoretische Grundlagen in 4.1.3 erörtert werden.

In den Fallstudien wurde sowohl das Projekt untersucht, in dem die wiederverwendbare Software entwickelt wurde, als auch eines der Projekte, in dem sie genutzt wurde. Wir führten jeweils mehrere Interviews mit Entwicklern aus beiden Projekten, studierten die zur Verfügung stehenden Dokumente und arbeiteten uns in den Quellcode ein. Ein Schwerpunkt der Untersuchung war die Architektur, die wir auf die Zerlegung in Komponenten und die Trennung der Zuständigkeiten untersuchten; dabei wandten wir das Konzept der Software-Kategorien an. Die Ergebnisse der Fallstudien werden vergleichend analysiert und Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software abgeleitet. Die Ableitung erfolgt induktiv (vgl. [Eis89, Hei95]), d.h. sie wird jeweils durch grundsätzliche Überlegungen untermauert.

5.1.4 Stand der Forschung und eigener Beitrag

Die Analyse der Literatur zu Wiederverwendung ergibt Forschungsbedarf insbesondere, was die empirische Untersuchung der Architektur wiederverwendbarer Software angeht (vgl. 3.5.1). In bisher veröffentlichten Fallstudien wird sie entweder gar nicht [Joo94, Gri94, GrW95, Lim94] oder auf einer zu groben Granularitätsebene [MeL95,MCG96, Ill99] behandelt. In den breiter angelegten Untersuchungen [FrF95, HaP97, MET99] wird sie allenfalls oberflächlich und unsystematisch betrachtet (vgl. 3.3).

Einen wissenschaftlichen Beitrag leisten wir dadurch, daß wir in diesem Kapitel in drei Fallstudien detaillierte Daten zur Architektur erheben. Bei der Analyse der Architektur steht die Variabilität im Mittelpunkt, die für die Wieder-

verwendbarkeit zentral ist (vgl. 2.1.2) und auch ein allgemeines Qualitätsmaß darstellt (vgl. [BSi00]). Wir untersuchen die Zerlegung jeweils auf Komponenten und die Realisierung der Kommunikation der Komponenten, insbesondere die Entkopplung über Schnittstellen. Das Ausmaß der Trennung der Zuständigkeiten bestimmen wir, indem wir die Komponenten nach Software-Kategorien klassifizieren (siehe 2.3.6). Darüber hinaus zeigen wir die enge Verbindung zwischen Architektur und dem Erfolg der Wiederverwendung.

5.2 Ergebnisse der Fallstudien

Im folgenden werden die Ergebnisse der drei Fallstudien vorgestellt. Die Gliederung ist dabei jeweils wie folgt: zunächst geben wir einen Überblick über das Projekt, in dessen Rahmen die wiederverwendbare Software entwickelt wurde. Dann beschreiben wir die ursprüngliche Entwicklung einschließlich eventueller Besonderheiten. Im Anschluß gehen wir auf die Nutzung der wiederverwendbaren Software ein. Abschließend ziehen wir ein Fazit der jeweiligen Fallstudie.

5.2.1 Projekt 1: Datenbankzugriffsschicht Quasar Database Interface (QDI)

Die sd&m AG arbeitet seit einiger Zeit daran, Lösungen für häufig auftretende Architekturprobleme zu erarbeiten ([BeS01a], 1). Aktuell sind diese Bemühungen im Projekt *Quasar*[59] gebündelt. Das Quasar-Projekt hat das Ziel, "im Rahmen einer Musterarchitektur Lösungen für Kernfragen der Architektur betrieblicher Informationssysteme bereitzustellen".

Das Quasar Database Interface (QDI) ist eine wiederverwendbare Datenbankzugriffsschicht, die im Rahmen des Quasar-Projekts entwickelt wurde. Bisher wurde das QDI im ersten Jahr nach der Fertigstellung fünfmal in verschiedenen Projekten erfolgreich genutzt. Die Nutzung im ersten dieser Projekte – WebLog – haben wir näher untersucht.

Wir führten im Rahmen der Fallstudie mehrere Interviews mit Herstellern und Nutzern des QDI; dabei verwendeten wir Interview-Leitfäden. Darüber hinaus analysierten wir eine Reihe von Dokumenten [BeS01a, BeS01b, SiF01, Ben01a, Ben01b, Rid01, RiU01a, RiU01b] und Teile des Quellcodes, der vollständig zur Verfügung stand.

Überblick des Projekts

Das QDI wurde *a priori* wiederverwendbar entwickelt. Das Quasar-Projekt, in dessen Rahmen die ursprüngliche Entwicklung stattfand, wurde nicht im Auftrag eines externen Kunden durchgeführt, sondern war ein internes Projekt der

[59] Quasar ist ein Akronym für Quality Software Architecture ([BeS01a], 1)

sd&m AG. Quasar liegt die bei betrieblichen Informationssystemen übliche Drei-Schichten-Architektur zugrunde (siehe 2.5.3). Das QDI stellt eine Musterarchitektur für die Datenverwaltungsschicht dar. Die Bestandteile sind ([BeS01a], 1):

- Drei externe Schnittstellen zur Manipulation, Transaktionskontrolle und Abfrage, die für den Anwendungskern zur Verfügung gestellt werden.
- Eine volle Beispielimplementierung und eine Dummy-Implementierung. Die Dummy-Implementierung simuliert die Datenbank im Hauptspeicher und kann für parallele Entwicklung des Anwendungskerns und zu Testzwecken verwendet werden. Sie ermöglicht, die gesamte Anwendung ohne Datenbankanbindung zu programmieren und diese zu beliebigem Zeitpunkt durch Austausch der Dummy-Implementierung durch die volle Implementierung herzustellen. Beide Implementierungen sind in der Programmiersprache Java geschrieben.

Das QDI ist eine Datenbankzugriffsschicht, die es ermöglicht, einen in einer objektorientierten Programmiersprache geschriebenen Anwendungskern von den technischen Spezifika einer relationalen Datenbank zu entkoppeln (siehe Abbildung 5-1). Dabei besteht die grundsätzliche Schwierigkeit darin, die Objekte des Anwendungskerns auf die Tabellenstruktur der Datenbank abzubilden, wobei z.B. komplexe objektorientierte Datentypen auf wenige einfache Datentypen in der Datenbank abgebildet werden müssen ([BeS01a], 3). Diese objektrelationale Abbildung übernimmt das QDI. Dazu werden an den externen Schnittstellen abstrakte Funktionen zur Verfügung gestellt:

- Die Manipulationsschnittstelle stellt die drei Methoden `insert`, `remove` und `update` zur Verfügung
- Die Transaktionsschnittstelle stellt die drei Methoden `beginTransaction`, `commit` und `rollback` bereit
- Die Abfrageschnittstelle dient der Formulierung von Suchbedingungen.

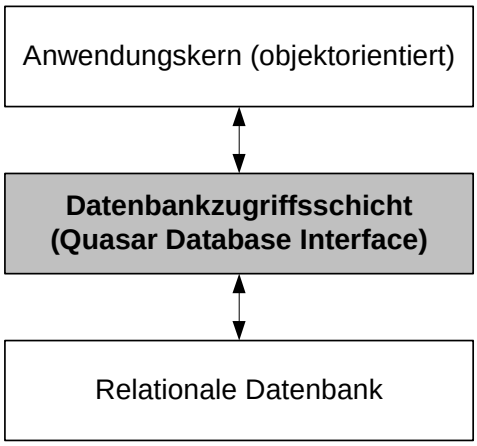

Abbildung 5-1: Schematische Darstellung des Datenbankzugriffs mit QDI (nach [BeS01a])

Die Anwendung kann gegen diese Schnittstellen programmiert werden, als gäbe es keine Datenbank. Die Datenbankanbindung wird dann durch eine Implementierung der Schnittstellen, z.B. die Quasar-Beispielimplementierung, zur Verfügung gestellt.

Das QDI wurde unter anderem in einem für ein Logistikunternehmen in der Fast Food-Branche durchgeführten Projekt namens WebLog erfolgreich wiederverwendet.

Herstellung des Quasar Database Interface

Das ursprüngliche Entwicklungsprojekt hatte zwei Phasen. In der ersten Phase, die einem über einen längeren Zeitraum verteilten Aufwand von ca. drei Bearbeitermonaten entsprach, wurde eine Vorgängerversion des QDI namens Quasar Datenverwaltung (QDV) entwickelt. Es handelte sich um ein sd&m-internes Projekt, das in Kooperation mit der Fachhochschule Rosenheim durchgeführt wurde. Ziel war dabei, die vorhandenen konzeptionellen Überlegungen – im Sinne eines *proof of concept* – durch eine Beispielimplementierung zu validieren; daneben sollte die QDV den Bedarf einer leicht verständlichen Datenbankzugriffsschicht für die Programmierausbildung decken, mit der auch Anfänger innerhalb weniger Tage eine persistente Anwendung entwickeln können. Die Implementierung erfolgte in der Programmiersprache Java unter Verwendung der JDBC-Schnittstelle (Java Database Connectivity). Als Datenbanksystem wurde Microsoft Access verwendet. Das verwendete Betriebssystem war Microsoft Windows NT. Motiviert war das Projekt dadurch, daß in den meisten Kundenprojekten Datenbankzugriffsschichten benötigt und diese regelmäßig von Grund auf neu entwickelt wurden. Deshalb wurde beschlossen, grundlegende Prinzipien herauszuarbeiten und darauf aufbauend eine wiederverwendbare Zugriffsschicht zu entwickeln [Sie01b]. Oberstes Ziel war dabei, nur die wesentlichen Anforderungen abzudecken; deshalb wurde versucht, die von der QDV implementierten externen Schnittstellen minimal zu halten.

Die QDV wurde im Jahr 1999 fertiggestellt und seither in der Programmierausbildung der sd&m AG bei den sog. Programmer Schools eingesetzt. Diese Einsätze fungierten als Beta-Testphase; in der Folgezeit wurde die QDV kontinuierlich gewartet, d.h. Fehler beseitigt, und weiterentwickelt.

Ende des Jahres 2000 wurde entschieden, aufbauend auf den Erfahrungen mit der QDV, eine neue Zugriffsschicht mit dem Namen QDI zu entwickeln. Für das QDI wurde ein neues Ziel gesetzt: die Mehrfachverwendung (vgl. 2.1.6) in Kundenprojekten. Der Aufwand für die Entwicklung (ohne Wartung und Weiterentwicklung) betrug dreieinhalb Bearbeitermonate. Die Programmierung erfolgte auch hier in Java unter Nutzung von JDBC; die Systemumgebung war dieselbe wie bei der QDV (s.o.).

Das QDI unterscheidet sich in der Außensicht – d.h. den zur Verfügung gestellten externen Schnittstellen – bis auf eine Ausnahme nicht von der QDV. Die Ausnahme betrifft die Art der Repräsentation von Objekten und die Art, wie die Transformation durchgeführt wird. Während bei der QDV der Code für die Transformation jeder Klasse hinzugefügt werden mußte, wird er bei dem QDI zentral im Repository abgelegt, was den Vorteil hat, daß die Klassen selbst zum Zweck der Persistenz nicht verändert werden müssen. In beiden Fällen handelt es sich um generierbaren Code der Kategorie R-Software. Bei der QDV war die Metainformation über die Struktur einer Klasse in den Code selbst einzufügen. Demgegenüber wird beim QDI die Struktur der Klassen in XML (Extensible Markup Language) beschrieben; aus dieser Beschreibung

wird der R-Code generiert. Auch die wesentlichen Komponenten blieben gleich. Eine große Anzahl von Verbesserungen wurde allerdings auf der Detailebene vorgenommen. Das Ziel, nur wesentliche Anforderungen abzudecken, wurde beibehalten, ebenso die davon abgeleiteten minimalen Schnittstellen (vgl. [SiF01], 9).

Großer Wert wurde bei der Entwicklung der QDI-Beispielimplementierung im Sinne der Wiederverwendbarkeit auf die Entkopplung der Komponenten durch Schnittstellen und die Trennung der Software-Kategorien gelegt (Details siehe unten).

Das QDI wurde bei Programmer Schools eingesetzt, wo es die QDV ablöste. Dort half es den Teilnehmern dabei, beim objektorientierten Programmieren schnell eine Datenbankverbindung für die Persistenz herzustellen. Gleichzeitig entsprach dieser Einsatz einer Beta-Testphase, denn die Erkenntnisse aus der Nutzung flossen in die Entwicklung zurück. Außerdem wurde nebenbei im Laufe der Zeit eine ganze Reihe potentieller späterer Nutzer im Umgang mit dem QDI geschult.

Daneben wurde es auf eine weitere Art praktisch getestet: es wurde frühzeitig Projekten zur Nutzung zur Verfügung gestellt. So gelang es auch hier, einen Rückfluß der Erfahrungen bei der Nutzung für die Weiterentwicklung sicherzustellen und diese iterativ im Entwicklungsprozeß zu berücksichtigen.

Der gesamte Aufwand für die Entwicklung des QDI kann als Investition in Wiederverwendbarkeit interpretiert werden, da es bei der Neuentwicklung hauptsächlich um die Herstellung der Mehrfachverwendbarkeit (vgl. 2.1.6) ging, wobei die Entwicklung der QDV dem ursprünglichen Entwicklungsprojekt entspricht. Der dafür angefallene Aufwand von drei Bearbeitermonaten enspricht also den Kosten der ursprünglichen Entwicklung K_e (vgl. 2.1.5), der Aufwand von dreieinhalb Bearbeitermonaten für die Entwicklung des QDI entspricht der Investition in Wiederverwendbarkeit $I_{wv.}$ Im Sinne der Nutzbarkeit für Entwickler wurde eine Reihe von Maßnahmen durchgeführt, für die insgesamt ein Aufwand von zwei Bearbeitermonaten anfiel: Zur Dokumentation wurden drei Dokumente erstellt: [SiF01] gibt einen Überblick über die Funktionalität des QDI im Zusammenhang von Quasar. In [BeS01b] wird die Funktionalität im Detail beschrieben. Auf die Implementierung und deren Nutzung im Detail geht [BeS01a] ein. Zur Sicherstellung von Qualität und Einheitlichkeit des Codes wurde ein Code Review durchgeführt. Für Regressionstests wurde eine Reihe von Testfällen bereitgestellt, die mit Hilfe des Tools JUnit genutzt werden können [Ben01b]. Darüber hinaus steht eine ausprogrammierte und getestete Beispielanwendung (Telecom Billing System) zur Verfügung, an der die Funktionsweise studiert werden kann. Alle diese Maßnahmen dienten der Verständlichkeit. Schließlich hielten die ursprünglichen Entwickler einige firmeninterne Vorträge, um auf das QDI aufmerksam zu machen und Einsatzweise und Vorteile zu erklären, was die Verfügbarkeit erhöhte. Auch die Schulung in der Benutzung des QDI bei den Programmer Schools diente der Verständlichkeit; der dafür anfallende Aufwand wurde jedoch nicht dem QDI-Entwicklungsprojekt angelastet.

Nutzung des Quasar Database Interface

Im Rahmen des Projekts WebLog wurde ein Web-Frontend für ein Warenwirtschaftssystem entwickelt. Für die Datenbankzugriffsschicht wurde das QDI eingesetzt. Die Systemumgebung des Projektes war in wesentlichen Punkten anders als bei der Entwicklung: als Datenbanksystem wurde Oracle 8i eingesetzt, das Betriebssystem auf dem Server war HP-UX (Unix-Variante der Firma Hewlett-Packard). Keine Änderung gab es bei der verwendeten Programmiersprache (Java) und dem Client-Betriebssystem (Microsoft Windows NT). Die notwendige Anpassung an die geänderte Datenbank wurden erfolgreich durchgeführt. Die Portierbarkeit der Programmiersprache Java machte den Wechsel auf das neue Betriebssystem ohne Aufwand möglich.

Der Aufwand für das gesamte Projekt betrug 15 Bearbeitermonate. Der Einsatz des QDI erfolgte auf Wunsch des Kunden, weil dadurch schneller und kostengünstiger entwickelt werden konnte. Der eingesparte Aufwand betrug drei Bearbeitermonate und kam in voller Höhe dem Kunden zugute, der nur den Anpassungsaufwand für die QDI bezahlen mußte. Neben der Ersparnis bot das QDI durch den Workspace Funktionalität, die bei einer Neuentwicklung nicht vorgesehen worden wäre, aber die Entwicklung des Anwendungskerns vereinfachte.

Neben der Erstellung eines neuen API-Experten für Oracle wurden im Rahmen der Anpassung eine Connection Pool-Funktion und eine optimierte Suchfunktion im Cache eingebaut. Darüber hinaus wurden einige Programmierungsfehler beseitigt und einige Verbesserungen am Code vorgenommen, die sich aus der praktischen Nutzung ergaben. Diese Anpassungen flossen alle an das QDI-Entwicklungsteam zurück. Die Fehlerkorrekturen und Verbesserungen wurden im Rahmen des Wartungsprozesses eingepflegt; der neue API-Experte steht für Wiederverwendung in einem anderen Projekt zur Verfügung. Die folgenden Nutzungsprojekte profitierten bereits von diesen Verbesserungen. Über den Einbau des Connection Pools und der optimierten Suchfunktion wurde noch nicht entschieden.

Explizit wurde von den Nutzern in einer abschließenden Bewertung die gute Erweiterbarkeit des QDI hervorgehoben [Rid01]. Die Dokumentation und die zur Verfügung stehenden Beispiele wurden als ausreichend bewertet. Bemängelt wurde die unzureichende Abdeckung der Regressionstests. Zwei der beteiligten Entwickler hatten vor dem Projekt eine Programmer School absolviert und waren dadurch im Umgang mit dem QDI bereits geschult.

Das typische Vorgehen beim Erschließen bisher unbekannter Teile der QDI beschrieb das Nutzer-Team in einem Interview folgendermaßen: zunächst prüfte es, ob die zur Verfügung stehenden Beispiele herangezogen werden konnten. War der in Frage stehende Fall von den Beispielen nicht abgedeckt, so konsultierte das Team die Dokumentation. Waren die Erklärungen darin nicht ausreichend, so wurde der Code analysiert. Schließlich bestand als ultima ratio die Möglichkeit, bei den QDI-Entwicklern Rat einzuholen.

Architektur des Quasar Database Interface

Wir untersuchen die Architektur der Beispielimplementierung des QDI, die in Abbildung 5-2 zu sehen ist. Sie ist in die folgenden, durch Schnittstellen entkoppelten Komponenten zerlegt: `Workspace`, `Query`, `DataStore`, `StatementManager` und `API-Expert`. Dabei wurde ein hoher Anteil von 0-Software erreicht. Im Sinne einer Trennung der Zuständigkeiten ist die Abhängigkeit vom technischen API in der `API-Expert`-Komponente konzentriert; sie ist die einzige Komponente des QDI, die nicht zur 0-Software-Kategorie gehört. Wird die genutzte Datenbank geändert, so muß lediglich diese Komponente angepaßt bzw. neu entwickelt werden; diese Anpassung wurde im WebLog-Nutzungsprojekt erfolgreich durchgeführt (s.o.).

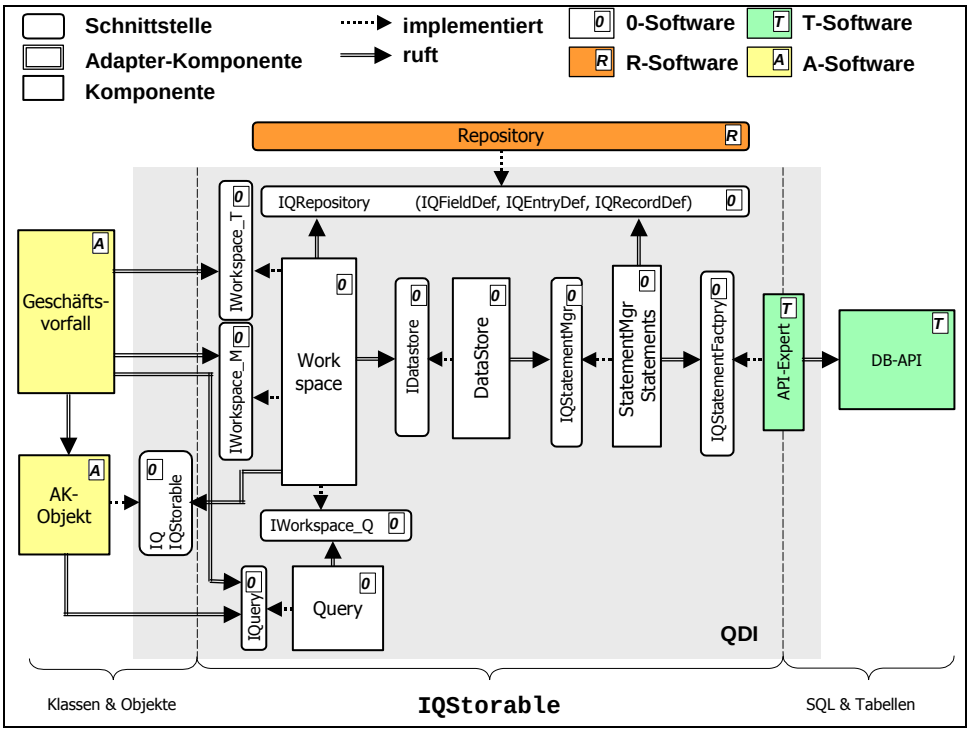

Abbildung 5-2: Architektur des Quasar Database Interface (nach [BeS01b])

Das QDI verbindet die Welt der A-Software (Anwendung: Klassen und Objekte, siehe Abbildung 5-2 unten) mit der Welt der T-Software (Datenbank: SQL und Tabellen). Die Daten werden zwischen diesen Welten hin- und hertransformiert, wobei sie als Zwischenstufe immer in Objekte der Kategorie 0-Software überführt werden, die das Interface `IQStorable` implementieren, das u.a. das Lesen und Schreiben der Felder ermöglicht. In der 0-Software-Welt des QDI wird also nur mit Objekten vom Typ `IQStorable` gearbeitet. Entscheidend dafür, daß diese Transformation mit 0-Software durchgeführt werden kann, ist die Abstraktion, die durch die externen Schnittstellen geleistet wird; das ermöglicht, innerhalb des QDI ausschließlich mit Objekten zu hantieren, die die Schnittstelle `IQStorable` (Kategorie 0-Software) implementieren. Die externen Schnittstellen (`IWorkspace_M`, `IWorkspace_T`, `IQuery`, `IQStatementFactory`) gehören alle zur Kategorie der 0-Software, d.h. sie abstrahieren in geeigneter Form von den Spezifika der Anwendung

(`IWorkspace_M`, `IWorkspace_T`, `IQuery`) bzw. des technischen API (`IQStatementFactory`). Die für die Transformation notwendige Strukturinformation ist im Repository abgelegt; dies ist die einzige Komponente, die von Anwendung und Technik beeinflußt ist. Sie ist allerdings in stereotyper, generierbarer Form als R-Software implementiert. Als einzige Bedingung muß jedes speicherbare Objekt eine der Schnittstellen `IQ` oder `IQStorable` implementieren, um das QDI vollständig von der Anwendung zu entkoppeln. Die `IQ` -Schnittstelle dient dazu, das Objekt speicherbar zu machen, indem seine Attribute lesbar gemacht werden. Die davon abgeleitete `IQStorable` -Schnittstelle erlaubt darüber hinaus, Transaktionsmanagement und optimistisches Locking der Datenbank durchzuführen.

Ebenso wie die Entkopplung über Schnittstellen erlaubt, einzelne Komponenten, also Schnittstellen-Implementierungen, auszutauschen, kann die gesamte QDI-Implementierung gegen eine andere ausgetauscht werden, die lediglich die externen Schnittstellen implementieren muß.

Der `Workspace` dient der Realisierung von Transaktionen, der Zustandsverwaltung, der Sicherstellung der Objektidentität und der Ausführung der von der `Query`-Komponente angestoßenen Abfragen. Er ist instanziierbar; für jede Transaktion wird ein neuer `Workspace` erzeugt, der die Repräsentationen aller an der Transaktion beteiligten Objekte enthält. Jede Instanz des `Workspace` verwaltet eine konsistente Sicht der in ihm enthaltenen Daten: Änderungen werden dort sofort vorgenommen.

Der `Workspace` implementiert drei Schnittstellen: zur Anwendung hin die Schnittstellen `IWorkspace_M` und `IWorkspace_T`, zur `Query`-Komponente die Schnittstelle `IWorkspace_Q`. Durch diese Aufteilung werden die Schnittstellen jeweils klein und spezifisch gehalten.

Fazit von Projekt 1

Einige wichtige Ergebnisse diskutieren wir hier im Detail. Die wesentlichen Ergebnisse der Fallstudie sind in Tabelle 5-2 (siehe S. 138) zusammengefaßt.

Große Anstrengungen wurden unternommen, um die Anforderungen in einem überschaubaren Rahmen zu halten, was sich in den minimalen externen Schnittstellen ausdrückt. Dies dient sowohl der Beherrschbarkeit des Herstellungsprojekts als auch dem Verständnis der Nutzer. Bei der Entwicklung a priori wiederverwendbarer Software ist keine Beschränkung durch die Anforderungen eines konkreten Projekts gegeben, so daß stets eine latente Gefahr besteht, ein zu hohes Maß an Variabilität anzustreben, weil man hohe Wiederverwendbarkeit erreichen will. Das kann zum Scheitern das Projekts führen. Insbesondere deshalb ist eine freiwillige Selbstbeschränkung wichtig.

Einige weitere Maßnahmen wurden ergriffen, die dazu beitrugen, das Projekt beherrschbar zu halten: früh wurden produktive Versionen entwickelt, die einem Betatest ausgesetzt wurden. Die Erfahrungen der Nutzer flossen an die Entwickler zurück, so daß Fehler in Spezifikation und Architektur früh aufgedeckt und beseitigt wurden. Die Einschränkung der Anforderungen war nicht zuletzt deshalb möglich, weil die Endbenutzer keinen direkten Einfluß hatten (Anforderungen der Endbenutzer sind in der Regel schwer zu standardisieren).

Die Architektur ist wegen der Zerlegung in durch Schnittstellen entkoppelte Komponenten gut zu verstehen und anzupassen; beide Aspekte spielten eine Rolle. Die Trennung der Software-Kategorien und Maximierung des Anteils von 0-Software half bei der Verwirklichung des Geheimnisprinzips und dadurch bei der Minimierung des Änderungsaufwands bei Anpassungen. Mit der verwendeten generierbaren R-Software wurde ein für Transformationen sehr effektiver Variabilitätsmechanismus gewählt.

Eine substantielle Investition in die Wiederverwendbarkeit erlaubte es, notwendige Maßnahmen zur Sicherstellung der Nutzbarkeit für Entwickler durchzuführen. Sie umfaßten die Erstellung einer umfassenden Dokumentation, Qualitätssicherung und das Erstellen eines für das intuitive Verständnis wichtigen Anwendungsbeispiels. Nicht zuletzt dienten die Programmer Schools der Schulung für die Nutzung des QDI. Defizite gab es lediglich bei den – ebenfalls der Nutzbarkeit für Entwickler dienenden – Regressionstests, mit deren Hilfe nach einer Anpassung überprüft werden kann, ob das System korrekt funktioniert.

5.2.2 Projekt 2: Data Warehouse Loader

Im zweiten untersuchten Projekt wurde ein wiederverwendbares Data Warehouse-Ladewerkzeug mit Namen *Data Warehouse Loader* (DWL) entwickelt, das erfolgreich wiederverwendet wurde. Wir geben zunächst einen Überblick über das Projekt, gehen dann auf die Herstellung der wiederverwendbaren Software ein, anschließend auf ihre Nutzung und analysieren schließlich die Architektur. Wesentliche Erkenntnisse faßt das Fazit zusammen.

Im Rahmen der Fallstudie wurden mehrere auf Leitfäden basierende Interviews sowohl mit Herstellern als auch mit Nutzern durchgeführt. Außerdem wurden eine Reihe von Dokumenten [KrL98a, KrL98b, KHL98, Kre99, HöS99] und der Quellcode analysiert, der komplett zur Verfügung stand.

Überblick des Projekts 2

Der Data Warehouse Loader wurde ursprünglich für einen Kunden entwickelt, der ein Netzwerk für den Buchungsverkehr zwischen Reisebüros und Reiseveranstaltern betreibt. Der Data Warehouse Loader hat die folgende Funktionalität:

- Extraktion von Daten aus unterschiedlichen Datenquellen (u.a. Datenbanken und ISAM[60]-Dateien)
- Konsolidierung der Daten, um Inkonsistenzen zu vermeiden
- Transformation der Daten in das Zielformat (Änderung von Struktur und Format der Daten)
- Einstellen ("Laden") der transformierten Daten in das Data Warehouse.

Der Data Warehouse Loader wurde einmal erfolgreich wiederverwendet. Die Wiederverwendbarmachung erfolgte *a posteriori*: die Entscheidung dafür fiel

[60] Die Abkürzung steht für: Indexed Sequential-Access Method [Kin99].

erst, nachdem die ursprüngliche Entwicklung abgeschlossen war, weil in einem anderen Projekt ein ähnliches Werkzeug benötigt wurde. In diesem Projekt, dessen Kunde ein Energieunternehmen war, wurde der Data Warehouse Loader erfolgreich genutzt, nachdem er vorher in einem separaten Projekt wiederverwendbar gemacht worden war. Später wurde er in einem weiteren Projekt für ein Inkassounternehmen erfolgreich genutzt. Diese beiden Nutzungen waren über einen Zeitraum von 4 Jahren verteilt.

Herstellung des Data Warehouse Loaders

Da es sich um a posteriori wiederverwendbare Software handelt, beschreiben wir zunächst das ursprüngliche Entwicklungsprojekt und anschließend die Wiederverwendbarmachung.

Ursprüngliches Entwicklungsprojekt

Das ursprüngliche Entwicklungsprojekt hatte einen Umfang von 1,5 Bearbeiterjahren[61]. Anwendungsfunktionalität spielte, abgesehen von den Datenformaten, in den Anforderungen keine Rolle; insbesondere bestanden keine direkten Anforderungen von Endbenutzern[62]. Die genutzte Systemumgebung bestand aus dem Betriebssystem Windows NT der Firma Microsoft und einem Datenbanksystem (DBMS) der Firma Informix.

Variabilität war ein vorrangiges Ziel bei der ursprünglichen Entwicklung. Zum einen sollte eine spätere Portierung auf das Betriebssystem Unix ermöglicht werden, weil unklar war, ob der anfangs gewählte Windows-Server den Anforderungen hinsichtlich der Zuverlässigkeit und der Last genügen würde. Sollte dies nicht der Fall sein, so war vorgesehen, einen in der jeweiligen Hinsicht stärkeren Server einzusetzen. Damit war jedoch ein Wechsel auf Unix unumgänglich. Um diese Portierbarkeit sicherzustellen, wurden folgende Entscheidungen getroffen: als Programmiersprache wählte man ANSI C, weil es standardisiert war und insofern bei der Portierung keine Kompatibilitätsprobleme zu befürchten waren. Zum Speichern der Zwischenergebnisse wurden flache Dateien gewählt, da sie ein einfaches und gut portierbares Format darstellen. In einer zweiten Hinsicht war Variabilität von Bedeutung: neue Datenquellen sollten leicht zu integrieren sein. Außerdem sollte auch ein späterer Wechsel der Warehouse-Datenbank möglich sein. Daher wurde entschieden, sowohl sämtliche Datenquellen als auch die Warehouse-Datenbank durch Schnittstellen zu entkoppeln. Zusätzlich wurde vorgesehen, das Metadatenmodell für das Data Warehouse flexibel zu halten, da das ursprüngliche Modell nicht dem üblicherweise verwendeten Stern-Schema (vgl. [BeS97], 169) entsprach. Schließlich sollte auch ermöglicht werden, die Menge der im Data Warehouse zu speichernden Daten zu erweitern, d.h. fachliche Adaptierbarkeit hergestellt werden. Dafür mußte die fachliche Transformation variabel gemacht werden. Zu diesem Zweck wurde auch sie über Schnittstellen entkoppelt. Der Code für die Transformation wurde unter Nutzung von Metadaten generierbar gemacht.

---

[61] Dies entspricht den ursprünglichen Entwicklungskosten K_e.

[62] Zur Definition siehe Kapitel 2.5.4

Wiederverwendbarmachung

Für die Wiederverwendbarmachung nach Abschluß der ursprünglichen Entwicklung wurde ein eigenes Projekt durchgeführt, für das ein Budget von 40 Bearbeitertagen zur Verfügung gestellt wurde, das die Investition in Wiederverwendbarkeit I_{wv} darstellt. Es wurde für die folgenden Maßnahmen genutzt [Kre99]:

- Ein Code Review wurde durchgeführt. Dabei wurde der Programmierstil vereinfacht und Variablennamen und Kommentare verallgemeinert, so daß spezifische Bezüge zum ursprünglichen Projekt beseitigt wurden.
- Eine aus zwei Dokumenten bestehende Dokumentation wurde erstellt. Ein Dokument diente der Beschreibung des Konzepts und der Funktionalität aus der Sicht des Endbenutzers. Das andere war ein Handbuch für Programmierer, das diese in den Code einweist.
- Ein kleines ausführbares Programm mit beispielhaften Daten wurde erstellt, um die Funktionsweise zu illustrieren.

Sowohl die geringe Nutzungsrate von 0,5 Nutzungen p.a. als auch die im Vergleich zu den ursprünglichen Entwicklungskosten geringe Investition in Wiederverwendbarkeit machen den Data Warehouse Loader zu einem klassischen Beispiel für Wiederverwendbarkeit im engeren Sinne (vgl. 2.1.6).

Nutzung des Data Warehouse Loaders

Die Systemumgebung im Nutzungsprojekt setzte sich zusammen aus einem Solaris-Betriebssystem der Firma Sun Microsystems und einem Oracle-Datenbanksystem. Das Data Warehouse wurde nach dem Stern-Schema (s.o.) strukturiert, so daß das Modell für die Metadaten modifiziert werden mußte. Der Code für die fachliche Transformation der Datenformate mußte ebenfalls angepaßt werden. Da der DWL bereits von der ursprünglichen Entwicklung her über die notwendige Variabilität verfügte, wurde entschieden, zunächst wie oben beschrieben in seine Wiederverwendbarmachung zu investieren.

Die notwendigen Anpassungen an das neue Betriebssystem, die neue Datenbank, das neue Data Warehouse-Schema und die neuen fachlichen Datenformate wurden anschließend erfolgreich durchgeführt. Dazu wurde zunächst der Generator angepaßt. Anschließend konnte der Code der Adapter-Komponenten wie geplant mit Hilfe der geänderten Metadaten neu generiert werden. Der Aufwand für diese Anpassungen betrug insgesamt 20 Bearbeitertage [HöS99].

Architektur des Data Warehouse Loaders

Die verschiedenen Operationen des Data Warehouse Loaders (s.o.) werden in mehreren Schritten durchgeführt, wobei das Ergebnis jedes Zwischenschritts in Dateien gespeichert wird.[63]

Die Architektur des Data Warehouse Loaders (siehe Abbildung 5-3) ist wesentlich von dem Gedanken bestimmt, daß die zentrale Ablaufsteuerung ("Loader Engine") über Schnittstellen von allen fachlichen und technischen Spezifika

[63] Dies entspricht dem Pipes and Filters-Architekturstil (vgl. [ShG96], 21f).

entkoppelt ist. Dies entspricht einer prozeduralen Variante des von Gamma et al. für objektorientiertes Design postulierten Prinzips der "Programmierung gegen Schnittstellen" (siehe 2.3.3). Die Kommunikation mit den verschiedenen Datenquellen und -senken erfolgt dementsprechend über Schnittstellen, die in der Programmiersprache C mit Hilfe von Funktionszeigern realisiert werden. Im Einzelnen hat der Data Warehouse Loader folgende Schnittstellen:

- Die Extraktions-Schnittstelle für die Extraktion der Daten aus den verschiedenen Datenquellen (vgl. [KrL98b], 89f)
- Die Datenbank-Schnittstelle für den Import der Daten in die Warehouse-Datenbank (vgl. [KrL98b], 51ff)
- Die Transformations-Schnittstelle für die fachliche Transformation der Quelldaten in das Zielformat
- Die Format-Schnittstelle für die Transformation der technischen Repräsentation der Daten von Dateien/Datenbanktabellen in C-Strukturen und zurück
- Die Betriebssystemfunktionen-Schnittstelle für diverse Funktionen des Betriebssystems, u.a. den Zugriff auf die Dateien, in denen die Zwischenergebnisse abgelegt werden.

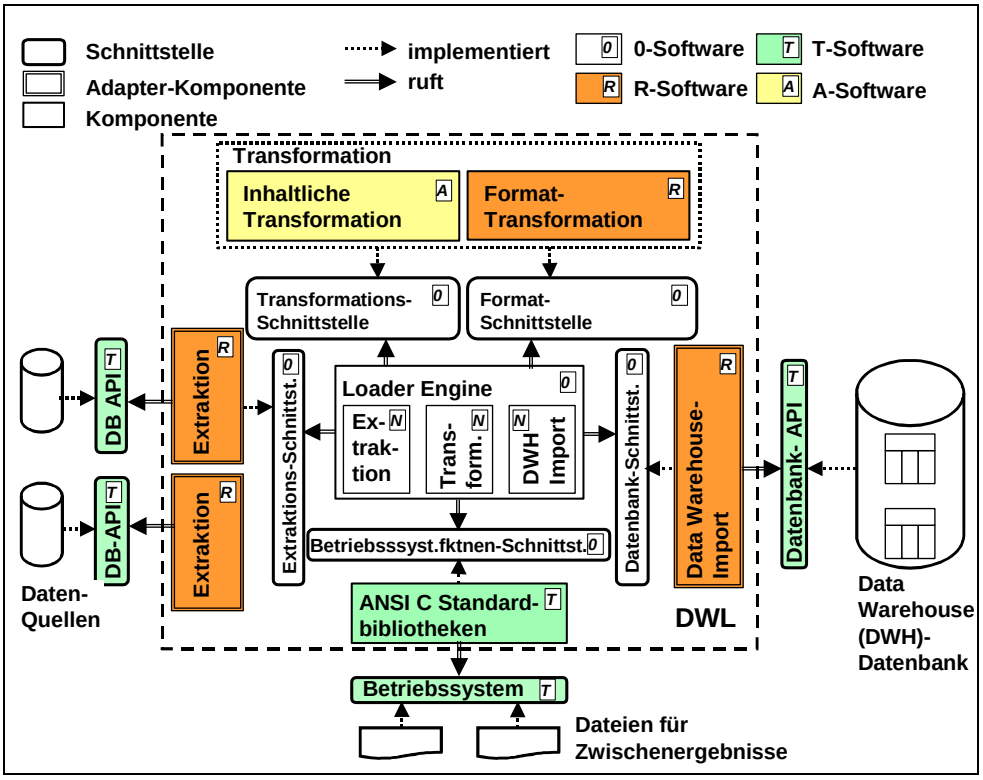

Abbildung 5-3: Architektur des Data Warehouse Loaders

Die Betriebssystemfunktionen-Schnittstelle wird durch die Standardbibliotheken[64] der Programmiersprache ANSI C implementiert. Die anderen Schnittstellen werden jeweils von entsprechenden Komponenten des Data Warehouse Loaders implementiert. Diejenigen Komponenten, die den Zugriff auf die jeweilige Datenquelle oder -senke realisieren, rufen wiederum deren Anwen-

[64] z.B. `stdio.h`

dungs-Schnittstelle (z.B. Datenbank-API) auf ([KrL98b], 53f, 89f). Für diese Module definieren wir den Namen *Adapter-Komponenten*. Sie arbeiten mit den Funktionen, die die jeweilige Datenquelle oder -senke an ihrer Anwendungs-Schnittstelle (API) zur Verfügung stellt und müssen daher auch für jede Datenquelle oder -senke individuell realisiert werden; wenn wie in dem untersuchten Projekt eine spezifische Adapter-Komponente für eine Informix-Warehouse-Datenbank vorhanden ist, so muß diese für den Einsatz einer Oracle-Datenbank überarbeitet oder ganz neu erstellt werden.

Die Schnittstellen zu den Datenquellen und -senken definieren allgemeine, abstrahierte Funktionen (z.B. "open", "read", "close" für die Extraktions-Schnittstelle oder "insert", "update" und "delete" für die Datenbank-Schnittstelle). Die technischen Spezifika der Realisierung sind vollständig in der implementierenden Adapter-Komponente versteckt. Daher ist die zentrale Ablaufsteuerung vollkommen frei von technischen Spezifika der jeweiligen Datenquelle oder -senke. Auch das notwendige Wissen über die Struktur der Daten ist in die Adapter-Komponenten ausgelagert, so daß die zentrale Ablaufsteuerung davon frei bleibt.

Die Adapter-Komponenten dienen der Transformation von Datenrepräsentationen und sind stereotyp. Demzufolge gehören sie zur Kategorie der R-Software. Ihr Code ist stereotyp; er unterscheidet sich für verschiedene Tabellen nur im Namen und der Struktur der Tabelle und im Namen und dem Typ der jeweiligen Tabellenspalten. Dieser Code kann vollständig generiert werden ([KHL98], 28); der Generator benötigt lediglich die o.g. Informationen über Name und Struktur der Datenbanktabellen. Im hier betrachteten Fall einer relationalen Datenbank gemäß SQL-Standard liegen sie als statische, vom Datenbanksystem verwaltete Metadaten vor: die Tabellennamen sind in der Tabelle "SYSTABLES" abgelegt, Namen und Datentypen der Spalten einer Tabelle sind in der Tabelle "SYSCOLUMNS" abgelegt ([KrL98a], 26f).

Da darüber hinaus auch die Funktionen zur Transformation von Quell- in Zieldaten über die Transformations- und Formatschnittstelle zur Verfügung gestellt werden, ist der Loader Engine vollkommen unbeeinflußt durch Spezifika der Anwendung, in diesem Fall dem Wissen über die Struktur der Quell- und Zieldaten. Die Komponente, die die inhaltliche Transformation durchführt, ist nur von der Anwendung beeinflußt und gehört somit zur Kategorie der A-Software. Die Komponente, die die Format-Transformation durchführt, gehört wie die Adapter-Komponenten zur Kategorie der R-Software und ist mit Hilfe der Metadaten vollständig generierbar ([KHL98], 28).

Bei der Loader Engine handelt es sich folglich um 0-Software. Neben der Wiederverwendbarkeit hat dies einen weiteren Vorteil, der diese Art des Designs ursprünglich motiviert hat: der Data Warehouse Loader ist problemlos erweiterbar, wenn neue Datenquellen hinzukommen oder eine andere Warehouse-Datenbank eingebunden werden soll. In diesem Fall muß lediglich die Adapter-Komponente angepaßt werden, die die jeweilige Schnittstelle implementiert. Dazu müssen API-Experte und Generator modifiziert oder neu geschrieben werden; die R-Software für die Datentransformationen kann anschließend mit Hilfe der Metainformationen komplett generiert werden.

Exemplarische Detailanalyse

Wir greifen beispielhaft den Import in die Warehouse-Datenbank heraus, um die Realisierung der Schnittstelle im Detail zu analysieren (für einen detaillierten Überblick über diesen Ausschnitt des Systems siehe Abbildung 5-4). Dabei zeigen wir, daß die Entkopplung von Komponenten in prozeduralen Sprachen zwar aufwendig, aber möglich ist. Der entsprechende Teil des Loader Engine ist in der Quelldatei `ldimport.c` realisiert. Diese greift auf die Datenbank-Schnittstelle zu, die vom Adapter-Modul für den Zugriff auf die Data Warehouse-Datenbank implementiert wird. Der Zugriff auf die Datenbank-Schnittstelle erfolgt in zwei Formen: über Basisfunktionen und über Funktionen, die für die jeweilige Datenbanktabelle spezifisch sind.

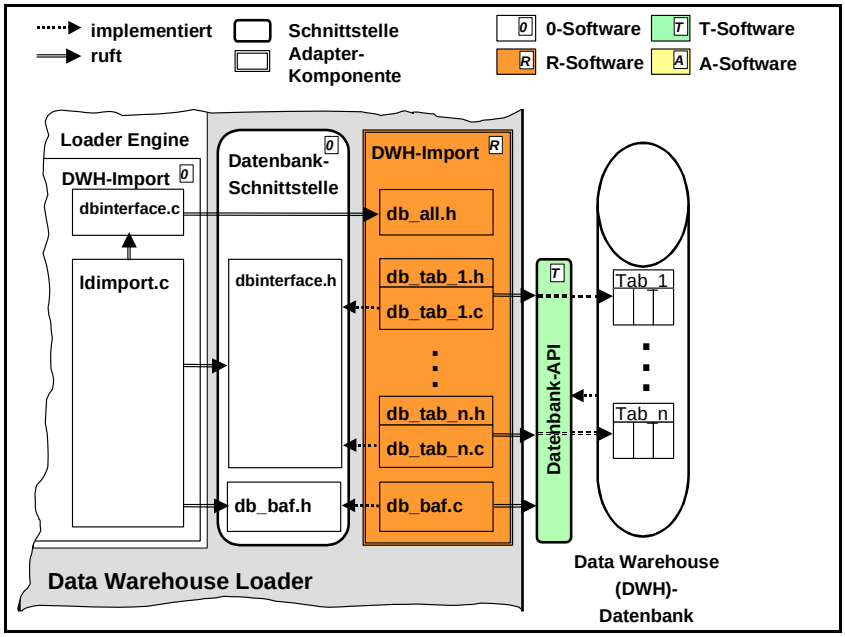

Abbildung 5-4: Architektur-Detail DWL (Datenbankschnittstelle)

Basisfunktionen, die unabhängig von der jeweiligen Datenbanktabelle sind, sind in der Header-Datei `db_baf.h` definiert; dieser Teil der Datenbank-Schnittstelle wird von der Quelldatei `db_baf.c` implementiert. Änderungen in der Datenbank lassen also die Schnittstelle (`db_baf.h`) unverändert; sie führen lediglich zu Änderungen in der Implementierung (`db_baf.c`).

Funktionen, die für die jeweilige Datenbanktabelle spezifisch sind, sind abstrakt in der Header-Datei `dbinterface.h` mit Hilfe einer Struktur von Funktionszeigern (`dbInterface_t`) definiert (in dem char-Array `sObject` ist der Name der Zieltabelle abgelegt):

```
typedef struct {
    char sObject[LEN_DBOBJECTNAME+1];
    int  (*fpDbNew)(); /* Import neuer Sätze */
    int  (*fpDbUpd)();  /* Import geänderter Sätze */
    int  (*fpDbDel)(); /* Import gelöschter Sätze */
    int  (*fpDbOpen)(); /* Cursor öffnen */
    int  (*fpDbFetch)();/* Cursor lesen */
    int  (*fpDbClose)();/* Cursor schließen */
} dbInterface_t;
```

Für jede der n Datenbanktabellen Tab_*i (i = 1, ... ,n)* gibt es eine Adapter-Komponente (bestehend aus der Header-Datei `db_tab_i.h` und der Quelldatei `db_tab_i.c`), die die in `dbInterface_t` definierten Funktionen zur Verfügung stellt und damit diesen Teil der Schnittstelle implementiert. Da `ldimport.c` nicht direkt auf sie zugreifen kann, wird ein Kunstgriff angewendet: `ldimport.c` ruft eine in `dbinterface.c` realisierte Hilfsfunktion auf, die einen Zeiger auf die zur jeweiligen Adapter-Komponente gehörende Struktur vom Typ `dbInterface_t` zur Verfügung stellt.

Im Einzelnen ist dies wie folgt realisiert. Die Header-Datei `db_all.h` enthält ein Array von Strukturen des Typs `dbInterface_t` (`dbInterfaces[]`), in denen die tatsächlichen Namen der Funktionen in den jeweiligen Adapter-Komponenten zu den verschiedenen Zieltabellen abgelegt sind. Die Quelldatei `dbinterface.c` stellt die Funktion `dbGetInterface()` zur Verfügung:

```
int dbGetInterface
(
    char                *sObject      /* in  */
    dbInterface_t       **pInterface  /* out */
)
```

Die Parameter von `dbGetInterface()` sind ein Zeiger auf den Namen der Zieltabelle (`*sObject`) und ein Zeiger (`**pInterface`) auf die Speicheradresse einer Struktur vom Typ `dbInterface_t`. Die Funktion sucht aus dem Array `dbInterfaces[]` die zu dem jeweiligen Namen gehörende Struktur heraus und gibt sie in (`**pInterface`) zurück. Damit hat `ldimport.c` über Zeiger Zugriff auf die Funktionen, die die Bearbeitung der jeweils gewünschten Datenbanktabelle ermöglichen.

Sämtliche beschriebenen Erkenntnisse gelten analog für die Extraktions-Schnittstelle, die exakt nach demselben Muster aufgebaut ist. Auch hier läßt sich die R-Software vollständig aus den Metadaten generieren.

Fazit von Projekt 2

Auf einige besonders bemerkenswerte Beobachtungen gehen wir hier im Detail ein. Eine Zusammenfassung wesentlicher Ergebnisse der Fallstudie findet sich in Tabelle 5-2 (siehe S. 138).

Der Aufwand für die Nutzung betrug nur 6,7 Prozent des Aufwands für die ursprüngliche Entwicklung, mit dem man bei einer Neuentwicklung hätte rechnen müssen. Der Nutzen in Form der eingesparten Kosten betrug also 280 Bearbeitertage.[65] Wenn man davon die Investition in die Wiederverwendbarkeit abzieht und davon absieht, daß der Nutzen wegen des späteren Geldflusses abgezinst werden müßte (vgl. 2.4.4), so ergibt sich ein Überschuß von 240 Bearbeitertagen. Dies ist ein außergewöhnlich gutes Ergebnis, vor allem im Hinblick darauf, daß nur ein einziges Nutzungsereignis stattfand. Der Grund dafür liegt in der hohen Variabilität, die in der ursprünglichen Entwicklung hergestellt wurde.

[65] Für ein Bearbeiterjahr setzen wir 200 Bearbeitertage an.

Besonders bemerkenswert ist, daß diese Variabilität nicht mit dem Ziel der Wiederverwendbarkeit, sondern dem der Wartbarkeit erzeugt wurde. Insbesondere stellten die ursprünglichen Entwickler fest, daß sie z.B. von Beginn an für neutrale Bezeichner hätten sorgen können, wenn Wiederverwendbarkeit ein Ziel gewesen wäre (vgl. [Kre99]); das hätte die Wiederverwendbarkeit zusätzlich erhöht. Dennoch war die Nutzung mit geringem Aufwand möglich; positiv wirkte sich dabei aus, daß keine Variabilität hinsichtlich der Anforderungen von Endbenutzern eingeplant werden mußte. Insgesamt zeigt dieses Beispiel, daß Wartbarkeit und Wiederverwendbarkeit ähnliche Anforderungen haben, d.h. zwei Seiten derselben Medaille sind.

Die Fallstudie belegt außerdem, daß Komponentenbildung und Entkopplung der Komponenten durch Programmieren gegen Schnittstellen auch mit prozeduralen Sprachen möglich sind; dies bestätigt unsere entsprechende Hypothese (siehe 3.4.3). Die Konstruktion wäre bei objektorientiertem Vorgehen aufgrund der möglichen Polymorphie bedeutend eleganter, aber grundsätzlich gleich.

Um dies zu illustrieren, übertragen wir den Datenbankzugriff sinngemäß auf die objektorientierte Programmiersprache Java. Die Datenbanktabellen würden hier als Objekte repräsentiert. Man definiert ein Interface `IDBTable`, das von sämtlichen Datenbanktabellen-Objekten implementiert wird:

```
interface IDBTable
{
 void insert( IDBElement element );
 void update( IDBElement element );
 void delete( IDBElement element );
 void openCursor();
 IDBElement fetchData();
 void closeCursor();
}
```

Dabei ist `IDBElement` das Interface, das von allen Datenbankelementen implementiert wird. Alle Datenbanktabellen implementieren `IDBTable` so daß die zentrale Ablaufsteuerung mit Objekten vom Typ `IDBTable` arbeitet und auf deren Methoden zugreift. Beispielhaft würde dies für eine Insert-Operation so aussehen:

```
public class LoaderEngine
{
 // Mehr Code...

 IDBTable dbtable;
 IDBElement dbelement;

 // Mehr Code...

 // Element in die Tabelle einfügen
 dbtable.insert( dbelement );

 // Mehr Code...
}
```

5.2.3 Projekt 3: Client-Framework

Im dritten der Projekte, die wir als Fallstudie untersuchen, wurde ein Framework (vgl. 2.1.3, 3.4) für die Client-Seite eines Großrechner-basierten Systems

entwickelt. Die geplante Wiederverwendung scheiterte. Wir geben zunächst einen Überblick über das Projekt, gehen dann auf die Herstellung der wiederverwendbaren Software ein, anschließend auf ihre Nutzung, analysieren die Architektur und ziehen abschließend ein Fazit der wesentlichen Erkenntnisse dieser Fallstudie.

Im Rahmen der Fallstudie führten wir mehrere Leitfaden-basierte Interviews sowohl mit Nutzern als auch mit Entwicklern, die nach dem Scheitern der Wiederverwendung ein Redesign durchführten. Von Letzterem versprachen wir uns ein besseres Verständnis der Gründe des Scheiterns. Außerdem studierten wir Dokumente zur ursprünglichen Entwicklung [Sal98, Shö98a, Shö98b, Fis98] und zum Redesign [WöS00] sowie den Quellcode, der komplett zur Verfügung stand.

Überblick des Projekts 3

Untersucht wurde ein Teilprojekt eines großen Projekts, in dem für einen Kunden aus der Automobilindustrie ein weltweites Bestellsystem entwickelt wurde. In diesem Teilprojekt wurde eine Anwendungskomponente des Gesamtsystems entwickelt, über die die Distribution der Fahrzeuge abgewickelt werden sollte. Daher nennen wir das Teilprojekt *DIST*. Das Gesamtprojekt wurde mit großem Erfolg abgeschlossen; die geplante Wiederverwendung des in DIST entwickelten Client-Frameworks jedoch scheiterte.

Das gesamte Bestellsystem besteht aus einer zentralen Host-Komponente und weltweit verteilten Client-Komponenten. Diese Komponenten sind über eine *logisches Netzwerk* genannte Middleware-Komponente verbunden, die das Intranet des Unternehmens nutzt.

Die Distribution war die erste Anwendung, für die eine Client-Komponente entwickelt werden sollte. Als Basis wurde ein aus einer sehr großen Zahl von Klassen ([Sal98], 29) bestehendes Framework entwickelt, das folgende Funktionen zur Verfügung stellen sollte ([Sal98], 28f):

- Querschnittsfunktionen wie Logging, Tracing, Assertions, Fehlerbehandlung etc.
- Serverseitige Anbindung an das logische Netzwerk
- Hilfsklassen für den Anwendungskern mit Datentypen und Stellvertreterobjekten für die zu rufenden Host-Module
- Hilfsklassen für die Benutzerschnittstelle (GUI), bestehend aus Dialogablaufsteuerung und den zu präsentierenden graphischen Elementen (Fenster, Widgets)

Es wurde entschieden, das Framework *a priori wiederverwendbar* zu entwickeln, um es auch bei der künftigen Entwicklung anderer Anwendungen als der Distribution einzusetzen. Dazu sollte die Architektur für andere Teilprojekte einsetzbar und stufenlos zu erweitern sein. Dies stand explizit in Gegensatz zum Auftrag des Teilprojekts DIST. Daher sollte der Zusatzaufwand für die Wiederverwendbarkeit gering gehalten werden; trotzdem wurde hinsichtlich der zu erreichenden Wiederverwendbarkeit keine klare Einschränkung gemacht

([Sal98], 8).[66] Es wurde also keine offizielle Entscheidung herbeigeführt, um zusätzliches Budget für die Wiederverwendbarmachung im Rahmen von DIST oder in einem separaten Projekt bewilligt zu bekommen.

Die Wiederverwendung des Frameworks bei der Entwicklung anderer Anwendungen war lediglich für die serverseitige Anbindung an das logische Netzwerk erfolgreich; für die Querschnittsfunktionen, den Anwendungskern und die Benutzerschnittstelle scheiterte sie. Es wurde ein Redesign notwendig, das mehrere Unzulänglichkeiten beseitigen sollte ([WöS00], 36), die sich teilweise bereits im Projekt DIST auswirkten:

- Die Portierung auf eine neue GUI-Bibliothek war nicht möglich
- Die rigide Dialogsteuerung ließ den Entwicklern keine ausreichende Flexibilität und mußte daher abgelöst werden.
- Das Framework war aufgrund seiner Komplexität nur schwer beherrschbar – dieses Problem bestand selbst in DIST – und konnte deshalb von den Entwickler nicht wie vorgesehen verwendet werden.

Herstellung des Client Frameworks

DIST hatte insgesamt einen Umfang von fünf Bearbeiterjahren, der den ursprünglichen Entwicklungskosten K_e entspricht. Als Programmiersprache wurde Java mit der GUI-Bibliothek AWT (Abstract Window Toolkit) eingesetzt. Da das Framework den Anwendungskern und die Benutzerschnittstelle umfaßte, hatten die Anforderungen der Endbenutzer starken Einfluß auf die Konstruktion.

Das Framework basiert auf einer Zerlegung in Komponenten ([Sal98], 29). Der Komponentenbegriff ist jedoch nicht klar bestimmt.[67] Als Variabilitätsmechanismus wurde Implementierungsvererbung eingesetzt. Spezialisierte Klassen sollten von allgemeineren abgeleitet werden und so eine Anpassung an spezifische Anforderungen erzielt werden (vgl. z.B. [Shö98a], 51, 63; [Shö98c], 25).

Die Konstruktions-Entscheidung, einen Anwendungskern für dezentrale fachliche Logik am Client vorzusehen, stellte sich gegen die für das gesamte Projekt gemachte Vorgabe, die fachliche Logik am Host zu konzentrieren ([Sal98], 7). Sie führte zu einer Erhöhung der Komplexität des Frameworks und hatte damit Anteil am Scheitern der Wiederverwendung.

Ziel beim Entwurf des Frameworks für die Benutzerschnittstelle war, typische Abläufe in abstrakter Form zu standardisieren, was zu einer sehr detaillierten Festlegung dieser Abläufe führte (vgl. [Sal98], 35). Das führte zu dem oben erwähnten Mangel an Flexibilität der Dialogsteuerung.

---

[66] Zitat aus [Sal99] (S. 8): "Aufwand dafür wurde im Sinne des Kunden äußerst restriktiv erbracht; aus einer gebotenen Weitsicht heraus aber immer dort, wo eine ‚geringfügig oder vordergründig billigere' Alternative eine spätere Erweiterung unmöglich gemacht hätte."

[67] Zitat aus [Sal99] (S. 29): "Wenn hier von *Komponente* gesprochen wird, ist einfach eine sinnvolle Einheit gemeint. I.d.R. sind das Klassen oder Klassenverbunde, manchmal ganze Packages, selten aber auch einfach ein Stück Code, das z.B. im Falle einer späteren Wiederverwendung evtl. in einer eigenen Klasse oder in einem separaten Package landen sollte."

Entsprechend der oben erwähnten Zielsetzung wurde keine explizite Investition in Wiederverwendbarkeit getätigt ($I_{wv} = 0$), was insbesondere zur Folge hatte, daß keine Maßnahmen im Sinne der Nutzbarkeit für Entwickler durchgeführt wurden.

Nutzung des Client Frameworks

Der Versuch, das Framework bei der Entwicklung weiterer Anwendungen einzusetzen, scheiterte, obwohl die Systemumgebung genau dieselbe war. In Interviews mit mehreren Entwicklern aus diesen Projekten kristallisierten sich zwei hauptsächliche Ursachen dafür heraus:

- Das Framework war zu unflexibel. Die bei der Nutzung notwendigen Änderungen waren zu schwer vorzunehmen.
- Die Entwickler verstanden das Framework nicht. Der Grund dafür lag in drei Unzulänglichkeiten. (1) Die Komplexität des Frameworks war zu groß: die Benutzerschnittstelle allein umfaßte 150 Klassen; zusätzlich erschwerte eine Vererbungshierarchie mit mehreren Ebenen die Übersicht. (2) Die Qualität der Dokumentation war unzureichend. Sie war zwar umfangreich – z.B. über 100 Seiten für die Benutzerschnittstelle –, aber schwer verständlich. Die Struktur trennte nicht zwischen der bereitgestellten Funktionalität und den Details ihrer Implementierung; außerdem gab es neben der Darstellung im Detail keine dem Überblick dienende Darstellung auf einer höheren Abstraktionsebene. (3) Es wurden keine Beispiele zur Erleichterung des Verständnisses zur Verfügung gestellt.

Architektur des Client Frameworks

Bei der Untersuchung der Architektur konzentrieren wir uns auf die Benutzerschnittstelle, da wegen der großen Anzahl der Klassen der Aufwand für die Analyse des Quellcodes sonst in keinem vertretbaren Verhältnis zum zusätzlichen Nutzen stünde.

Es zeigt sich, daß zwar eine Zerlegung in Komponenten vorliegt, daß diese aber größtenteils nicht über Schnittstellen entkoppelt sind. Das führt dazu, daß in vielen Fällen das Geheimnisprinzip verletzt wird, d.h. daß Änderungen nicht lokal auf die jeweilige Komponente begrenzt bleiben, sondern schwer kontrollierbare Auswirkungen haben. Eine zusätzliche Verletzung des Geheimnisprinzips stellt die oben erwähnte Vererbungshierarchie mit mehreren Ebenen dar (vgl. [BrS02]). Insgesamt wird durch diese Merkmale der Architektur die Anpassung bei der Nutzung sehr erschwert.

Eine Untersuchung der Software-Kategorien (siehe 2.3.6) ergibt darüber hinaus, daß keine Trennung der Zuständigkeiten vorliegt: ein großer Teil der Komponenten ist sowohl von der Anwendungslogik als auch von einer technischen Anwendungsschnittstelle (API) beeinflußt und ist folglich der Kategorie AT zuzuordnen. Das erhöht den Anpassungsaufwand bei der Nutzung, da die Zahl der von einer Änderung betroffenen Komponenten nicht minimiert wird.

Wir analysieren exemplarisch einen Ausschnitt der Architektur der Benutzerschnittstelle im Detail. Um zu zeigen, wie die beschriebenen Unzulänglichkeiten zu vermeiden wären, stellen wir diesen Ausschnitt einer Vorbild-Architek-

tur gegenüber. Wir wählen dafür das Model-View-Controller- (MVC-) Muster, das ein klassisches Entwurfsmuster für Benutzerschnittstellen ist ([BMR00], 124ff; [Bal99], 692ff). Es besteht aus drei Komponenten für Model, View und Controller. View und Controller sind beide von Anwendung und technischer Anwendungsschnittstelle (API) beeinflußt, jedoch stereotyp. Daher können sie als R-Software realisiert werden, die mit Hilfe von Meta-Information generierbar ist. Das Model hingegen sollte bei einer optimalen Realisierung nicht vom technischen API beeinflußt sein und als A-Software realisiert werden. Dazu muß es von View und Controller jeweils über eine Schnittstelle der Kategorie A entkoppelt werden; dafür ist das Observer-Entwurfsmuster ([GHJ94], 293 ff) geeignet, das eine entsprechende update-Methode zur Verfügung stellt. Den Vergleich zwischen der Framework-Architektur und der Vorbild-Architektur zeigt Abbildung 5-5.

In der GUI-Framework-Architektur hingegen sind alle drei MVC-Komponenten als AT-Software realisiert. Bei View und Controller ist der gleichzeitige Einfluß von Anwendung und Technik nicht zu vermeiden; die Folgen sind jedoch abzumildern, indem man sie wie in der Vorbild-Architektur als R-Software realisiert. Das Model jedoch enthält eine Window Management-Komponente, die von der Technik beeinflußt ist und so unnötigerweise die gesamte Model-Komponente zu AT-Software macht. Die Kommunikation von View und Controller mit Model und umgekehrt erfolgt durch direkte Aufrufe; sie könnte jedoch über Schnittstellen realisiert werden, wie die Vorbild-Architektur zeigt. Dadurch wäre eine Entkopplung möglich.

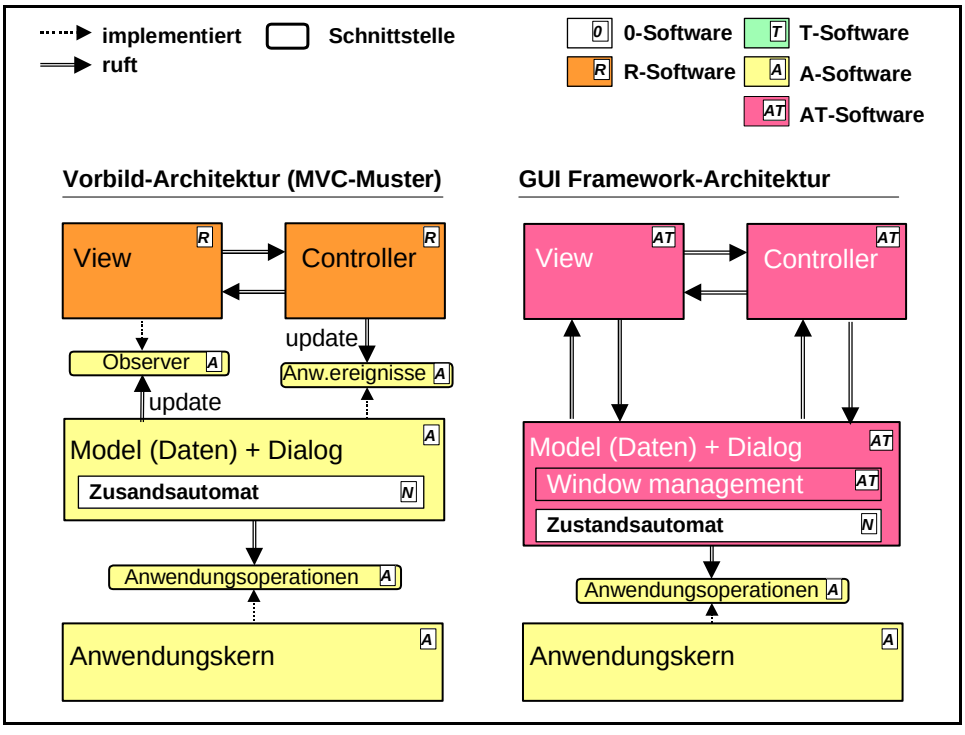

Abbildung 5-5: Architekturvergleich von klassischem MVC-Muster und GUI Framework

Fazit von Projekt 3

Einige zentrale Erkenntnisse werden hier im Detail erläutert. Eine Zusammenfassung wesentlicher Ergebnisse der Fallstudie ist in Tabelle 5-2 auf S. 138 zu finden.

Die Schwierigkeit der Aufgabe wurde offensichtlich unterschätzt. Das zeigt sich, wenn man die Grundlagen untersucht, auf denen das Projekt basierte: eine bewußte Investition in Wiederverwendbarkeit wird nicht gemacht; gleichzeitig wird ein hohes Maß von A-Priori-Wiederverwendbarkeit angestrebt. Dabei wird kein Versuch unternommen, die zu erfüllenden Anforderungen klar zu begrenzen. Darüber hinaus gibt es eine Schwachstelle, die sich in der Konstruktion negativ auswirkt: der Komponentenbegriff ist unklar.

Wesentlich zur Schwierigkeit der Aufgabe trug bei, daß entsprechend dem Anteil der Anwendungsfunktionalität die Anforderungen der Endbenutzer eine wichtige Rolle bei der Spezifikation des Frameworks spielten. Trotzdem wurde speziell im Fall der Dialogablaufsteuerung der Versuch unternommen, von der Komplexität dieser Anforderungen zu abstrahieren und sie zu standardisieren. Dieses Ziel ist jedoch grundsätzlich schwer zu erreichen, da es sich um inhärente, unsystematische Komplexität handelt, die nicht auf ein fundamentales Prinzip zurückgeführt werden kann [Bro87]. Eine Lösung kann nur dadurch herbeigeführt werden, daß der Endbenutzer dazu gezwungen wird, die Standardisierung zu akzeptieren und seinen Arbeitsablauf daran anzupassen. Entsprechender Zwang kann aufgrund der Marktmacht eines großen Produktanbieters gegeben sein[68]; für einen Anbieter wie die sd&m AG kollidiert er jedoch im Kern mit dem Anspruch, individuelle Software zu entwickeln, bei der sich der Nutzer gerade nicht den Einschränkungen unterwerfen muß, die ihm von Produkten auferlegt werden. Der beschriebene Versuch der Standardisierung ist also auch aus strategischen Gründen grundsätzlich fragwürdig.

Schwächen der Architektur und Versäumnisse bei der Implementierung führten zum Scheitern an der ohnehin schwierigen Aufgabe. Was die Architektur angeht, so ist die Einhaltung des Geheimnisprinzips insbesondere zur Erreichung der Variabilität wichtig, die eine grundlegende Eigenschaft wiederverwendbarer Software ist (siehe 2.1.2). Die Architektur des Client-Frameworks verletzt das Geheimnisprinzip auf drei Arten:

- Die Komponenten sind nicht über Schnittstellen entkoppelt
- Die Software-Kategorien werden nicht in ausreichendem Maße getrennt
- Vererbung wird als Variabilitätsmechanismus eingesetzt. Das führt dazu, daß Anpassungen nur in sehr engem Rahmen möglich sind.[69]

Zusätzlich wird die Nutzbarkeit für Entwickler (siehe 2.1.2) aufgrund mehrerer Unzulänglichkeiten erschwert:

---

[68] Ein klassisches Beispiel hierfür stellt das R/3-Produkt der Firma SAP AG dar, das bezüglich Buchhaltung und Controlling einen De-Facto-Standard setzte.

[69] Ein Zitat illustriert dies ([Shö98c], 31): "Falls die angenommene strukturelle Ähnlichkeit von Objektgeflecht und Oberfläche nicht gegeben sein sollte, sind die in diesem Kapitel beschriebenen Mechanismen nur bedingt anwendbar. In diesem Fall muß die Entwicklung wieder durch 'Handarbeit' erfolgen."

- Aufgrund der großen Anzahl von Klassen ist es für Nutzer sehr schwer, die Funktionsweise zu verstehen und sich einen Überblick zu verschaffen.
- Die Dokumentation ist schlecht strukturiert: (1) Keine Trennung von Außen- und Innensicht. (2) Keine hierarchische Struktur mit unterschiedlichen Abstraktionsebenen.
- Es stehen keine ausführbaren Beispiele zur Verfügung, die das intuitive Verständnis erleichtern würden.

Es sei angemerkt, daß lediglich für den letzten Punkt die höheren Kosten als Begründung ins Feld geführt werden können. Die beiden anderen Unzulänglichkeiten und sämtliche Schwächen der Architektur wären bei geändertem Vorgehen, das nicht notwendigerweise mehr Aufwand bedeutet hätte, grundsätzlich zu vermeiden gewesen.

5.3 Analyse der Ergebnisse und Schlußfolgerungen

Zur Analyse stellen wir zunächst die wesentlichen Ergebnisse der drei Fallstudien zusammenfassend gegenüber. Wir untersuchen, was sich daraus für die in 3.5.3 formulierten Hypothesen ergibt. Schließlich leiten wir aus den Ergebnissen verallgemeinernd Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software ab.

5.3.1 Vergleichende Analyse der Projekte

Wir vergleichen die drei beschriebenen Fallstudien anhand einer Reihe von Kriterien bezüglich allgemeiner Merkmale des jeweiligen Projekts sowie dem Vorgehen bei Spezifikation, Konstruktion und Programmierung. Wir wählen diejenigen Kriterien, in denen sich das nicht erfolgreiche Projekt (Client Framework) von mindestens einem der erfolgreichen (QDI, DWL) unterscheiden (mit Ausnahme der Komponenten-Architektur).

Nach folgenden Kriterien wird untersucht:

- *Allgemeine Projektmerkmale und Spezifikation*: Anzahl erfolgreicher Nutzungen, Zeitraum der Nutzungen, Ursprüngliche Entwicklungskosten K_e, Höhe der Investition in Wiederverwendbarkeit I_{wv}, Zeitpunkt der Wiederverwendbarmachung (a priori/a posteriori), gewählter Entwicklungsprozeß und Anzahl der in seinem Verlauf erstellten produktiven Versionen (d.h. Versionen, die einsatzfähig sind und die wesentliche Funktionalität besitzen), Einfluß der Endbenutzer-Anforderungen (vgl. 2.5.4), Begrenzung der Anforderungen
- *Konstruktion*: Zerlegung in Komponenten, Entkopplung der Komponenten über Schnittstellen, Variabilitätsmechanismus, Menge an AT-Software
- *Programmierung*: Verwendete Programmiersprache (und Klassifikation als objektorientiert oder prozedural), Einsatz von Generierung, Durchführung eines Code Reviews, Trennung von Außen- und Innensicht bei der Dokumentation, Bereitstellung einer Beispielanwendung

Eine Zusammenfassung des Vergleichs ist in Tabelle 5-2 dargestellt (fett umrandet sind die Kriterien, die die erfolgreichen Projekte unterscheiden).

Kriterium		QDI	DWL	Client Framework
Allgemeine Projektmerkmale/Spezifikation	Erfolgreiche Nutzungen	5	2	**keine**
	Zeitraum der Nutzungen [Jahre]	1	4	–
	Ursprüngliche Entwicklungskosten K_e [Bearbeitermonate]	3	18	60
	Explizite Investition in Wiederverwendbarkeit I_{wv} [Bearbeitermonate]	3,5 (ohne Schulungen)	2	0
	Wiederverwendbarmachung	A priori	A posteriori	A priori
	Entwicklungsprozeß und Anzahl produktiver Versionen in dessen Verlauf	Mehrere Iterationen; produktive Versionen am Ende jeder Iteration	Zwei hauptsächliche Iterationen mit jeweils produktiver Version	Eine produktive Version am Ende des Prozesses
	Einfluß der Endbenutzer-Anforderungen	Gering	Gering	Hoch
	Anforderungen von vornherein begrenzt?	Ja	Ja	Nein
Konstruktion	Architektur: Zerlegung in Komponenten?	Ja	Ja	Ja
	Entkopplung der Komponenten über Schnittstellen?	Ja	Ja	Nein
	Hauptsächlicher Variabilitätsmechanismus	Schnittstellen	Schnittstellen	Vererbung
	Menge an AT-Software	Keine	Keine	Hoch
Programmierung	Verwendete Programmiersprache	Java (OO)	ANSI-C (prozedural)	Java (OO)
	Generierung eingesetzt?	Ja	Ja	Nein
	Code Review durchgeführt?	Ja	Ja	Nein
	Dokumentation: Außen- und Innensicht getrennt?	Ja	Ja	Nein
	Beispielanwendung bereitgestellt?	Ja	Ja	Nein

Tabelle 5-2: Vergleichende Zusammenfassung wesentlicher Ergebnisse

5.3.2 Schlußfolgerungen für die empirisch zu überprüfenden Hypothesen

Aus den Ergebnissen ziehen wir einige Schlußfolgerungen für die in 3.5.3 formulierten Hypothesen. Da wir gezielt Projekte betrachtet haben, in denen wiederverwendbare Software entwickelt wurde, können wir nicht generell Aussagen zu den allgemeinen Rahmenbedingungen ableiten, zumal eines der Projekte nicht erfolgreich war. Wir beschränken uns daher auf die Hypothesen, zu denen sich eine eindeutige Aussage ergibt; eine Übersicht des aktualisierten Status zeigt Tabelle 5-3.

Ein wesentliches Ergebnis der in diesem Kapitel behandelten Fallstudien ist die große Bedeutung der Architektur für den Erfolg. Das bestätigt die bisher offene Hypothese [Archi].

Zusätzliche Indizien, die das Ergebnis aus Kapitel 4 stützen, ergeben sich für vier Hypothesen. Eindeutig ergibt sich, daß eine objektorientierte Vorgehensweise keine Voraussetzung für den Erfolg der Wiederverwendung darstellt. Dies bestätigt zusätzlich Hypothese [OO]. In keinem der untersuchten Projekte gab es ein Problem wegen negativer Einstellung der Entwickler zur Wiederverwendung; dies ist ein zusätzliches Indiz für die Widerlegung von Hypothese [NegEinst]. Da insbesondere keine Rede vom Not-invented-here-Syndrom sein kann, ist dies auch ein zusätzliches Indiz für die Widerlegung der Hypothese [NIH].

Nr	Hypothese	Ergebnis	Änderung	Kapitel
5	[NegEinst] Die Entwickler sind grundsätzlich negativ zur Wiederverwendung eingestellt.	Widerlegt.	Zusätzliches Indiz	4.2.2, 5.2
10	[GrundsProb] Wiederverwendung ist grundsätzlich möglich; es gibt keine inhärenten, grundsätzlich nicht lösbaren Probleme (z.B. schlechtere Performance, suboptimale Lösungen).	Bestätigt.	Zusätzliches Indiz	4.2.2, 4.2.3, 5.2
13	[NIH] Das Not-invented-here-Syndrom spielt eine große Rolle.	Widerlegt.	Zusätzliches Indiz	4.2.3, 5.2
17	[Archi] Die Architektur hat großen Einfluß auf den Erfolg der Wiederverwendung.	Bestätigt.	Neu	5.2
18	[OO] Objektorientierte Entwicklung ist keine Voraussetzung für den Erfolg.	Bestätigt.	Zusätzliches Indiz	4.2.3, 5.2

Tabelle 5-3: Aktualisierter Status betroffener Hypothesen

5.3.3 Softwaretechnische Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software

Aus der vergleichenden Analyse leiten wir verallgemeinernd drei übergeordnete Erfolgsfaktoren ab:

1. Durchführbarkeit sicherstellen
2. Variabilität richtig realisieren
3. In Nutzbarkeit für Entwickler investieren

Ad 1. Softwareentwicklungsprojekte können daran scheitern, daß die Aufgabe zu schwierig ist. Sie wird zu wesentlichen Teilen festgelegt bei der Spezifikation der Anforderungen an die zu entwickelnde Software. Da erhöhte Wiederverwendbarkeit von Software in der Regel durch eine Steigerung von Anzahl und Komplexität der unterschiedlichen abgedeckten Anforderungen erreicht wird (vgl. 2.1.2), ist die Gefahr besonders groß, bei ihrer Spezifikation das Maß des Machbaren zu überschreiten.

Eine Reihe von Maßnahmen hilft, das zu vermeiden. Zunächst steigt die Gefahr mit der Projektgröße; Projekte zur Entwicklung wiederverwendbarer Software sollten daher so klein wie möglich gehalten werden. Es gibt zwei weitere wirksame Möglichkeiten zur Begrenzung der Anforderungen. (i) Die ursprüngliche Ausrichtung auf einen konkreten Anwendungsfall bei a posteriori wiederverwendbarer Software erleichtert die Begrenzung der Anforderungen. (ii) Anforderungen bezüglich der Anwendungsfuktionalität sind besonders schwer zu standardisieren und zu begrenzen (vgl. 2.5.4). Man sollte darum Software umso bevorzugter zur Wiederverwendbarmachung auswählen, je geringer sie dadurch bestimmt ist.

Was den Entwicklungsprozeß angeht, so ist ein frühzeitiger iterativer Rückfluß von Erfahrungen der Nutzer hilfreich bei der Spezifikation. Dazu sollten früh produktive Versionen erstellt und einem Beta-Test unterzogen werden. Bei der Wiederverwendbarmachung a posteriori ist dies zu einem gewissen Grad immer gegeben, da zunächst eine produktive Version in Form nicht systematisch wiederverwendbarer Software vorliegt, die in einem zweiten Schritt wiederverwendbar gemacht wird. Im Falle der Wiederverwendbarmachung a priori muß dieser Prozeß explizit eingeführt werden.

Schließlich ist aus Sicht des Managements der Zwang zur realistischen Bestimmung des für die Wiederverwendbarmachung notwendigen Budgets ein geeignetes Mittel zur Disziplinierung bei der Spezifikation. Dieses Budget muß dann auch zur Verfügung gestellt werden, um sicherzustellen, daß die geplanten Maßnahmen realisiert werden können.

Ad 2. Variabilität ist ein wesentliches Merkmal wiederverwendbarer Software. Wichtig ist, daß Variabilität auf geeignete Weise und in ausreichendem Umfang realisiert wird. Entscheidend ist dafür die Einhaltung des Geheimnisprinzips, das die Grundlage für die Änderbarkeit von Software generell darstellt. Dazu dienen zwei Maßnahmen auf Ebene der Architektur: die Zerlegung des Systems in durch Schnittstellen entkoppelte Komponenten und die Trennung der Zuständigkeiten in Form der Software-Kategorien. Objektorientierte Pro-

grammierung ist dafür keine Voraussetzung, kann aber hilfreich sein, wenn sie richtig eingesetzt wird.

Als hauptsächlicher Variabilitätsmechanismus sollte der Austausch von Implementierungen eingesetzt werden. Wird objektorientiert programmiert, so konkurriert damit die Spezialisierung durch Implementierungsvererbung. Sie hat jedoch den großen Nachteil, daß sie das Geheimnisprinzip verletzt [BrS02] und sollte daher vermieden werden. Im speziellen Fall der Software für Formattransformationen sollte Generierung als Variabilitätsmechanismus eingesetzt werden. Voraussetzung dafür ist, daß sie als stereotype R-Software implementiert wird und daß die entsprechenden Metainformationen zur Verfügung stehen.

Ad 3. Alle bisher erwähnten Maßnahmen sind zwar Voraussetzungen für erfolgreiche Nutzung, reichen aber für sich genommen nicht aus. In die Nutzbarkeit für Entwickler muß investiert werden. Diese Investition dient folgenden Zielen: der Verständlichkeit, der Änderungseffizienz und der Verfügbarkeit.

Im Sinne der Verständlichkeit ist eine Dokumentation zu erstellen, die zwischen Außen- und Innensicht trennt. Dafür muß ausreichend Zeit zur Verfügung stehen, die im Sinne der Qualität (d.h. Strukturiertheit) und nicht der Quantität der Dokumentation einzusetzen ist. Durch Code Reviews kann Entwicklern das Verständnis des Codes erleichtert werden werden, indem er vereinheitlicht und allgemein verständlich dokumentiert wird; erfolgt die Wiederverwendbarmachung a posteriori, ist in diesem Zusammenhang bei Variablennamen und Kommentaren vom ursprünglichen Anwendungsfall zu abstrahieren. Eine Beispielimplementierung dient ebenfalls der Verständlichkeit, weil sie intuitives Verständnis ermöglicht. Nicht zuletzt sind Investitionen in Schulungen und Support-Leistungen Mittel, das Verständnis der Nutzer zu erhöhen.

Eine wesentliche Investition in die Änderungseffizienz stellen Testfälle für Regressionstests dar, weil sie ermöglichen, schnell das korrekte Funktionieren nach dem Durchführen einer Änderung sicherzustellen. Ein häufig auftretender Nebeneffekt ist, daß Testfälle auch die Verständlichkeit erhöhen. Darüber hinaus verringern Generatoren den Aufwand bei der Anpassung, weil zu ändernder Code nicht von Hand geschrieben werden muß.

Der Verfügbarkeit dienen alle Maßnahmen, die das Auffinden der Software erleichtern: Publikationen über wiederverwendbare Komponenten, Einstellen in eine Bibliothek, Schulungen.

5.4 Zusammenfassung

Wir haben drei Projekte zur Entwicklung wiederverwendbarer Software in Form von Fallstudien untersucht. Schwerpunkt war dabei die Architektur, die im Detail betrachtet wurde. Aus der vergleichenden Analyse wesentlicher Charakteristika dieser Projekte haben wir Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software abgeleitet. Drei Aspekte sind dabei wichtig: die Durchführbarkeit der Aufgabe an sich, die Architektur und hier besonders die Variabilität, und schließlich Maßnahmen, welche die Nutzung erleichtern.

Kapitel 6

Ökonomische Bewertung wiederverwendbarer Software: das ReValue-Modell

Ziel dieses Kapitels ist, eine Methode zu entwickeln, mit der das ökonomische Potential der Entwicklung wiederverwendbarer Software in Abhängigkeit von den jeweiligen Rahmenbedingungen bestimmt werden kann. Dabei sollen alle wesentlichen Faktoren berücksichtigt werden, um zu möglichst realistischen Aussagen zu gelangen.

Hierzu stellt das Kapitel zunächst das ReValue-Modell für die ökonomische Bewertung von Projekten zur Entwicklung wiederverwendbarer Software vor. Das Modell hat drei Bestandteile: Es stellt eine Bewertungsmethode bereit, die auf praktikable Weise die Investitionsunsicherheit berücksichtigt. Zusätzlich werden Herstellung und Nutzung in umfassender Weise ökonomisch modelliert. Schließlich umfaßt es einen Algorithmus, der die Nutzung mit Hilfe der Monte-Carlo-Methode simuliert. Auf dieser Grundlage präsentieren wir Simulationsergebnisse für verschiedene exemplarische Szenarien, analysieren sie und ziehen Schlußfolgerungen für das ökonomische Potential in Abhängigkeit von den Rahmenbedingungen.

Inhalt: **Seite**

6.1 Einleitung

In der Einleitung erläutern wir zunächst, wodurch wir dazu motiviert worden sind, das ReValue-Modell zu entwickeln. Wir beschreiben dann softwareökonomische und softwaretechnische Grundlagen des Modells. Anschließend erläutern wir, wie Monte-Carlo-Simulationen als Mittel ökonomischer Analyse eingesetzt werden. Abschließend fassen wir den Stand der Forschung zusammen und stellen unseren wissenschaftlichen Beitrag dar.

6.1.1 Motivation für die Entwicklung des Modells

Das am häufigsten angeführte Argument für die Entwicklung wiederverwendbarer Software ist ökonomischer Art: es wird eine Senkung der Kosten in der Softwareentwicklung und – damit verbunden – eine bedeutende Produktivitätssteigerung erwartet (vgl. [Cox90]; [Gol95], 204; [Den91], 17; [Kar95], 8, 11; [Tra95], 132f; [GrW95]; [JGJ97], 4ff; [McC97], 3; [SoS99], 5; siehe auch 3.5.3), für die Faktoren von bis zu zehn angegeben werden (vgl. [Tra95], 132f; [Gol95], 204).

Erste Zweifel daran äußerten Margono und Rhoads bereits 1992 [MaR92]. Erschwerend kommt hinzu, daß die empirische Basis für den ökonomischen Nutzen der Wiederverwendung äußerst schwach ist: In der Regel werden einige wenige Werte kolportiert, ohne Details oder – in vielen Fällen – auch nur eine Quelle anzugeben.[70] In den wenigen Fällen, in denen detaillierte Daten erhoben werden [Lim94, MSG96], sind Bewertungsmethode und Analyse der Rahmenbedingungen mangelhaft (vgl. 3.3).

Aus diesem Grund besteht dringender Bedarf, Daten zu erheben (vgl. 3.5.1). Bei dem Versuch, emipirische Daten aus der industriellen Praxis zu gewinnen, machten wir allerdings die Erfahrung, daß ausreichendes Zahlenmaterial nicht verfügbar ist. Daher greifen wir auf das Mittel der Simulation mit Hilfe der Monte-Carlo-Methode zurück und ermitteln Werte auf Basis eines detaillierten ökonomischen Modells von Herstellung und Nutzung wiederverwendbarer Software. Dabei machen wir Annahmen und Simulationsalgorithmus transparent, um eine Überprüfung und Weiterentwicklung zu ermöglichen. Der Name des Modells – *ReValue* – ist ein Akronym für 'Reusable Software Valuation Model'. Gleichzeitig steht ReValue[71] im wörtlichen Sinne für Neubewertung,

---

[70] Vgl. z.B. [JGJ97]: "Organizations producing business systems have improved by 10% a year; engineering systems, by 8%; real-time systems, by 6% (Putnam and Myers, 1996). The best record – 16% per year, sustained over 15 years – was made by the business-software division of a large telecommunications company" ([JGJ97], 5). In [McC97] findet sich folgende Aussage: "For example, during its first object-oriented development project, the Virginia-based system integrator company did not enjoy any real benefits from reuse. [...] In the second project, becuase 70 percent of the system was built from reusable components, the project was completed six months ahead of schedule" ([McC97], 6).

[71] revalue: engl. für "neu bewerten" [LEO02]

da wir glauben, daß eine differenzierte Neubewertung der ökonomischen Aspekte der Entwicklung wiederverwendbarer Software notwendig ist.

6.1.2 Grundlagen des Modells

Softwareökonomische Grundlagen

Das ReValue-Modell baut auf den in 2.4 dargelegten Grundlagen der Softwareökonomie auf. Die wichtigsten davon sind die Folgenden:

- Die Ausgaben für die Wiederverwendbarmachung von Software werden als Investition betrachtet.
- Die Entscheidung über die Wiederverwendbarmachung von Software ist eine Investitionsentscheidung unter Unsicherheit. Maß für die Unsicherheit ist die statistische Streuung der erwartetem Rendite.
- Die Investition wird mit Hilfe der Kapitalwertmethode bewertet.
- Die Rechnungslegung erfolgt quartalsweise. Dies ist bei in den USA börsennotierten Unternehmen vorgeschrieben und wird auch von anderen Unternehmen zunehmend übernommen.

Softwaretechnische Grundlagen

Von den in Kapitel 2 gelegten softwaretechnischen Grundlagen sind für das ReValue-Modell folgende von besonderer Bedeutung:

- Wir konzentrieren uns auf Komponenten (vgl. 2.1.3, 2.3.3).
- Die Komponenten veralten, wobei typische Innovationszyklen im Durchschnitt bei 18 bis 24 Monaten liegen (vgl. 2.2.2).
- Bei der Nutzung liegt der Schwerpunkt auf White-Box-Wiederverwendung: Die Komponenten werden in in angepaßter Form eingesetzt, wobei die Anpassung Modifikationen des Quellcodes einschließt (vgl. 2.1.4).

6.1.3 Monte-Carlo-Simulation als Mittel ökonomischer Analyse

Die Monte-Carlo-Methode dient dazu, Prozesse statistisch zu untersuchen, indem sie mit Hilfe eines Rechners simuliert werden. Dazu werden die Wahrscheinlichkeitsverteilungen der den Prozeß beschreibenden Variablen ermittelt und durch Parameter beschrieben. Diese Parameter sind die Eingangsgrößen der Simulation. In einer großen Anzahl von Simulationsläufen werden auf Basis der Wahrscheinlichkeitsverteilungen zufällig Werte für die Variablen des Systems bestimmt und damit die Ausgangsgrößen für den jeweiligen Simulationslauf berechnet. Anschließend wird die Verteilung der Ausgangsgrößen statistisch ausgewertet. Die zufällige Bestimmung der Variablenwerte erfolgt unter Nutzung vom Rechner generierter Pseudo-Zufallszahlsequenzen (vgl. [CEP96], 1f).

Simulationen mit der Monte-Carlo-Methode werden insbesondere dann eingesetzt, wenn die Bestimmung der Ausgangsgrößen nicht in einer geschlossenen Lösung möglich ist. Als Methode der Projektanalyse dienen sie außerdem

dazu, den gesamten Parameterraum zu untersuchen. Monte-Carlo-Simulationen sind somit nicht auf eine begrenzte Anzahl von Kombinationen der Modellvariablen beschränkt (vgl. [Her68]; [BrM95], 223ff); zudem lassen sich die Abhängigkeiten der Parameter abbilden [ChN95].

Wir wählen die Monte-Carlo-Methode, um die Investition in wiederverwendbare Software ökonomisch zu analysieren. Folgende Gründe sind dafür ausschlaggebend:

- Um die Unsicherheit der Investition in angemessener Weise zu berücksichtigen, muß die statistische Streuung der Meßgröße ermittelt werden. Monte-Carlo-Simulationen lösen dieses Problem, da sie es ermöglichen, die Wahrscheinlichkeitsverteilungen der Ausgangsgrößen zu bestimmen.
- Die ökonomische Modellierung der Entwicklung wiederverwendbarer Software muß die Wartung einbeziehen und Abhängigkeiten zwischen verschiedenen Variablen abbilden. Dies führt dazu, daß die Ermittlung des Kapitalwerts geschlossen nicht mehr möglich ist, weil sowohl Einnahmen als auch Kosten zeitlich verteilt auftreten (vgl. 2.4.3). Bei Einsatz der Monte-Carlo-Methode stellt das kein Problem dar.
- Angesichts des unzureichenden Datenmaterials ist es notwendig, den Parameterraum so weiträumig wie möglich zu erforschen. Monte-Carlo-Simulationen sind dafür ein ideales Mittel.

6.1.4 Stand der Forschung und eigener Beitrag

Eine Reihe von Veröffentlichungen, die wir in 3.2 und 3.4.2 eingehend untersuchen, behandelt das Problem der ökonomischen Bewertung wiederverwendbarer Software [JGJ97, Kar95, McC97, PoC93, Lim94, Wit96, FFF97, WiB98, Sci99, MSR99, MFG00]. Zusätzlich bauen wir auf den Ergebnissen dreier Übersichten [BSa97, Lim98, Wil99] über ältere Veröffentlichungen zum Thema auf. Wir fassen im Folgenden den Forschungsbedarf noch einmal kurz zusammen (siehe auch 3.5.1).

Eine Lücke besteht einerseits im Hinblick auf die Behandlung der Unsicherheit der Investition: einige Modelle berücksichtigen sie nicht, andere Modelle bauen auf Konzepten auf, die im fraglichen Kontext nicht praktikabel sind. Andererseits sind die zugrundeliegenden ökonomischen Modelle der Entwicklung teilweise unzureichend, weil die Wartung nicht modelliert wird.

Unser wissenschaftlicher Beitrag besteht in der Entwicklung einer Bewertungsmethode, die auf der statistischen Verteilung des internen Zinssatzes basiert und die Investitionsunsicherheit berücksichtigt, ohne auf schwer verfügbare Parameter zurückgreifen zu müssen. Daneben entwerfen wir ein detailliertes ökonomisches Modell von Herstellung und Nutzung wiederverwendbarer Software, das die Wartung einbezieht.

Dieses Modell ist Grundlage für die Simulation der Nutzung wiederverwendbarer Software mit Hilfe der Monte-Carlo-Methode. Der beschriebene Simulationsalgorithmus ermöglicht hierbei, für beliebige Parameterkonstellationen des beschriebenen ökonomischen Modells die statistische Verteilung des inter-

nen Zinssatzes zu ermitteln und so verschiedene wiederverwendbare Komponenten zu bewerten. Daneben beinhaltet er eine Erweiterung der Definition des internen Zinssatzes für den Fall, daß der Kapitalwert für alle Zinssätze kleiner Null ist.

Angesichts des Mangels an verfügbaren empirischen Daten (vgl. 3.5) leisten wir insgesamt einen Beitrag zur Ermittlung des ökonomischen Potentials wiederverwendbarer Software, insbesondere, indem wir Simulationsergebnisse für einige exemplarische Szenarien präsentieren und analysieren.

6.2 Das ReValue-Modell

In diesem Abschnitt präsentieren wir das ReValue-Modell. Zunächst stellen wir es im Überblick dar und gehen anschließend auf die drei Bestandteile ein: die Bewertungsmethode, die ökonomische Modellierung von Herstellung und Nutzung wiederverwendbarer Software und den Algorithmus der Monte-Carlo-Simulation.

6.2.1 Das Modell im Überblick

Das ReValue-Modell dient dazu, Investitionen in die Entwicklung wiederverwendbarer Software ökonomisch zu bewerten. Es besteht aus

- einer Bewertungsmethode für Investitionen in die Entwicklung wiederverwendbarer Software, die auf praktikable Weise die Unsicherheit der Investition berücksichtigt,
- einer ökonomischen Modellierung von Herstellung und Nutzung wiederverwendbarer Software, d.h. der Bestimmung von Parametern und Berechnungsvorschriften, die es erlauben, alle relevanten Geldflüsse abzubilden,
- einem Algorithmus für die Ermittlung des internen Zinssatzes, der auf dem ökonomischen Modell beruht und die Nutzung mit Hilfe der Monte-Carlo-Methode simuliert.

Das Modell erlaubt, die wesentlichen Parameter frei zu wählen und so beliebige Konstellationen zu analysieren. Der in der Programmiersprache Matlab erstellte Code ist ein Ergebnis dieser Arbeit.

Einige Ergebnisse für exemplarische Szenarien stellen wir in 6.3 vor. Damit ist das ReValue-Modell jedoch keinesfalls erschöpft, vielmehr kann es allgemein als Hilfsmittel für die Entscheidung über Investitionen in wiederverwendbare Software eingesetzt werden.

6.2.2 Bewertungsmethode

Die in der Literatur vorgeschlagenen Bewertungsmethoden fallen in zwei Gruppen, die beide jeweils Mängel im Hinblick auf die Behandlung der Investitionsunsicherheit haben. Die eine Gruppe berücksichtigt sie nicht oder nicht

ausreichend. Die andere Gruppe dagegen stellt sie in Rechnung, tut dies allerdings auf eine Weise, die die Praktikabilität der Methode in Frage stellt, weil die notwendigen Parameter nicht verfügbar sind (vgl. 2.4.4, 3.5.1, 6.1.4).

Unser Verfahren beseitigt diese Schwächen, indem es auf Basis des internen Zinssatzes eine intuitiv verständliche Zielgröße bestimmt, die die Information über die Unsicherheit enthält und dabei ausschließlich auf verfügbare Größen zurückgreift.

Zielgröße

Die Zielgröße unseres Verfahrens ist der interne Zinssatz, der mit Hilfe der Kapitalwertmethode bestimmt wird und der folgende zwei Vorteile hat (vgl. 2.4.4):

- Er ist intuitiv besser verständlich als der Kapitalwert (vgl. [HaV99], 221f; [Bra01]) und kann daher auch von Softwareentwicklern angewendet werden.
- Er ermöglicht den Vergleich zwischen verschiedenen alternativen Projekten, bei unterschiedlicher Investitionshöhe.

Behandlung der Unsicherheit

Als Meßgröße für die Unsicherheit der Investition verwenden wir die statistische Streuung der erwarteten Werte für den internen Zinssatz (vgl. [Her68]; [BrM95], 131ff). Sie steht als Ergebnis der Monte-Carlo-Simulation zur Verfügung.

Eine Möglichkeit, die Unsicherheit zu bewerten, ist die statistische Bestimmung einer unteren Grenze, die der interne Zinssatz mit hoher Wahrscheinlichkeit nicht unterschreitet. Ein entsprechendes Verfahren wird von Banken als Standard bei der Bestimmung des mit einer Geldanlage verbundenen Risikos eingesetzt: Zur Berechnung des sog. 'value at risk' wird die Untergrenze eines einseitigen Konfidenzintervalls für den prognostizierten Wert ermittelt, wobei ein Konfidenzniveau von 99 Prozent vorgeschrieben ist [Bun00].

Dieses Verfahren übertragen wir auf das vorliegende Problem: die Verteilung der Werte für den internen Zinssatz liegt vor, so daß die untere Grenze eines einseitigen Konfidenzintervalls einfach zu bestimmen ist. Das Konfidenzniveau kann je nach Ausrichtung des Unternehmens frei gewählt werden und sollte umso höher liegen, je größer das Sicherheitsbedürfnis ist. Wir gehen davon aus, daß in der Softwareentwicklung etwas höhere Risiken als im Bankgeschäft akzeptabel sind und setzen daher das Konfidenzniveau mit 95 Prozent an, zumal dieser Wert oft auch in anderem Kontext üblich ist. Das vorgeschlagene Verfahren ist jedoch mit jedem beliebigen anderen Wert durchführbar.

Wir fassen zusammen: Der Ausgabewert unserer Bewertungsmethode ist die interne Rendite, die von der Investition mit der dem Konfidenzniveau entsprechenden Wahrscheinlichkeit mindestens erzielt werden kann. Dadurch wird der Vergleich von Investitionen mit unterschiedlichen Erwartungswerten und Varianzen anhand eines einzigen intuitiv verständlichen Wertes möglich.

6.2.3 Ökonomische Modellierung von Herstellung und Nutzung

Basis der Monte-Carlo-Simulation ist ein geeignetes ökonomisches Modell der Herstellung und Nutzung wiederverwendbarer Software. Viele herkömmliche Verfahren gehen zu grob vor und vernachlässigen beispielsweise die Wartung (vgl. 6.1.4).

Im Folgenden beschreiben wir, wie die entsprechenden Geldflüsse modelliert werden. Da wir die Kapitalwertmethode zugrundelegen, werden sämtliche zukünftigen Geldflüsse auf die Gegenwart abgezinst (vgl. 2.4.4).

Geldflüsse der Herstellung

Bezugsgröße bei der Modellierung der Herstellungskosten einer wiederverwendbaren Komponente sind die Kosten für die Entwicklung derselben Komponente ohne Investition in Wiederverwendbarkeit. Bei a posteriori wiederverwendbarer Software entspricht dies den Kosten für die ursprüngliche Entwicklung, bei a priori wiederverwendbarer Software sind diese Kosten fiktiv und müssen geschätzt werden. Wir nennen sie in beiden Fällen *ursprüngliche Entwicklungskosten* und bezeichnen sie mit K_e.

Für die Wiederverwendbarmachung der Komponente fallen einmalig zusätzliche Kosten an, die die *Investition in Wiederverwendbarkeit* I_{wv} darstellen. Wir messen diese Kosten relativ zu den ursprünglichen Entwicklungskosten und modellieren sie durch den Faktor i_{wv}.

Daneben entstehen regelmäßig Zusatzkosten für die Wartung der wiederverwendbaren Komponente. Wir gehen davon aus, daß diese Wartungskosten mit der Anzahl der Nutzungen n ansteigt und beschreiben sie im Modell als Funktion von n, die sich asymptotisch einem frei wählbaren Endwert w annähert. Diesen Endwert modellieren wir als Faktor relativ zur Investition in Wiederverwendbarkeit. Als Funktion für den Faktor wählen wir eine Funktion der Gestalt

$$v(n) = w\left(1 - e^{-\frac{n}{\alpha}}\right),$$

wobei $v(n)$ der effektive Faktor ist und α so bestimmt wird, daß bei einer vorgegebenen Anzahl n_{90} von Nutzungsereignissen 90 Prozent des asympotischen Endwerts w erreicht werden.

Außerdem nehmen wir an, daß der Wartungszeitraum begrenzt ist. Wir modellieren dies dadurch, daß wir die maximale Wartungsdauer, gemessen ab dem letzten Nutzungsereignis, als Parameter T_w vorsehen. Wo nicht anders erwähnt, setzen wir sie mit zehn Jahren an.

Die Herstellungskosten umfassen also sowohl die einmalig anfallenden Kosten für die Wiederverwendbarmachung – auch als Investition in die Wiederverwendbarkeit bezeichnet – als auch die regelmäßig anfallenden zusätzlichen Wartungskosten, die abgezinst werden müssen.

Geldflüsse der Nutzung

Wenn die Kosten für die Nutzung einer wiederverwendbaren Komponente insgesamt niedriger sind als die Kosten für die jeweilige Neuentwicklung einer entsprechenden Komponente, werden durch Wiederverwendung Kosten eingespart. Entscheidend ist, daß diese Kostenersparnis in das Modell eingeht und nicht vernachlässigt wird. Sie umfassen das Suchen nach der Komponente, die Einarbeitung und gegebenenfalls die Anpassung an den jeweiligen Kontext. Alle diese Kosten, die einmalig anfallen, fassen wir als *Nutzungskosten* K_n zusammen und modellieren sie durch den Nutzungskostenfaktor κ, der sie als Anteil der ursprünglichen Entwicklungskosten angibt.

Wir nehmen an, daß die Kosten für die Neuentwicklung den ursprünglichen Entwicklungskosten entsprechen. Die Kosteneinsparung durch die Nutzung einer wiederverwendbaren Komponente berechnet sich also als Differenz zwischen den ursprünglichen Entwicklungskosten und den Nutzungskosten. Diese Kosteneinsparung ΔK entspricht einer Einnahme (vgl. 2.4.2), die abgezinst werden muß.

Zudem gehen wir davon aus, daß neben den Wartungskosten für den Hersteller auch für den Nutzer regelmäßig spezifische Wartungskosten anfallen, weil der Nutzer die Wartung durch den Hersteller verfolgen und gegebenenfalls seine eigenen Anpassungen warten muß. Wir nehmen an, daß der Anteil dieser spezifischen Wartungskosten an den Wartungskosten des Herstellers dem Anteil der Anpassungskosten an den ursprünglichen Entwicklungskosten entspricht.

Den spezifischen Wartungskosten stehen die Wartungskosten in voller Höhe gegenüber, die anfallen würden, wenn eine Neuentwicklung genutzt würde. Somit entstehen bei der Wartung periodisch Kosteneinsparungen in Höhe der Differenz zwischen den vollen und den spezifischen Wartungskosten. Auch sie entsprechen Einnahmen, die abgezinst werden müssen.

Modellierung des Nutzungsprozesses

Die entscheidende Kenngröße für die Beschreibung des Nutzungsprozesses ist die Anzahl der Nutzungen pro Zeiteinheit, in der Regel pro Jahr (vgl. [Kar95], 177). Wir nehmen an, daß die Zeitpunkte verschiedener Nutzungsereignisse statistisch gesehen nicht voneinander abhängen, da der Bedarf für bestimmte Komponenten innerhalb eines Projekts grundsätzlich nicht von dem in anderen Projekten abhängt.

Ein solcher Zählvorgang wird statistisch üblicherweise durch eine Poisson-Verteilung modelliert, deren Parameter λ die Anzahl der Ereignisse pro Zeiteinheit angibt ([FKP99], 258; [Sob94], 35f). Wir nennen λ die *Nutzungsrate* einer wiederverwendbaren Komponente. Typische andere Vorgänge, die sich durch einen Poisson-Prozeß beschreiben lassen, sind telefonische Verbindungsanfragen bei einer Vermittlungseinrichtung, Eintreffen von Kunden in einem Einkaufszentrum oder Auto-Durchfahrten an einem Punkt des Straßennetzes ([Koh72], 84). Gehorchen die Nutzungszahlen einer Poissonverteilung, so sind die Zeitintervalle zwischen jeweils zwei Nutzungsereignissen exponentialverteilt mit Parameter λ ([FKP99], 277f).

Daneben muß bei der Modellierung der Nutzung beachtet werden, daß wiederverwendbare Komponenten veralten (vgl. 6.1.2). Erfahrungen aus der industriellen Praxis zeigen, daß deshalb ihre Wiederverwendbarkeit im Laufe der Zeit abnimmt ([Kar95], 22). Dies läßt sich wie ein Zerfallsprozeß in der Physik beschreiben, der eine bestimmte Halbwertszeit hat, nach der die Hälfte der Teilchen zerfallen sind (vgl. [CuS95], 279). Wir übertragen dies, indem wir die Nutzungsrate λ so modellieren, daß sie, ausgehend vom Startwert λ_0 im Laufe der Zeit mit einer frei wählbaren Halbwertszeit T_h abnimmt (vgl. [HMS92], 577):

$$\lambda(t) = \lambda_0 e^{-\frac{\ln 2}{T_h} t}$$

Ein solcher Poisson-Prozeß, dessen Parameter λ eine Funktion der Zeit ist, heißt *inhomogener Poisson-Prozeß* [Vir01].

A priori und a posteriori wiederverwendbare Software

Das Modell kann zur Untersuchung der Entwicklung von a priori und a posteriori wiederverwendbarer Software gleichermaßen eingesetzt werden, da nur die Kosten der Wiederverwendbarmachung, nicht aber die ursprünglichen Entwicklungskosten in die Berechnung eingehen. Dadurch ist offensichtlich der A-priori-Fall abgedeckt. Im Fall a posteriori wiederverwendbarer Software steht den ursprünglichen Entwicklungskosten ein sicheres Nutzungsereignis zum Zeitpunkt $t = 0$ gegenüber. Dies gleicht sich bis auf einen etwaigen Gewinn oder Verlust aus diesem Nutzungsereignis aus, den wir der ursprünglichen Entwicklung zurechnen.

6.2.4 Algorithmus der Monte-Carlo-Simulation

Der Simulationsalgorithmus baut auf der in 6.2.3 beschriebenen ökonomischen Modellierung von Herstellung und Nutzung wiederverwendbarer Komponenten auf. Er wurde in der Programmiersprache Matlab implementiert. Der Algorithmus hat zwei wesentliche Elemente, die wir im Folgenden detailliert behandeln:

- Simulation des Nutzungsprozesses
- Numerische Nullstellenberechnung zur Bestimmung des internen Zinssatzes

Bevor wir dazu kommen, gehen wir jedoch kurz auf Ein- und Ausgangsgrößen der Simulation ein und geben einen Überblick über die Struktur des Simulationsprogramms.

Ein- und Ausgangsgrößen der Simulation

Eingangsgrößen der Simulation sind die in 6.2.3 bereits erwähnten Modellparameter, die frei gewählt werden können. Wir fassen sie hier noch einmal zusammen (vgl. auch 2.1.5):

- *Parameter der Herstellung.* Ursprüngliche Entwicklungskosten K_e, Faktor der Investition in Wiederverwendbarkeit i_{wv} (Investition in Wiederverwendbarkeit $I_{wv} = i_{wv}\ K_e$), asymptotischer Endwert des Wartungskostenfakors w (relativ zu I_{wv}), 90-Prozent-Grenze der Nutzungsanzahl n_{90}[72], maximale Wartungsdauer T_w
- *Parameter des Nutzungsprozesses.* Anfängliche Nutzungsrate λ_0, Halbwertszeit der Nutzungsrate T_h
- *Parameter der Nutzung.* Nutzungskostenfaktor κ (Nutzungskosten $K_n = \kappa K_e$), Kostenersparnis pro Nutzung $\Delta K = K_e - K_n = (1-\kappa)K_e$

Ausgangsgröße der Simulation ist die statistische Verteilung des internen Zinssatzes *IRR*[73]. Sie wird beschrieben durch den Erwartungswert des internen Zinssatzes E(*IRR*), seine Standardabweichung σ_{IRR} und die Untergrenzen der Konfidenzintervalle für Konfidenzniveaus von 95 und 99 Prozent, IRR_{95} und IRR_{99}.

Struktur des Simulationsprogramms

Das Simulationsprogramm besteht aus folgenden drei großen Blöcken (vgl. Abbildung 6-1):

- Initialisierung der Parameter
- Iterationsschleife
- Statistische Auswertung der Simulationsdaten und Ausgabe der Ergebnisse.

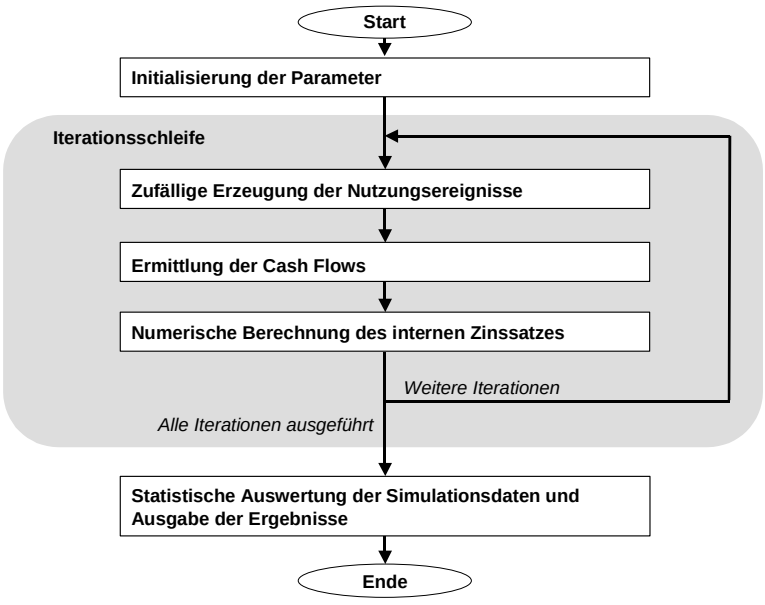

Abbildung 6-1: Struktur des Simulationsprogramms

[72] Wo nicht anders angemerkt, wird für n_{90} ein Wert von drei angesetzt.

[73] IRR ist die Abkürzung für *Internal Rate of Return* (vgl. 2.4.4).

Bei jedem Durchlauf der Iterationsschleife werden von neuem per Zufall Nutzungsereignisse erzeugt. Für den jeweiligen Nutzungsfall, d.h. die Anzahl und den zeitlichen Verlauf der Nutzungsereignisse, werden dann die Cash Flows für Kosten und Ersparnisse ermittelt. Daraus wird mit Hilfe einer numerischen Nullstellenberechnung der interne Zinssatz bestimmt. Jede Ausführung der Iterationsschleife entspricht dabei einem Simulationslauf der Monte-Carlo-Simulation.

Die wesentlichen Ergebnisse jedes Simulationslaufs werden gesammelt. Neben dem internen Zinssatz sind das die zeitlichen Verläufe der verschiedenen Cash Flows für Kosten und Einnahmen. Nach Ende der Iterationen werden die Simulationsdaten für den internen Zinssatz statistisch ausgewertet, indem Mittelwert und Standardabweichung sowie die Untergrenze eines einseitigen Konfidenzintervalls berechnet werden. Das Konfidenzniveau kann dabei frei gewählt werden. Diese Ergebnisse werden ausgegeben. Daneben wird der zeitliche Verlauf der kumulierten Einnahmen und Kosten einzeln und saldiert graphisch dargestellt.

Simulation des Nutzungsprozesses

Wir simulieren den Nutzungsprozeß mit der Monte-Carlo-Methode auf Basis einer vom Rechner generierten Zufallszahlsequenz so, daß die Anzahl der Nutzungsereignisse pro Zeiteinheit statistisch einer inhomogenen Poissonverteilung gehorcht, wobei die Nutzungsrate λ mit der Halbwertszeit T_h abnimmt (s.o.):

$$\lambda(t) = \lambda_0 e^{-\frac{\ln 2}{T_h} t}$$

Da die statistische Bibliothek der Sprache Matlab – wie die meisten derartigen Bibliotheken – nur gleichverteilte oder normalverteilte Zufallsvariablen zur Verfügung stellt, gewinnen wir poissonverteilte Variablen aus gleichverteilten durch Transformation der Wahrscheinlichkeitsdichtefunktion. Den präsentierten Simulationsergebnissen liegt eine Anzahl von 10.000 Iterationen zugrunde.

Numerische Nullstellenberechnung

Der interne Zinssatz ist derjenige Diskontierungszinssatz z_0, für den der Kapitalwert K Null ist (vgl. 2.4.4). Wir müssen also die Nullstelle des Kapitalwerts K als Funktion des Diskontierungszinssatzes z bestimmen. Da wir lediglich einzelne Werte dieser Funktion durch Simulation berechnen können, aber keine Information über die Steigung der Kurve besitzen, bestimmen wir diese Nullstelle mit Hilfe der Regula falsi in der verfeinerten Form des Pegasus-Verfahrens (vgl. [Bar99], 117).

Um den internen Zinssatz eindeutig zu bestimmen, müssen wir sicherstellen, daß genau eine Nullstelle existiert. Zunächst muß ausgeschlossen werden, daß mehrere Nullstellen existieren. Da dies analytisch nicht möglich ist, gehen wir dabei heuristisch vor: Wir simulieren für eine Vielzahl unterschiedlicher Parameterkonstellationen und Nutzungsverläufe den Kapitalwert K als Funktion des Zinssatzes z. Dabei zeigt sich, daß sämtliche Verläufe von $K(z)$ die Form ha-

ben, die Abbildung 6-2 beispielhaft zeigt, d.h. *K(z)* hat im positiven Bereich maximal eine Nullstelle.

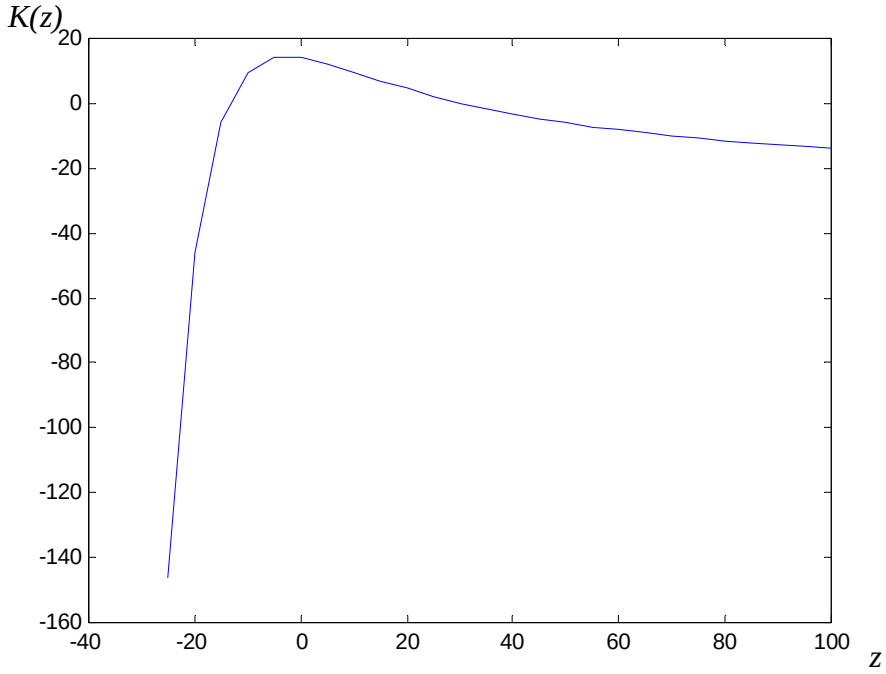

Abbildung 6-2: Kapitalwert als Funktion des Zinssatzes (typischer Verlauf)

Wenn eine Nullstelle existiert, so können wir sie also eindeutig bestimmen. Dazu stellen wir sicher, daß bei der numerischen Berechnung die Randbedingung $z > 0$ erfüllt ist.

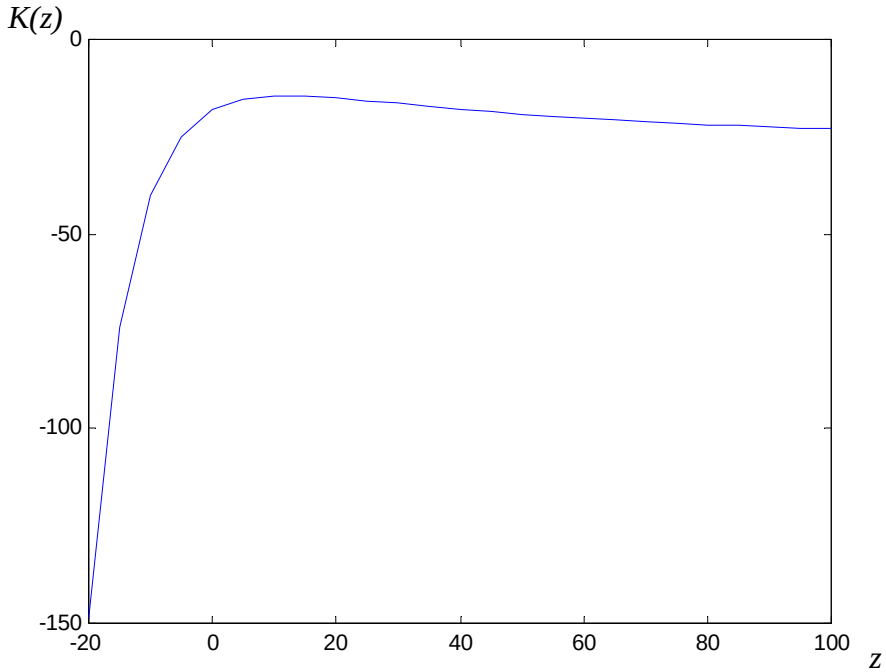

Abbildung 6-3: Kapitalwert als Funktion des Zinssatzes (Verlauf ohne Nullstelle)

Ob eine Nullstelle existiert, stellen wir fest, indem wir den Wert von *K(z)* bei *z = 0* berechnen. Ist *K(z = 0)* ≥ 0, so existiert eine Nullstelle, ist dagegen *K(z = 0)* < 0, so gibt es keine Nullstelle. Einen exemplarischen Verlauf von *K(z)* zeigt Abbildung 6-3.

Auch in diesem Fall läßt sich – in Erweiterung der klassischen Definition (vgl. 2.4.4) – sinnvoll ein interner Zinssatz bestimmen. Wir ermitteln das Maximum K_{max} von *K(z)*, wie in Abbildung 6-4 am Beispiel der Kurve aus Abbildung 6-3 gezeigt. Der entsprechende Zinssatz sei z_{max}; den Punkt (z_{max}, K_{max}) bezeichnen wir mit P_{max}. Dann stellen wir uns vor, daß das Koordinatensystem so nach untern verschoben wird, daß die *z*-Achse (gestrichelte Linie) die Kurve *K(z)* im Punkt P_{max} berührt. Diese Verschiebung entspricht einer Senkung der Anfangsinvestition I_{wv} um den Betrag ΔI. Wäre also die Anfangsinvestition I_{wv} um ΔI geringer, d.h. $\bar{I}_{wv} = I_{wv} - \Delta I$, dann wäre K_{max} eine Nullstelle. Wir zerlegen daher die Anfangsinvestition in zwei Teile, für die wir den internen Zinssatz getrennt bestimmen. Der interne Zinssatz für $\bar{I}_{wv}$ ist nach der obigen Überlegung z_{max}. Der verbleibende Teil der Investition, ΔI, geht komplett verloren; der interne Zinssatz ist also mit -100 Prozent anzusetzen. Folglich berechnet sich der interne Zinssatz *IRR* für die gesamte Investition zu

$$IRR = \frac{I_{wv} - \Delta I}{I_{wv}} z_{\max} - \frac{\Delta I}{I_{wv}} 100\%$$

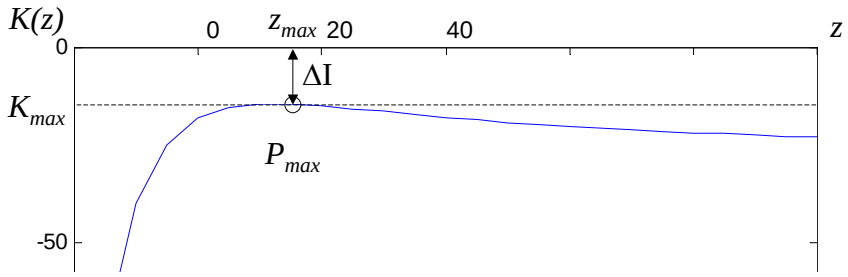

Abbildung 6-4: Bestimmung des internen Zinssatzes ohne Nullstelle

6.3 Exemplarische Simulationsergebnisse

Um das ökonomische Potential der Wiederverwendung mit Hilfe des ReValue-Modells zu untersuchen, haben wir exemplarische Szenarien definiert und simuliert. Die Ergebnisse präsentieren wir in diesem Abschnitt.

6.3.1 Szenarien

Da der von den Simulationsparametern aufgespannte Raum im Rahmen dieser Arbeit nicht vollständig untersucht werden konnte, haben wir typische Szenarien identifiziert, für die wir Simulationsergebnisse ermittelt haben.

Vorgehen bei Definition und Simulation

Die verschiedenen Szenarien orientieren sich an den Rahmenbedingungen der in Kapitel 5 vorgestellten sd&m AG. Sie wurden in Workshops mit projekterfahrenen Mitarbeitern definiert. Grundlage sind die Software-Kategorien gemäß 2.3.6. Zur Orientierung dienten die verfügbaren Werte aus der Literatur, die allerdings nicht nach Software-Kategorien differenzieren, sondern überwiegend für T-Software gelten (vgl. 3.5.1): nach Brooks ist bei der Generalisierung eines Programms insgesamt ein Aufwand notwendig, der das Neunfache der ursprünglichen Entwicklungskosten beträgt ([Bro95], 5); Jacobson et al. geben Faktoren zwischen 1,5 und 3 für den Mehraufwand an [JGJ97], 23]). Tracz nennt einen Faktor von zwei bis drei ([Tra95], xi). Für die Halbwertszeit wird ein Wert von 18 Monaten angegeben ([CuS95], 279)

Ausgangspunkt der Simulation ist ein Parametersatz, der den Arbeitspunkt darstellt. Für ihn werden Erwartungswert $E(IRR)$ und Standardabweichung σ_{IRR} des internen Zinssatzes sowie die Untergrenzen IRR_{95} und IRR_{99} der einseitigen Konfidenzintervalle für IRR mit Konfidenzniveaus von 95 respektive 99 Prozent[74] bestimmt.

Für die wesentlichen Parameter führen wir eine Sensitivitätsanalyse durch, d.h. wir lenken diese Parameter um zehn Prozent in beiden Richtungen aus dem Arbeitspunkt aus und ermitteln, wie sich das Ergebnis für IRR_{95} in der Folge ändert, wobei wir die Änderung mit ΔIRR_{95} bezeichnen. Die wesentlichen Parameter sind: Faktor der Investition in Wiederverwendbarkeit i_{wv}, anfängliche Nutzungsrate λ_0, Halbwertszeit der Nutzungsrate T_h, Nutzungskostenfaktor κ, asymptotischer Endwert des Wartungskostenfakors w. Aus dem Ergebnis der Sensitivitätsanalyse läßt sich insbesondere ablesen, wie sich Fehler bei der Schätzung der Parameter auf das Endergebnis auswirken. Diese Schätzfehler werden somit getrennt von der Investitionsunsicherheit behandelt, die durch die Zufälligkeit des Nutzungsprozesses entsteht.

Außerdem stellen wir für ausgewählte Fälle Projektionen auf eine der Ebenen des Parameterraums dar, indem wir einen größeren Wertebereich für den entsprechenden Parameter untersuchen und die anderen Parameter im Arbeitspunkt belassen.

[74] Kurz: 95- bzw. 99-Prozent-Konfidenzintervall

Überblick über die simulierten Szenarien

Die Szenarien stellen typische Beispiele für Komponenten der verschiedenen Software-Kategorien dar. Einen Überblick über die Parameterkombinationen im Arbeitspunkt gibt Tabelle 6-1. Auf die Parameterkonstellationen im einzelnen gehen wir in den folgenden Abschnitten ein.

Parameter		Szenario (Software-Kategorie)			
		Null	T	A	AT
Investition in Wiederverwendbarkeit (Faktor)	i_{wv}	5	1	10	10
Anfängliche Nutzungsrate p.a.	λ_0	10	5	5	5
Halbwertszeit (Jahre)	T_h	5	1	5	1
Nutzungskostenfaktor	κ	30%	20%	50%	50%
Wartungskostenfakor (asymptotischer Endwert)	w	5%	5%	5%	5%

Tabelle 6-1: Simulierte Szenarien

6.3.2 Null-Software

In unserem ersten Szenario untersuchen wir Komponenten der Kategorie Null-Software. Als Beispiel dafür legen wir eine Komponente für die Manipulation von Kalenderdaten zugrunde. Den Parametersatz im Arbeitspunkt für dieses Szenario zeigt Tabelle 6-2. Da Software der Kategorie Null von vielen Änderungen unabhängig ist, gehen wir von einer Halbwertszeit von fünf Jahren aus, die deutlich über dem Industriedurchschnitt von 18 Monaten (vgl. [CuS95], 279) liegt. Wir nehmen an, daß die Kosten für die Wiederverwendbarmachung das Fünffache der Entwicklung einer nicht systematisch wiederverwendbaren Komponente beträgt, was dadurch begründet ist, daß wir hohe Anforderungen an die Variabilität und damit verbunden erheblichen Generalisierungsaufwand erwarten. Gleichzeitig nehmen wir mit zehn Nutzungen pro Jahr die im Vergleich mit den anderen Szenarien höchste aller Raten an, da Null-Software grundsätzlich am besten wiederverwendbar ist. Wir gehen davon aus, daß die Nutzungskosten 30 Prozent der Kosten einer Neuentwicklung betragen, was zwischen den Werten für T-Software und A-Software liegt, da wir einerseits einen geringeren Standardisierungsgrad als bei T-Software und damit höheren Anpassungsaufwand, andererseits eine geringere Komplexität als bei A-Software erwarten. Die Wartungskosten schließlich betragen gemäß dem sd&m-Erfahrungswert fünf Prozent. Dieser Wert wird in allen Szenarien angesetzt.

Investition in Wiederverwend-barkeit (Faktor)	Anfängliche Nutzungsrate p.a.	Halbwerts-zeit (Jahre)	Nutzungs-kostenfaktor	Wartungskosten-fakor (asymptotisch)
i_{wv}	λ_0	T_h	κ	w
5	10	5	30%	5%

Tabelle 6-2: Parameter für Szenario Null-Software

Die Simulationsergebnisse im Arbeitspunkt zeigt Tabelle 6-3. Für den internen Zinssatz ergibt sich ein Erwartungswert von 214,4 Prozent, was einer sehr hohen Rendite entspricht. Die als Maß für die Investitionsunsicherheit dienende Standardabweichung beträgt 99,4, was zu Werten von 106,9 Prozent für IRR_{95} und 83,9 Prozent für IRR_{99} führt. Auch unter Berücksichtigung der Unsicherheit ergibt sich also eine sehr hohe Rendite.

E(IRR) (Prozent)	σ_{IRR}	IRR_{95} (Prozent)	IRR_{99} (Prozent)
214,4	99,4	106,9	83,9

Tabelle 6-3: Simulationsergebnisse Null-Software

Ausgelenkter Parameter	ΔIRR_{95} (absolut)		ΔIRR_{95} (relativ)	
	-10%	+10%	-10%	+10%
i_{wv}	14,3	-13,3	13,4%	-12,4%
λ_0	-17,9	15,6	-16,7%	14,6%
T_h	-6,8	2,3	-6,4%	2,2%
κ	3,7	-7,2	3,5%	-6,7%
w	-2,2	-1,6	-2,1%	-1,5%

Tabelle 6-4: Sensitivitätsanalyse Null-Software

Wir führen nun eine Sensitivitätsanalyse für die wesentlichen Parameter durch, d.h. wir lenken die Parameter jeweils einzeln aus dem Arbeitspunkt aus und bestimmen, wie sich die Ausgangsgröße IRR_{95} dadurch ändert. Dies ist auch hinsichtlich eventueller Fehler bei der Schätzung der Parameter von Bedeutung. Die Ergebnisse der Sensitivitätsanalyse sind in Tabelle 6-4 zusammengefaßt. Dabei ist die Änderung ΔIRR_{95} von IRR_{95} zum Wert im Arbeitspunkt in absoluter und relativer Form angegeben. Die größten Auslenkungen ergeben sich dabei für Änderungen von i_{wv} und λ_0. Die Maximalwerte haben als Betrag 13,4% bzw. 16,7%, was einer leichten Verstärkung der zehnprozentigen Auslenkung entspricht. Die Änderungen in Halbwertszeit T_h und Nutzungskostenfaktor κ sind mit 6,4% respektive 6,7% abgeschwächt. Die Änderung im War-

tungskostenfakor *w* fällt kaum ins Gewicht; auffällig ist jedoch, daß die Änderung in beiden Fällen negativ ist. Wir untersuchen das unten im Detail.

Um den Parameterraum weiträumiger zu erkunden, untersuchen wir, wie sich IRR_{95} als Funktion verschiedener Parameter verhält, wobei wir die jeweils anderen Parameter im Arbeitspunkt belassen. Dies entspricht einer Projektion auf verschiedene Ebenen des Parameterraums.

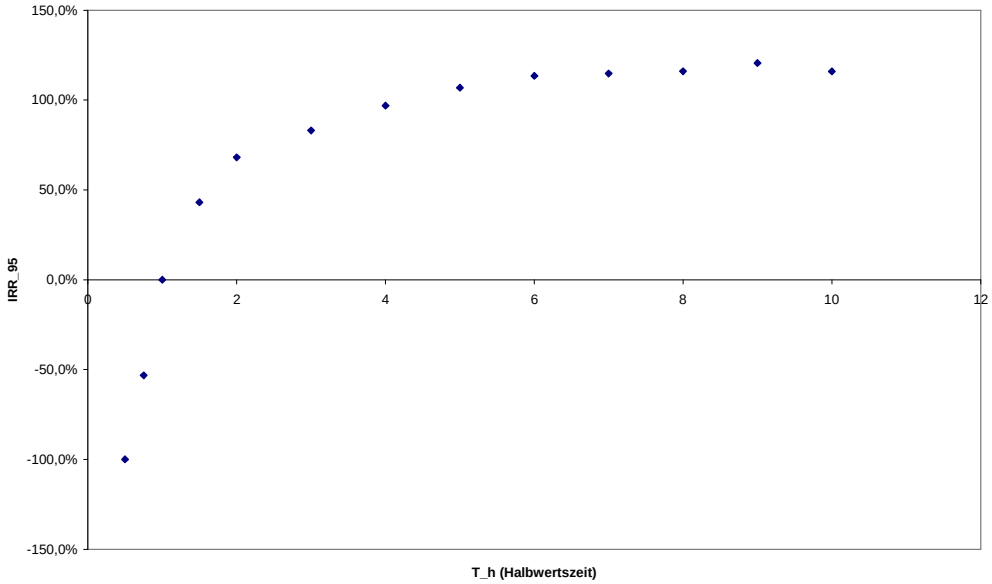

Abbildung 6-5: IRR_{95} als Funktion der Halbwertszeit (Null-Software)

Die Abhängigkeit von IRR_{95} von der Halbwertszeit T_h wird in Abbildung 6-5 dargestellt. Die Kurve verläuft um den Arbeitspunkt herum relativ flach und steigt für höhrere Werte von T_h wenig an; asymptotisch scheint sie sich einem Endwert von etwa 120 Prozent anzunähern. Das kann dadurch begründet werden, daß der Abzinsungseffekt mit steigendem Zinssatz immer stärker wird und so zukünftige Geldflüsse immer weniger ins Gewicht fallen.[75] Mit fallender Halbwertszeit wird die Kurve dagegen immer steiler. Bei einer Halbwertszeit von etwa einem Jahr hat sie eine Nullstelle und geht in den negativen Bereich. Es zeigt sich also, daß die Sensitivität auf Änderungen der Halbwertszeit mit fallender Halbwertszeit zunimmt.

In Abbildung 6-6 ist IRR_{95} als Funktion der anfänglichen Nutzungsrate λ_0 aufgetragen. Der Nulldurchgang der Kurve liegt bei einer Nutzungsrate von knapp unter drei Nutzungen pro Jahr. Darunter fällt die Kurve steil ab. Für steigende Werte von λ_0 nimmt die Steigung der Kurve etwas zu.

[75] Da die Kurve jedoch erst oberhalb eines internen Zinssatzes von über 50 Prozent merklich abflacht, spielt dieser Effekt keine wesentliche Rolle für die absolute Entscheidung über die Investition, da sich auf jeden Fall ein sehr vorteilhaftes Bild zeigt. Lediglich für den Vergleich verschiedener Alternativen ist dies ein Nachteil; dabei sollten allerdings ohnehin andere Faktoren berücksichtigt werden, die möglicherweise eine deutlichere Differenzierung erlauben.

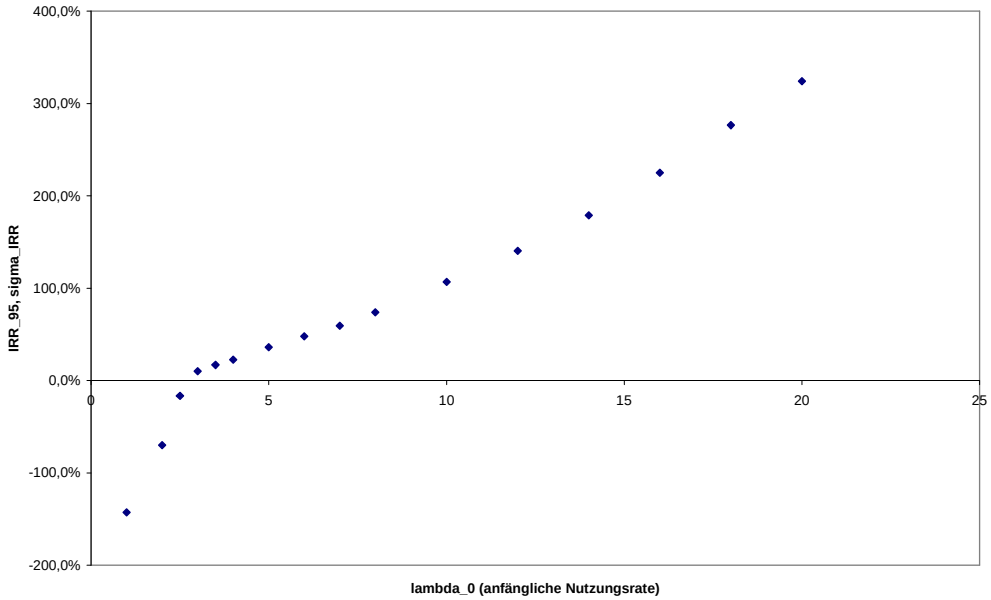

Abbildung 6-6: IRR_{95} als Funktion von λ_0 (Null-Software)

Abbildung 6-7 zeigt den Verlauf von IRR_{95} als Funktion der Investition in Wiederverwendbarkeit, gemessen durch den Faktor i_{wv}. Die Betrachtung der Kurve ergibt, daß der interne Zinssatz bei Werten von i_{wv}, die unterhalb des Arbeitspunkts liegen, stark ansteigt. Mit der Erhöhung der Investition in Wiederverwendbarkeit wird der interne Zinssatz immer kleiner und allmählich negativ. Der Nulldurchgang von IRR_{95} liegt zwischen den Werten $i_{wv} = 20$ und $i_{wv} = 25$, d.h. wenn die Investitionskosten zu hoch sind, werden sie von den Ersparnissen nicht mehr kompensiert.

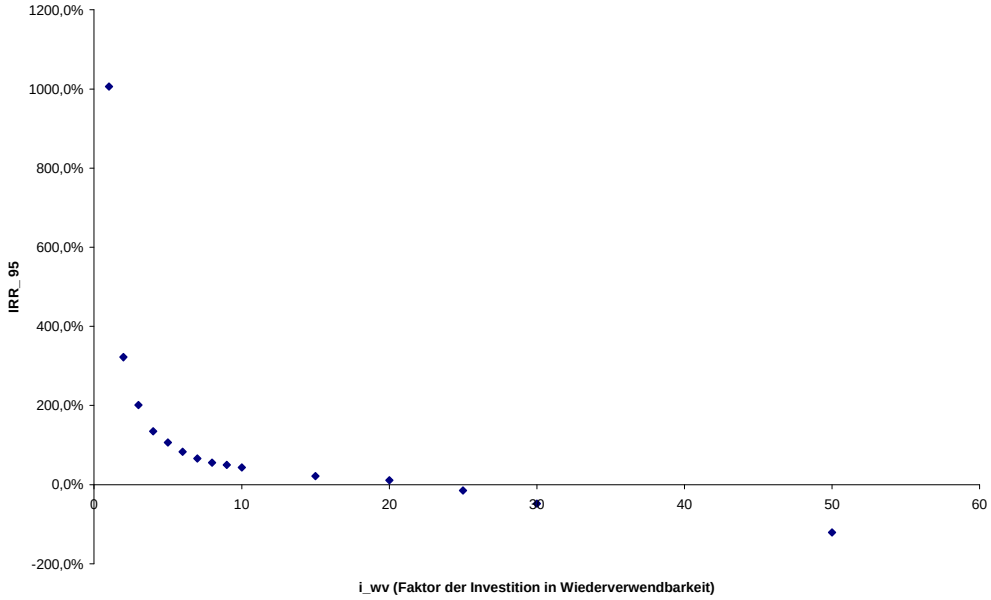

Abbildung 6-7: IRR_{95} als Funktion von i_{wv} (Null-Software)

Schließlich analysieren wir, wie sich der interne Zinssatz als Funktion von i_{wv} und λ_0 verhält. Dies ist in Form einer Kurvenschar in Abbildung 6-8 dargestellt. Die Kurven haben alle die gleiche Gestalt wie die für $\lambda_0 = 10$, die bereits

in Abbildung 6-7 dargestellt ist. Mit steigenden Werten von λ_0 werden die Kurven nach oben verschoben, für eine Erhöhung der Nutzungsrate steigt IRR_{95} also bei gleichbleibender Investition an.

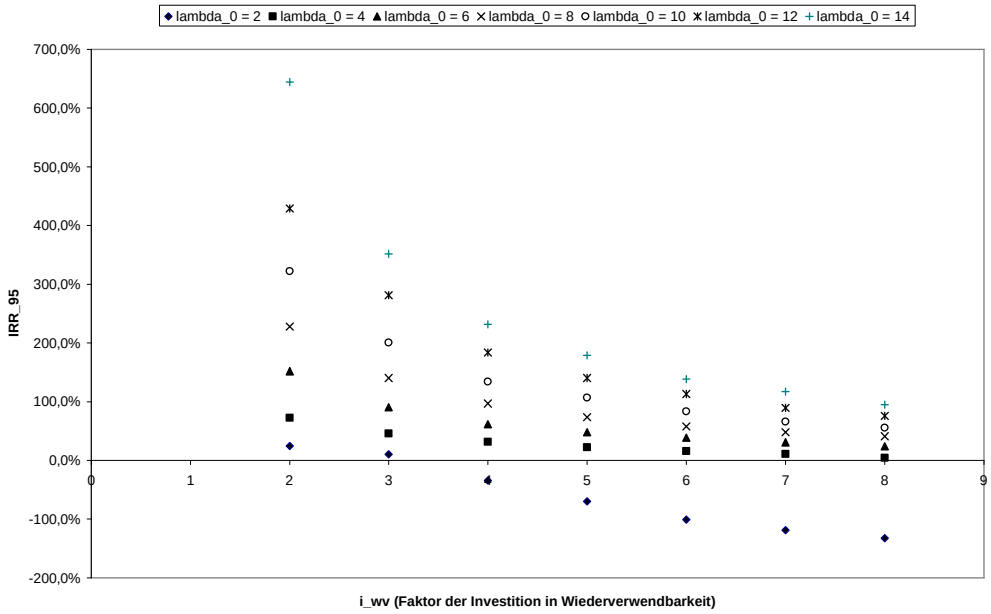

Abbildung 6-8: IRR_{95} als Funktion von i_{wv} und λ_0 (Null-Software)

Zur besseren Anschaulichkeit präsentieren wir in Abbildung 6-9 eine dreidimensionale Darstellung derselben Daten.

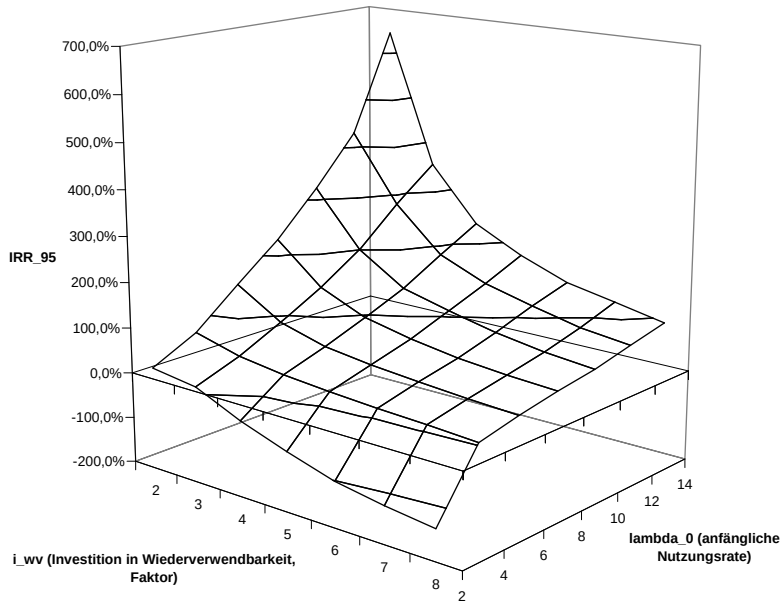

Abbildung 6-9: IRR_{95} als Funktion von i_{wv} und λ_0 (Null-Software) – dreidimensionale Darstellung

Schließlich analysieren wir den Einfluß des Wartungskostenfakor *w* auf IRR_{95} genauer. Dazu betrachten wir zunächst den in Abbildung 6-10 gezeigten Verlauf von IRR_{95} als Funktion von *w*. Die Kurve verläuft annähernd flach, mit ei-

nigen geringen Schwankungen. Die Höhe des Wartungskostenfakors *w* scheint auf IRR_{95} keinen Einfluß zu haben, was sich dadurch erklären läßt, daß sich die Kosten für die zentrale Wartung und die Ersparnisse bei der individuellen Wartung etwa ausgleichen, wie wir unten zeigen werden.

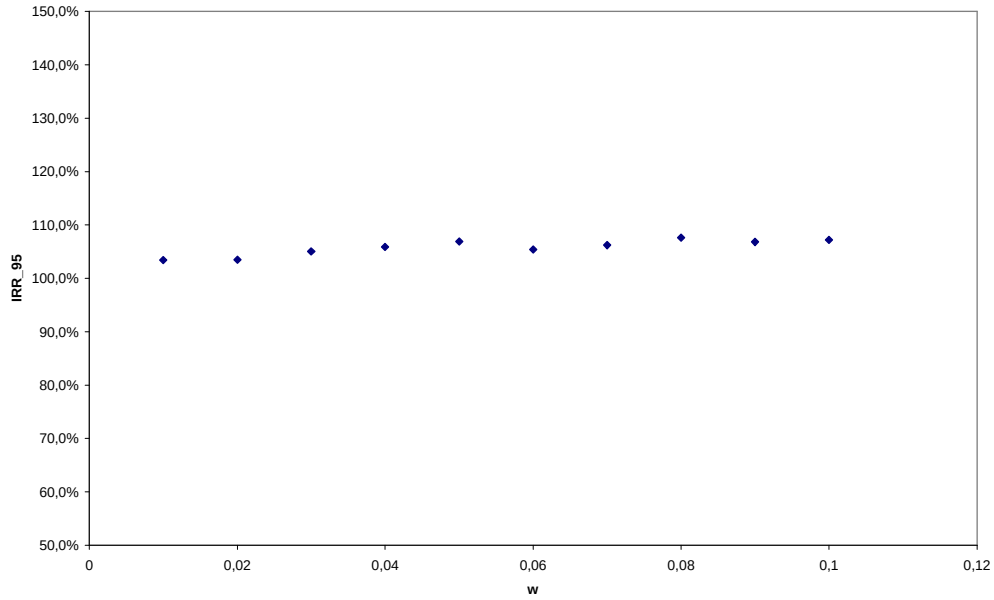

Abbildung 6-10: IRR95 als Funktion von w

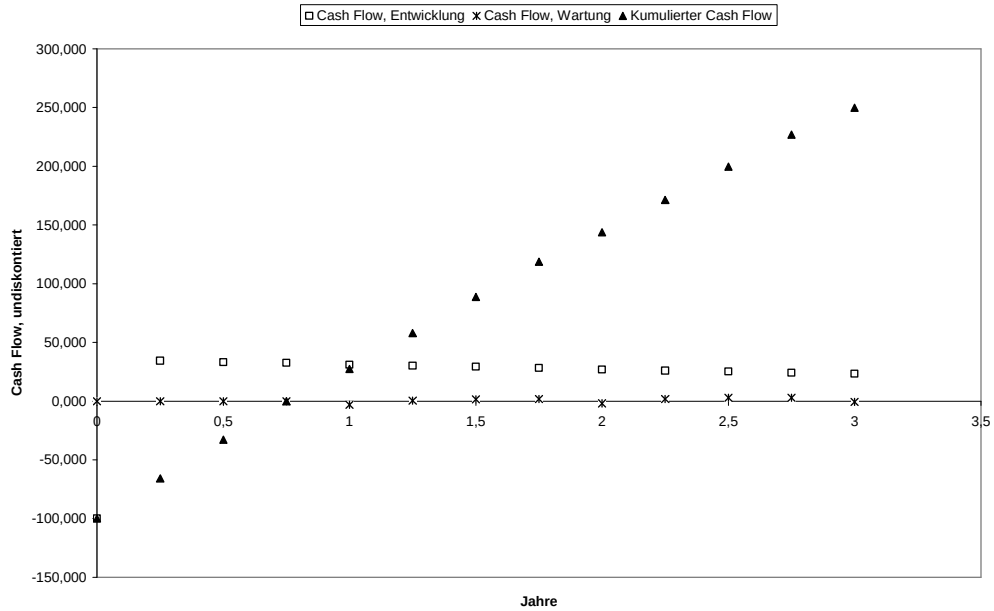

Abbildung 6-11: Zeitlicher Verlauf der Cash Flows (Null-Software)

Der zeitliche Verlauf der Erwartungswerte der Cash Flows für Entwicklung und Wartung in den ersten drei Jahren ist in Abbildung 6-11 dargestellt. Dabei sind Kosten und Ersparnisse für Entwicklung und Wartung jeweils saldiert; außerdem ist die kumulierte Summe aller Cash Flows aufgetragen. Der Nulldurchgang für den Erwartungswert des kumulierten Cash Flows, der dem nominalen Break-Even entspricht, erfolgt etwa nach einem dreiviertel Jahr. Die Investition ist also nach dieser Zeit nominal (ohne Abzinsung) amortisiert. Die

Cash Flows für Wartung oszillieren um den Wert Null, was die oben genannte These bestätigt, daß sich Kosten und Ersparnisse etwa ausgleichen.

Um auch bei dieser Untersuchung die Investitionsunsicherheit zu berücksichtigen, untersuchen wir die Untergrenze des 95-Prozent-Konfidenzintervalls für den kumulierten Cash Flow. Der Verlauf ist im Vergleich zum Erwartungswert in Abbildung 6-12 dargestellt. Da der Verlauf der 95-Prozent-Untergrenze einem konkreten Nutzungsfall entspricht, ist er nicht geglättet wie der Erwartungswert. Deshalb ist außerdem eine Regressionsgerade aufgetragen, die einen Nulldurchgang zwischen 1,25 und 1,5 Jahren hat. Bei Berücksichtigung der Unsicherheit liegt die Amortisationszeit also mehr als ein halbes Jahr über der Zeit für den Erwartungswert.

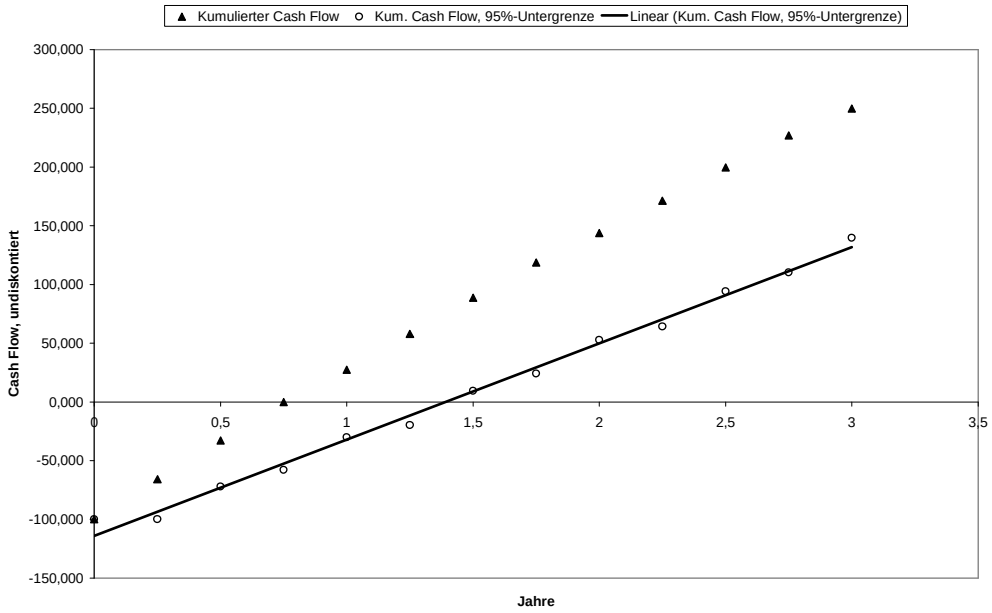

Abbildung 6-12: Erwartungswert und 95-Prozent-Untergrenze der kumulierten Cash Flows (Null-Software)

Das hier beschriebene Null-Software-Szenario ist durch folgende Merkmale gekennzeichnet:

- Die zu erwartende Rendite der Investition in Wiederverwendbarkeit, gemessen durch den Erwartungswert des internen Zinssatzes, ist mit 214,4 Prozent sehr hoch.
- Die Investitionsunsicherheit aufgrund des zufälligen Nutzungsprozesses ist mäßig: Die Standardabweichung des internen Zinssatzes liegt mit 99,4 deutlich niedriger als der Erwartungswert. Dies bedeutet, daß die um diese Unsicherheit bereinigte Rendite ebenfalls sehr hoch ist: IRR_{95} beträgt 106,9 Prozent, IRR_{99} beträgt 83,9 Prozent.
- Der Erwartungswert für den Break-Even liegt etwa bei 0,75 Jahren, die Untergrenze für das 95-Prozent-Konfidenzintervall zwischen 1,25 und 1,5 Jahren. Diese Zeitpunkte entsprechen der Amortisation der Investition zu Nominalwerten.
- Der Wert von IRR_{95} steigt mit steigender Halbwertszeit, steigender Nutzungsrate und fallender Investition in die Wiederverwendbarkeit. Ände-

rungen der Parameter um den Arbeitspunkt wirken sich maximal mit etwa 1,7-facher Verstärkung auf IRR_{95} aus. Verstärkungen ergeben sich für den Faktor der Investition in Wiederverwendbarkeit i_{wv} und die anfängliche Nutzungsrate λ_0, wobei die Faktoren maximal ca. 1,4 respektive ca. 1,7 betragen. Änderungen der Halbwertszeit T_h, des Nutzungskostenfaktors κ und des Wartungskostenfakors w werden dagegen abgeschwächt.

6.3.3 T-Software

Als nächstes analysieren wir die ökonomischen Aspekte wiederverwendbarer Komponenten der Kategorie T-Software, wobei wir das Beispiel eines Datenbankadapters analog zu dem in 5.2.1 beschriebenen wählen. Der Parametersatz für den Arbeitspunkt dieses Szenarios ist in Tabelle 6-5 dargestellt. T-Software hängt von Änderungen der technischen Systemumgebung ab und hat deshalb eine kürzere Halbwertszeit als andere Software-Kategorien; wir gehen hier von einem Jahr aus. Was die Investition in Wiederverwendbarkeit angeht, macht sich der hohe Standardisierungsgrad der Schnittstellen (APIs) der technischen Systemumgebung positiv bemerkbar. Wir nehmen daher einen Faktor von i_{wv} = 1 an. Ein weiterer Vorteil der hohen Standardisierung ist, daß die Nutzung vereinfacht wird. Daher setzen wir für T-Software mit 20 Prozent den niedrigsten Nutzungskostenfaktor an. Da T-Software nur in bestimmten Systemumgebungen eingesetzt werden kann, liegt die Nutzungsrate grundsätzlich niedriger als bei Null-Software. Wir setzen sie in unserem Beispiel mit fünf Nutzungen pro Jahr an.

Investition in Wiederverwendbarkeit (Faktor)	Anfängliche Nutzungsrate p.a.	Halbwertszeit (Jahre)	Nutzungskostenfaktor	Wartungskostenfakor (asymptotisch)
i_{wv}	λ_0	T_h	κ	W
1	5	1	20%	5%

Tabelle 6-5: Parameter für Szenario T-Software

Die Simulationsergebnisse für den Arbeitspunkt faßt Tabelle 6-6 zusammen. Als Erwartungswert des internen Zinssatzes ergibt sich mit 1746 Prozent ein sehr hoher Wert. Dasselbe gilt allerdings für die Standardabweichung σ_{IRR}, die mit 3345 etwa doppelt so hoch ist. Entsprechend liegen die Untergrenzen der Konfidenzintervalle deutlich unter dem Erwartungswert: während jedoch IRR_{95} eine Größenordnung darunter liegt, aber mit 89,4 immer noch einer hohen Rendite entspricht, ist IRR_{99} bereits negativ.

E(IRR) (Prozent)	σ_{IRR}	IRR_{95} (Prozent)	IRR_{99} (Prozent)
1746	3345	89,4	-10,0

Tabelle 6-6: Simulationsergebnisse T-Software

Die Ergebnisse der Sensitivitätsanalyse für die wichtigsten Parameter finden sich in Tabelle 6-7. Der Einfluß von Schwankungen in i_{wv}, λ_0 und T_h auf IRR_{95} ist bei T-Software deutlich höher als beim Null-Software-Szenario (siehe Tabelle 6-4). Die Verstärkungsfaktoren betragen maximal ca. 2,2 für den Investitionsfaktor i_{wv}, ca. 3,9 für die anfängliche Nutzungsrate λ_0 und ca. 1,7 für die Halbwertszeit T_h.

Ausgelenkter Parameter	ΔIRR_{95} (absolut)		ΔIRR_{95} (relativ)	
	-10%	+10%	-10%	+10%
i_{wv}	17,0	-19,2	19,0%	-21,5%
λ_0	-20,8	34,5	-23,3%	38,6%
T_h	-14,7	10,7	-16,4%	12,0%
κ	2,9	-5,6	3,2%	-6,3%
w	-2,7	1,9	-3,0%	2,1%

Tabelle 6-7: Sensitivitätsanalyse T-Software

Wir analysieren dazu IRR_{95} als Funktion der Halbwertszeit. Wie Abbildung 6-13 zeigt, liegt der Arbeitspunkt in einem deutlich steileren Bereich der Kurve, die eine ähnliche Form hat wie die in Abbildung 6-5 gezeigte für Null-Software. Der Nulldurchgang für das T-Software-Szenario liegt jedoch niedriger, nämlich bei einer Halbwertszeit von weniger als einem dreiviertel Jahr. Für hohe Werte von T_h nähert sich die Kurve asymptotisch einem Endwert von etwa 300 Prozent für IRR_{95} an.

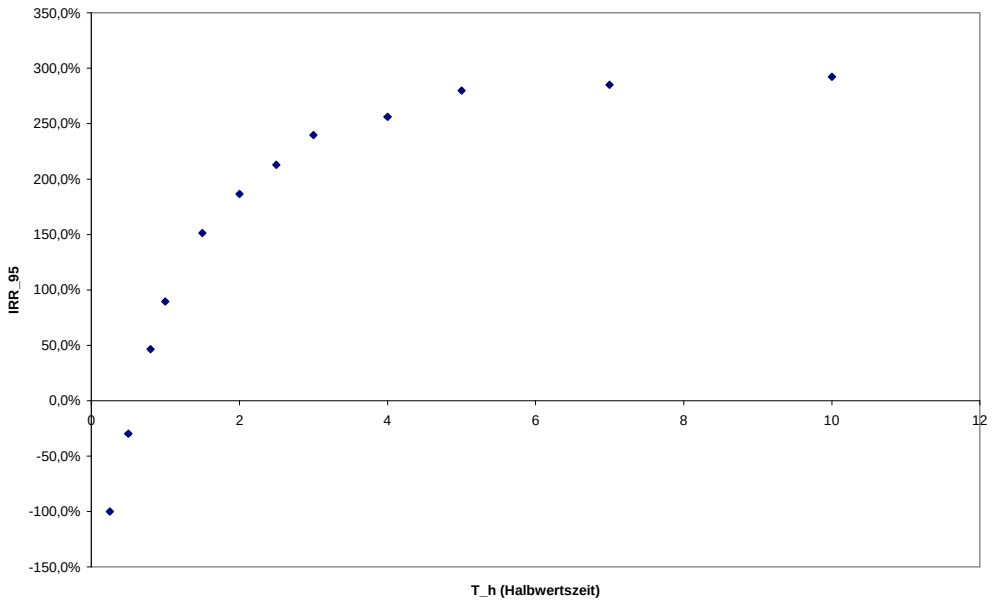

Abbildung 6-13: IRR_{95} als Funktion der Halbwertszeit T_h (T-Software)

In Abbildung 6-14 ist IRR_{95} als Funktion der anfänglichen Nutzungsrate λ_0 aufgetragen. Der Nulldurchgang der Kurve liegt bei einer Nutzungsrate von etwas über drei Nutzungen pro Jahr. Für Werte von zwei Nutzungen pro Jahr und darunter ergibt sich für IRR_{95} ein konstanter Wert von minus 100 Prozent, was darauf hindeutet, daß die Untergrenze des Konfidenzintervalls Fällen entspricht, in denen kein Nutzungsereignis stattfindet, so daß die Investition völlig verloren geht. Für steigende Werte von λ_0 nimmt die Steigung der Kurve ebenso wie bei Null-Software (vgl. Abbildung 6-6) leicht zu.

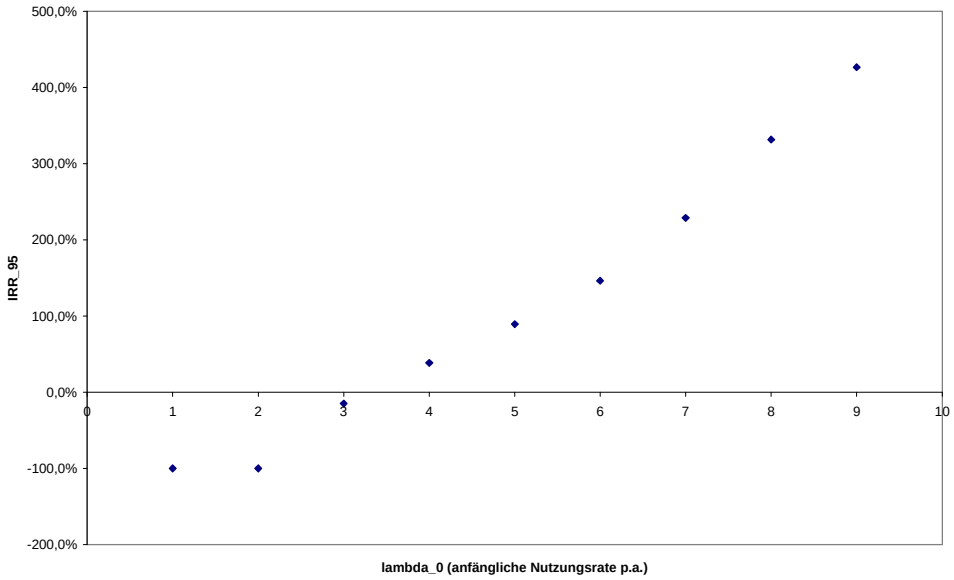

Abbildung 6-14: IRR_{95} als Funktion von λ_0 (T-Software)

Der in Abbildung 6-15 gezeigte Verlauf von IRR_{95} als Funktion der Investition in Wiederverwendbarkeit, gemessen durch den Faktor i_{wv}, hat eine ähnliche Form wie für Null-Software (vgl. Abbildung 6-7), allerdings findet der Nulldurchgang bereits bei einem Wert von i_{wv} statt, der zwischen 2 und 2,5 liegt.

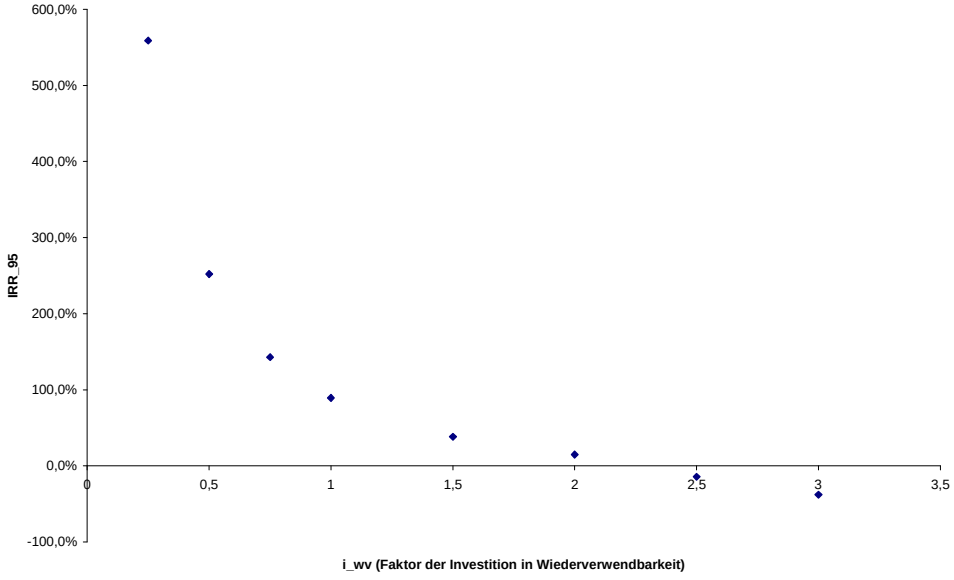

Abbildung 6-15: IRR_{95} als Funktion von i_{wv} (T-Software)

Den zeitlichen Verlauf der Erwartungswerte der Cash Flows für Entwicklung und Wartung in den ersten drei Jahren zeigt Abbildung 6-16. Analog zu dem Verlauf für Null-Software (vgl. Abbildung 6-11) sind Kosten und Ersparnisse für Entwicklung und Wartung jeweils saldiert aufgetragen, zusätzlich die kumulierte Summe aller Cash Flows. Der Nulldurchgang für den Erwartungswert des kumulierten Cash Flows – d.h der Break-Even – erfolgt etwa nach einem viertel Jahr, also deutlich früher als bei Null-Software.

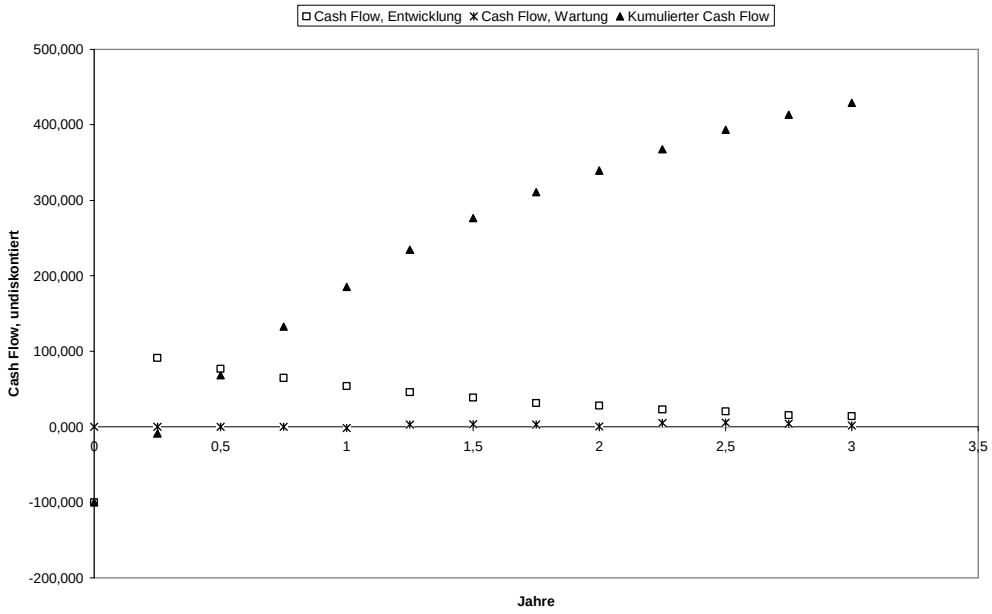

Abbildung 6-16: Zeitlicher Verlauf der Cash Flows (T-Software)

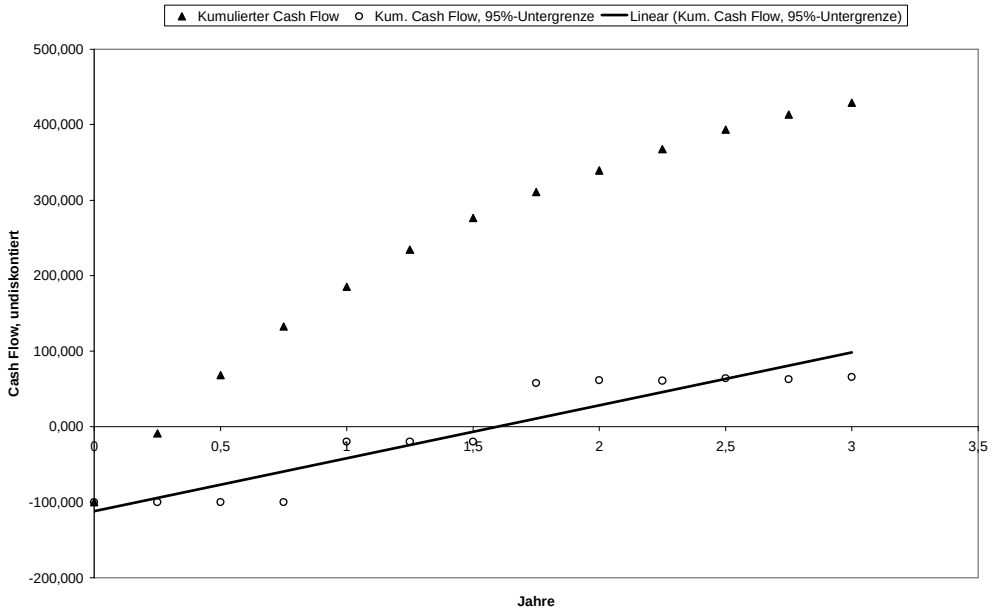

Abbildung 6-17: Erwartungswert und 95-Prozent-Untergrenze der kumulierten Cash Flows (T-Software)

Auch in diesem Fall wollen wir die Investitionsunsicherheit berücksichtigen und ermitteln daher die Untergrenze des 95-Prozent-Konfidenzintervalls für

den kumulierten Cash Flow. In Abbildung 6-17 ist dieser Verlauf im Vergleich zum Erwartungswert dargestellt. Die Regressionsgerade hat eine Nullstelle knapp oberhalb von 1,5 Jahren. Bei Berücksichtigung der hohen Investitionsunsicherheit verlängert sich die Amortisationszeit also etwa um den Faktor sechs.

Zusammenfassend sind die charakteristischen Merkmale des T-Software-Szenarios:

- Der Erwartungswert des internen Zinssatzes *IRR* ist mit 1746 Prozent äußerst hoch, was zudem eine starke Auswirkung der Abzinsung bedeutet: Der Diskontierungsfaktor nach einem Jahr beträgt etwa 0,06.
- Die Investitionsunsicherheit aufgrund des zufälligen Nutzungsprozesses ist ebenfalls äußerst hoch: Die Standardabweichung des internen Zinssatzes liegt mit 3345 Prozent noch über dem Erwartungswert. Die um die Unsicherheit bereinigte Rendite ist, gemessen durch IRR_{95}, mit 89,4 Prozent immer noch sehr hoch, liegt aber bereits etwa um Faktor 20 unter dem Erwartungswert für *IRR*. Legt man ein höheres Sicherheitsbedürfnis zugrunde und zieht IRR_{99} für die Messung der Rendite heran, so ergibt sich ein negativer Wert.
- Der Erwartungswert für den Zeitpunkt des Break-Even, der der Amortisation der Investition zu Nominalwerten entspricht, liegt etwa bei 0,25 Jahren, die Untergrenze für das 95-Prozent-Konfidenzintervall bei über 1,5 Jahren.
- Das grundsätzliche Verhalten von IRR_{95} als Funktion der verschiedenen Parameter entspricht dem bei Null-Software beobachteten: der Wert steigt mit steigender Halbwertszeit, steigender Nutzungsrate und fallender Investition in die Wiederverwendbarkeit. Änderungen der Parameter um den Arbeitspunkt wirken sich insgesamt stärker – mit bis zu beinahe vierfacher Verstärkung (für λ_0) – auf IRR_{95} aus. Der wesentliche Unterschied zum Null-Software-Szenario liegt darin, daß die Sensitivität auf Schwankungen in der Halbwertszeit T_h deutlich höher ist. Der Verstärkungsfaktor liegt für eine Verringerung von T_h etwa bei 1,6.

6.3.4 A-Software

Für die ökonomische Analyse von wiederverwendbaren Komponenten der Kategorie A-Software wählen wir als Beispiel eine Komponente für die Kundenverwaltung in einem betrieblichen Informationssystem. Tabelle 6-8 faßt die für den Arbeitspunkt gewählten Parameter zusammen. Wir gehen wegen der geringen Standardisierung bei Anwendungskomponenten davon aus, daß eine sehr hohe Variabilität erforderlich ist und setzen daher für die nötige Investition in Wiederverwendbarkeit den Faktor 10 an. Da sich die Rahmenbedingungen für Anwendungskomponenten deutlich langsamer ändern als für technische Komponenten, setzen wir die Halbwertszeit mit fünf Jahren deutlich höher, nämlich genauso hoch wie bei Null-Software-Komponenten an. Aufgrund der hohen Komplexität erwarten wir den höchsten Anpassungsaufwand der drei Kategorien A, T und Null. Wir wählen deshalb einen Nutzungskostenfaktor von 50 Prozent.

Investition in Wiederverwend-barkeit (Faktor)	Anfängliche Nutzungsrate p.a.	Halbwerts-zeit (Jahre)	Nutzungs-kostenfaktor	Wartungskosten-fakor (asymptotisch)
i_{wv}	λ_0	T_h	κ	w
10	5	5	50%	5%

Tabelle 6-8: Parameter für Szenario A-Software

Die Simulationsergebnisse im Arbeitspunkt zeigt Tabelle 6-9. Bereits der Erwartungswert des internen Zinssatzes ist mit -4,1 Prozent negativ. Die Standardabweichung von 23,8 schlägt sich in deutlich negativen Werten von -54,9 Prozent für IRR_{95} und -78,3 Prozent für IRR_{99} nieder, so daß aus ökonomischer Sicht die Wiederverwendbarmachung in diesem Szenario nicht sinnvoll ist.

E(IRR) (Prozent)	σ_{IRR}	IRR_{95} (Prozent)	IRR_{99} (Prozent)
-4,1	23,8	-54,9	-78,3

Tabelle 6-9: Simulationsergebnisse A-Software

Bei der Betrachtung der Sensitivitätsanalyse (siehe Tabelle 6-10) fällt im Vergleich zu den anderen Szenarien besonders der hohe Einfluß des Nutzungskostenfaktors κ auf, für den die Verstärkung ca. 3,3 beträgt. Auch für den Wartungskostenfaktor w ist eine leichte Verstärkung zu konstatieren. Für i_{wv} beträgt die maximale Verstärkung ca. 3,6, für λ_0 ca. 3,9 und für T_h ca. 2,3.

Ausgelenkter Parameter	ΔIRR_{95} (absolut)		ΔIRR_{95} (relativ)	
	-10%	+10%	-10%	+10%
i_{wv}	19,6	-16,2	35,7%	-29,5%
λ_0	-17,3	21,3	-31,5%	38,8%
T_h	-10,9	12,3	-19,9%	22,4%
κ	17,9	-17,7	32,6%	-32,2%
w	7,0	-6,6	12,8%	-12,0%

Tabelle 6-10: Sensitivitätsanalyse für A-Software

Wie bei den anderen Szenarien untersuchen wir den Verlauf von IRR_{95} als Funktion der Halbwertszeit T_h. Er ist in Abbildung 6-18 dargestellt. Die Kurve zeigt die bekannte Form, hat jedoch den Nulldurchgang erst bei einer sehr hohen Halbwertszeit von etwa sieben Jahren.

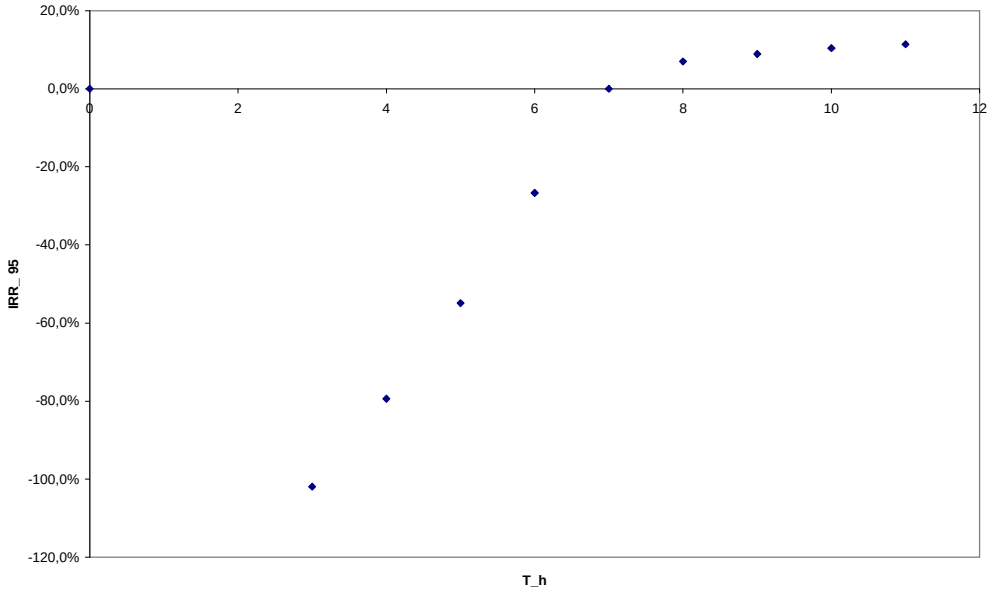

Abbildung 6-18: IRR_{95} als Funktion der Halbwertszeit T_h (A-Software)

Den Verlauf von IRR_{95} als Funktion der Investition in Wiederverwendbarkeit, gemessen durch den Faktor i_{wv}, zeigt Abbildung 6-19. Der Nulldurchgang liegt zwischen den Werten $i_{wv} = 7$ und $i_{wv} = 8$.

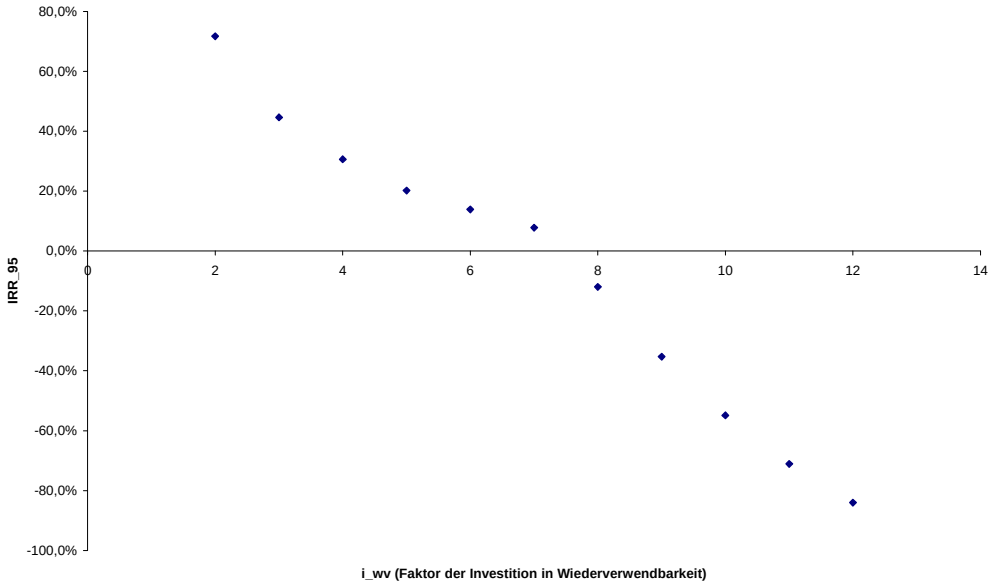

Abbildung 6-19: IRR_{95} als Funktion von i_{wv} (A-Software)

Um die Abhängigkeit zwischen IRR_{95} und κ näher zu untersuchen, tragen wir IRR_{95} in Abbildung 6-20 als Funktion von κ auf. Dabei zeigt sich, daß die Kurve eine Nullstelle zwischen $\kappa = 0{,}3$ und $\kappa = 0{,}4$ hat und im negativen Bereich, in dem der Arbeitspunkt liegt, deutlich steiler verläuft. Das erklärt die hohe Empfindlichkeit von IRR_{95} gegenüber Schwankungen in κ.

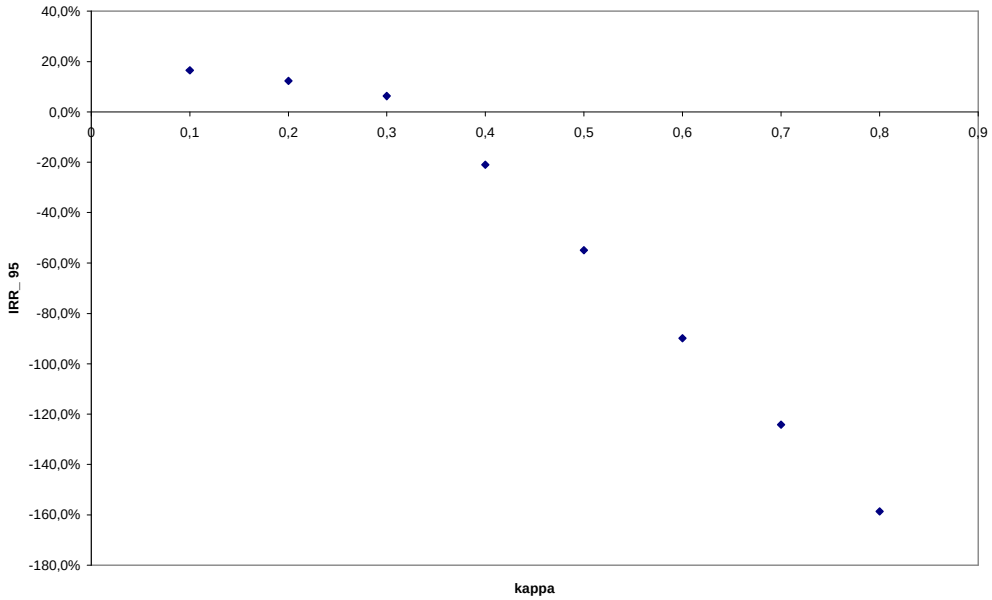

Abbildung 6-20: IRR_{95} als Funktion von κ (A-Software)

In Abbildung 6-21 ist IRR_{95} als Funktion der anfänglichen Nutzungsrate λ_0 aufgetragen. Auch diese Kurve zeigt in der Form große Ähnlichkeit mit der des Szenarios Null-Software. Die Nullstelle von IRR_{95} liegt für das A-Software-Szenario allerdings bei einer deutlich höheren Nutzungsrate λ_0 von zwischen sechs und sieben.

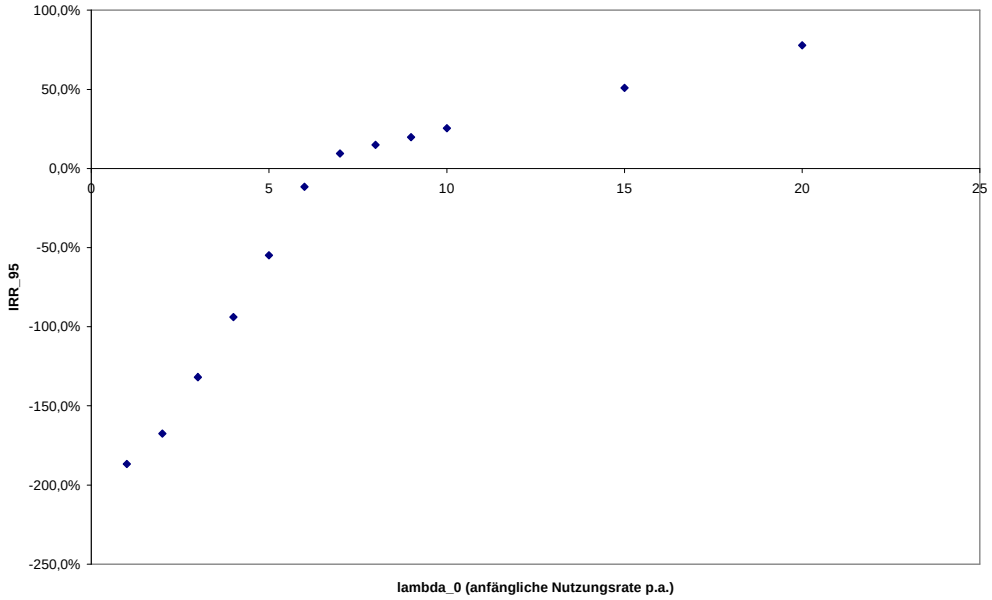

Abbildung 6-21: IRR_{95} als Funktion von λ_0 (A-Software)

Im Folgenden fassen wir kurz die wesentlichen Merkmale des A-Software-Szenarios zusammen:

- Der Erwartungswert des internen Zinssatzes ist negativ: er liegt bei -4,1 Prozent.
- Die Investitionsunsicherheit aufgrund des zufälligen Nutzungsprozesses ist sehr hoch. Die Standardabweichung des internen Zinssatzes liegt eine Größenordnung über dem Erwartungswert: sie beträgt 23,8 Prozent. Bei Berücksichtigung dieser Unsicherheit ergibt sich eine Rendite, die sich, gemessen als IRR_{95}, auf -54,9 Prozent beläuft. Dies bedeutet, daß mehr als die Hälfte der Investitionssumme verloren geht.
- Auch hier entspricht das grundsätzliche Verhalten von IRR_{95} als Funktion der verschiedenen Parameter demjenigen, das wir bei Null-Software beobachten. Die Sensitivitätsanalyse zeigt die bisher höchsten Ausschläge. Änderungen der Parameter um den Arbeitspunkt wirken sich mit über dreifacher Verstärkung für i_{wv}, λ_0 und κ aus. Die größten Unterschiede zu den beiden bereits untersuchten Szenarien ergeben sich für den Investitionsfaktor i_{wv} und den Nutzungskostenfaktor κ.

6.3.5 AT-Software

Obwohl AT-Software, wie in 2.3.6 dargelegt, grundsätzlich zu meiden ist, untersuchen wir sie auch unter ökonomischen Gesichtspunkten. Damit belegen wir, daß die angeführten softwaretechnischen Nachteile sich auch ökonomisch negativ auswirkten. Der in Tabelle 6-11 gezeigte Parametersatz für den Arbeitspunkt des Szenarios wurde durch Kombination der Parameter für T-Software und A-Software gewonnen, wobei wegen der Kombination der beiden jeweils der ungünstigere Fall angenommen werden muß.

Investition in Wiederverwendbarkeit (Faktor)	Anfängliche Nutzungsrate p.a.	Halbwertszeit (Jahre)	Nutzungskostenfaktor	Wartungskostenfakor (asymptotisch)
i_{wv}	λ_0	T_h	κ	w
10	5	1	50%	5%

Tabelle 6-11: Parameter für Szenario AT-Software

Die Simulationsergebnisse für das Szenario sind in Tabelle 6-12 zusammengefaßt. Sie sprechen für sich: bereits der Erwartungswert des internen Zinssatzes liegt unter -100 Prozent und damit deutlich schlechter als für die beiden Einzel-Kategorien A und T. Es wird also mehr Geld verloren als investiert. Das ist dadurch zu erklären, daß zusätzlich zur Anfangsinvestition Wartungskosten anfallen, die bei entsprechend ungünstiger Konstellation den Verlust erhöhen können.

E(IRR) (Prozent)	σ_{IRR}	IRR_{95} (Prozent)	IRR_{99} (Prozent)
-113,3%	16,2%	-134,2%	-135,5%

Tabelle 6-12: Simulationsergebnisse für AT-Software

Die obige Behauptung ist somit belegt. Wir sehen von einer detaillierteren Analyse ab, da AT-Software aus softwaretechnischen Gründen ohnehin gemieden werden sollte.

6.4 Interpretation der Simulationsergebnisse und Schlußfolgerungen

Im folgenden interpretieren wir die Simulationsergebnisse im Zusammenhang und ziehen allgemeine Schlußfolgerungen, insbesondere für die empirisch zu überprüfenden Hypothesen.

6.4.1 Interpretation der Simulationsergebnisse im Zusammenhang

Wesentliche Ergebnisse der Analyse der Szenarien für Null-, T- und A-Software faßt Tabelle 6-13 zusammen. Da wir AT-Software nicht im Detail analysiert haben (s.o.), erscheint das Szenario nicht in der Tabelle.

Szenario	Rendite für Konfidenzniveau		Parameter, deren Schwankungen verstärkt werden	Maximale Verstärkung
	95 Prozent	99 Prozent		
Null-Software	Sehr hoch	Sehr hoch	i_{wv}, λ_0	1,7
T-Software	Sehr hoch	Verlust	i_{wv}, λ_0, T_h	3,9
A-Software	Sehr hoher Verlust	Sehr hoher Verlust	i_{wv}, λ_0, T_h, κ, w	3,9

Tabelle 6-13: Wesentliche Ergebnisse der Szenarienanalyse

Aus der Gesamtheit der Ergebnisse lassen sich die folgenden generellen Schlußfolgerungen ableiten:

1. Die Investition in die Herstellung wiederverwendbarer Software lohnt sich entgegen der weitverbreiteten Überzeugung nicht immer. Je nach den Rahmenbedingungen, die sich in der Parameterkonstellation ausdrücken, ergeben sich große Unterschiede in der zu erwartenden Rendite. Die Schwankungsbreite reicht dabei von sehr hohen Gewinnen bis

zu sehr hohen Verlusten. Auch die Investitionsunsicherheit schwankt sehr stark mit den Rahmenbedingungen. Pauschale Aussagen über die Rentabilität wiederverwendbarer Software allgemein sind daher nicht sinnvoll, wenn nicht gar gefährlich.

2. Aus diesen Gründen sollte die Investition in Wiederverwendbarmachung auf Basis eines detaillierten Modells genau analysiert werden. Eine mögliche Methode dafür ist die Anwendung des hier vorgestellten ReValue-Modells, das es ermöglicht, gewinnbringende von verlustträchtigen Szenarien zu unterscheiden. Dabei ist es wichtig, die Wartung zu berücksichtigen, denn bei geringer Nutzung entsteht durch die Wartungsverpflichtung ein zusätzlicher Verlust, der sogar dazu führen kann, daß der Gesamtverlust die ursprüngliche Investition übersteigt.
3. Die Zeitverzögerung, mit der der Break-Even (d.h. die Amortisation der Investition zu Nominalwerten) eintritt, ist je nach Szenario und Berücksichtigung der Unsicherheit unterschiedlich: wir beobachten Erwartungswerte zwischen einem viertel Jahr (für T-Software) und einem dreiviertel Jahr (für Null-Software). Die Untergrenze eines 95-Prozent-Konfidenzintervalls liegt für diese beiden Werte bei über eineinviertel respektive eineinhalb Jahren. Der Nutzen, insbesondere die Produktivitätsverbesserung, tritt also mit erheblicher Verzögerung ein.
4. Das ökonomische Potential verschiedener Software-Kategorien bezüglich der Wiederverwendbarkeit ist sehr unterschiedlich. Am besten schneidet Null-Software ab, die sehr hohe Renditen bei geringer Unsicherheit verspricht und zudem am wenigsten empfindlich auf Fehler bei der Parameterschätzung reagiert. Sehr hohe Renditen können auch mit wiederverwendbaren Komponenten der Kategorie T-Software erzielt werden, allerdings bei ebenfalls sehr hoher Investitionsunsicherheit und größerer Empfindlichkeit auf Schätzfehler in den Parametern. A-Software zeigt geringes ökonomisches Potential bei gleichzeitig hoher Unsicherheit und der im Vergleich höchsten Empfindlichkeit auf Paramterschwankungen. Von der Entwicklung wiederverwendbarer AT-Software ist auch unter ökonomischen Aspekten nachdrücklich abzuraten, da sie sehr hohe Verluste bringt.

Schließlich stellen die Ergebnisse für die präsentierten Szenarien einen Beitrag zur Exploration des Parameterraums insgesamt dar. Die Analyse trägt so, unabhängig von der Wahl der Arbeitspunkte der Szenarien, zu einem besseren intuitiven Verständnis der ökonomischen Zusammenhänge bei.

6.4.2 Schlußfolgerungen für die empirisch zu überprüfenden Hypothesen

Für die in 3.5.3 formulierten Hypothesen zu den ökonomischen Aspekten ergeben sich aus den in 6.3 präsentierten Ergebnissen dieses Kapitels folgende Schlußfolgerungen: Hypothese [GroßesPot] ist widerlegt, während sich Hypothese [Invest] bestätigt (vgl. Aussage 1 in 6.4.1). Hypothese [Varianz] bestätigt

sich ebenfalls (vgl. Tabelle 6-13). Eine aktualisierte Übersicht über die betroffenen Hypothesen zeigt Tabelle 6-14.

Nr	Hypothese	Ergebnis	Änderung
22	[GroßesPot] Wiederverwendung hat generell großes ökonomisches Potential aufgrund von Produktivitäts-steigerungen.	Widerlegt.	Neu
23	[Invest] Die Kosten der Herstellung wiederverwendbarer Software sollten als Investition betrachtet und bewertet werden.	Bestätigt.	Neu
24	[Varianz] Es gibt eine große Varianz in der erzielbaren Rendite.	Bestätigt.	(Indizien bestanden bereits)

Tabelle 6-14: Aktualisierter Status betroffener Hypothesen

6.5 Zusammenfassung

Wir haben in diesem Kapitel das ReValue-Modell zur ökonomischen Bewertung von Investitionen in wiederverwendbare Software vorgestellt. Es besteht aus einer neuartigen Bewertungsmethode, einem umfassenden und detaillierten Modell der Herstellung und Nutzung wiederverwendbarer Software und einem Algorithmus zur Monte-Carlo-Simulation. Aus den präsentierten Simulationsergebnissen für verschiedene exemplarische Szenarien ergibt sich, daß ökonomisches Potential und Investitionsunsicherheit in Abhängigkeit von den Rahmenbedingungen stark schwanken.

Kapitel 7

Leitlinien für die Entwicklung wiederverwendbarer Software

Ziel dieses Kapitels ist, Leitlinien für die Entwicklung systematisch wiederverwendbarer Software als Synthese der bisher präsentierten Ergebnisse der Arbeit zu formulieren. Die Anwendung dieser Leitlinien soll den Erfolg der Entwicklung in softwaretechnischer und wirtschaftlicher Hinsicht bewirken.

Zunächst stellen wir die Rahmenbedingungen und den Status quo der Wiederverwendung dar, insbesondere in Form der empirisch überprüften Hypothesen. Dann präsentieren wir eine Verfeinerung des Wiederverwendbarkeitsbegriffs, die den Zusammenhang der Softwareentwicklung berücksichtigt und sich an der notwendigen Investition orientiert. Basierend auf einer systematischen Nutzenanalyse stellen wir anschließend strategische Leitlinien vor, die den Beitrag wiederverwendbarer Software zur Erreichung strategischer Ziele umfassen. Dann geben wir Leitlinien für die Bewertung von Projektvorhaben nach softwaretechnischen und ökonomischen Kriterien an. Schließlich formulieren wir softwaretechnische Leitlinien. Sie betreffen die softwaretechnisch und ökonomisch sinnvolle Realisierung von Variabilität und die Zerlegung des Systems in Komponenten.

Inhalt: **Seite**

7.1 Einleitung

Grundlage dieses Kapitels sind die Ergebnisse aus Kapitel 4, Kapitel 5 und Kapitel 6. Die Leitlinien bauen auf ihnen in mehrfacher Hinsicht auf. Zum einen gehen wir von den Rahmenbedingungen aus, die wir empirisch erhoben und in Simulationen mit dem ReValue-Modell untersucht haben und die wir zusammengefaßt im nächsten Abschnitt darstellen. Zum anderen greifen wir auf die Analysen und Schlußfolgerungen zurück, die wir synthetisieren und ergänzen.

Bezüglich der wissenschaftlichen Einordnung und des Beitrags der einzelnen Abschnitte dieses Kapitels verweisen wir auf die ausführliche Darstellung in 3.5.

7.2 Rahmenbedingungen und Status quo

Zunächst wollen wir die Annahmen zu den Rahmenbedingungen der Entwicklung wiederverwendbarer Software explizit darstellen, von denen wir bei der Erarbeitung der Hypothesen ausgehen. In den vorangegangenen Kapiteln haben wir eine Reihe von Annahmen empirisch erhärtet bzw. durch Simulationsergebnisse untermauert. Wir fassen diese in Form der überprüften Hypothesen aus 3.5.3 zusammen. Anschließend behandeln wir die Auswirkungen der Industriestruktur auf die Bereitschaft zur Investition in die Entwicklung wiederverwendbarer Software. Die Kenntnis der Rahmenbedingungen ist für die Leitlinien deshalb von großer Bedeutung, weil sie auf realistische Annahmen aufgebaut werden müssen, um für die industrielle Praxis relevant zu sein. Damit wollen wir der bestehenden Tendenz entgegenwirken, implizit auf nicht überprüften Annahmen aufzubauen (vgl. Kapitel 3).

Schließlich stellen wir die wesentlichen in dieser Arbeit gewonnenen Erkenntnisse zum Status quo zusammengefaßt dar.

7.2.1 Empirisch überprüfte Hypothesen

In Kapitel 4, Kapitel 5 und Kapitel 6 werden die in 3.5.3 formulierten Hypothesen überprüft. Die Ergebnisse sind eine wichtige Grundlage der Leitlinien. Wir fassen sie in Tabelle 7-1 nochmals zusammen, um die Annahmen explizit darzustellen, von denen wir bei der Erarbeitung der Leitlinien ausgehen. Gleichzeitig dokumentieren wir damit ein auch für sich genommen wichtiges Ergebnis dieser Arbeit.

Nr	Hypothese	Ergebnis	Kapitel
Hypothesen zum aktuellen Vorgehen			
1	[Code] Wiederverwendung findet heute überwiegend in der Programmierung statt.	• Bestätigt für unmittelbar konkret und mittelbar ins Endprodukt eingehende Artefakte. • Widerlegt für abstrahiert eingehende Artefakte (Spezifikation/Templates).	4.2.1
2	[AdHoc] Der Schwerpunkt liegt auf Ad-Hoc-Wiederverwendung (Entwickler selbst oder innerhalb der Gruppe).	Bestätigt.	4.2.1
3	[EigEntw] Eigene Entwicklung ist für die Entwickler die wichtigste Quelle für wiederverwendbare Software.	• Bestätigt für Entwickler mit einem Dienstalter von zweieinhalb Jahren und mehr. • Widerlegt für Entwickler mit einem geringeren Dienstalter.	4.2.1
4	[MangelSW] Es besteht ein Mangel an systematisch wiederverwendbarer Software.	Bestätigt.	4.2.4, 4.2.1
Hypothesen zur Einschätzung der Wiederverwendung			
5	[NegEinst] Die Entwickler sind grundsätzlich negativ zur Wiederverwendung eingestellt.	Widerlegt.	4.2.2, 5.2
6	[ErhProd] Die Erhöhung der Entwicklungsproduktivität ist der wichtigste Vorteil von Wiederverwendung.	Widerlegt.	4.2.2
7	[VerkZeit] Verkürzung der Entwicklungszeit ist ein wichtiger Vorteil der Wiederverwendung.	Widerlegt.	4.2.2
8	[SenkWart] Senkung der Wartungslast ist ein wichtiger Vorteil der Wiederverwendung.	Bestätigt.	4.2.2

9	[QualiErh] Erhöhung der Qualität ist ein wichtiger Vorteil der Wiederverwendung.	Bestätigt.	4.2.2
10	[GrundsProb] Wiederverwendung ist grundsätzlich möglich; es gibt keine inhärenten, grundsätzlich nicht lösbaren Probleme (z.B. schlechtere Performance, suboptimale Lösungen).	Bestätigt.	4.2.2, 4.2.3, 5.2
11	[ZusatzAuf] Wichtigster Nachteil ist der Zusatzaufwand für die Herstellung wiederverwendbarer Software.	Bestätigt.	4.2.2
Hypothesen zu Hindernissen und notwendigen Maßnahmen			
12	[Ress] Es werden keine ausreichenden Ressourcen für die Herstellung wiederverwendbarer Software zur Verfügung gestellt.	Bestätigt.	4.2.3
13	[NIH] Das Not-invented-here-Syndrom spielt eine große Rolle.	Widerlegt.	4.2.3, 5.2
14	[Abschreck] Der Zusatzaufwand für Dokumentation und Wartung schreckt von der Entwicklung wiederverwendbarer Software ab (gegenläufiger Anreiz).	Bestätigt.	4.2.3
15	[InfoTool] Mangelnde Information und mangelnde Tool-Unterstützung sind wichtige Hindernisse.	• Bestätigt für Information. • Widerlegt für Tools.	4.2.3
16	[Prozess] Wiederverwendung ist momentan im Prozess nicht ausreichend verankert.	Bestätigt.	4.2.4
17	[Archi] Die Architektur hat großen Einfluß auf den Erfolg der Wiederverwendung.	Bestätigt.	5.2

18	[OO] Objektorientierte Entwicklung ist keine Voraussetzung für den Erfolg.	Bestätigt.	4.2.3, 5.2
19	[HindAbb] Abbau von Hindernissen ist wichtiger als die Schaffung positiver Anreize.	Bestätigt.	4.2.4
20	[FinAnr] Finanzielle Anreize haben hohe Bedeutung.	Widerlegt.	4.2.4
21	[ZentrOrg] Die Unterstützung der Wiederverwendung durch eine zentrale Organisationseinheit ist notwendig.	• Bestätigt für Sammeln & Screenen, Verwaltung der Library, Verwaltung der Reuse-Prozesse. • Widerlegt für Tätigkeiten der Entwicklung (Entwicklung, Generalisierung, Dokumentation, Erstellung von Testfällen).	4.2.4
Hypothesen zu ökonomischen Aspekten			
22	[GroßesPot] Wiederverwendung hat generell großes ökonomisches Potential aufgrund von Produktivitätssteigerungen.	Widerlegt.	6.3, 6.4
23	[Invest] Die Kosten der Herstellung wiederverwendbarer Software sollten als Investition betrachtet und bewertet werden.	Bestätigt.	6.3, 6.4 4.2.3
24	[Varianz] Es gibt eine große Varianz in der erzielbaren Rendite.	Bestätigt.	6.3, 6.4 4.2.5

Tabelle 7-1: Auswertung der zu überprüfenden Hypothesen

7.2.2 Auswirkungen der Industriestruktur

Um zu bestimmen, wie sich die grundsätzliche Struktur der Softwareindustrie auf die Rahmenbedingungen für Wiederverwendung auswirkt, führen wir eine Branchen-Strukturanalyse nach Porter (vgl. [Por99], 33ff) durch. Aus ihr ergibt sich, daß in der Software-Industrie zwei der fünf strukturellen Determinanten der Wettbewerbsintensität eine wesentliche Rolle spielen: Rivalität unter den bestehenden Unternehmen und potentielle neue Konkurrenten. Wir erläutern zunächst, wie wir zu diesem Ergebnis kommen und untersuchen dann, was hinsichtlich der Wiederverwendung daraus folgt.

Die Rivalität zwischen den bestehenden Unternehmen ist besonders hart, weil Differenzierungsvorteile aufgrund der kurzen Innovationszyklen (vgl. 2.2.2) nur von kurzer Dauer sind: Ständig muß neue Expertise aufgebaut und – im Fall von Produktunternehmen – neue Produkte entwickelt werden. Die Kosten für diese dauernde Innovation sind hoch.

Bei Produktunternehmen kommt verschärfend hinzu, daß das sog. *Law of increasing returns* gilt: Der Marktführer hat allein aufgrund seiner Marktposition bessere Absatzchancen als seine Konkurrenten, weil ihm mehr Vertrauen entgegengebracht wird und das auf sein Produkt abgestimmte Angebot von Drittherstellern größer ist (vgl. [Hoc99], 41). Wer nicht Marktführer in seinem jeweiligen Segment ist, hat deshalb deutliche Nachteile, was zu einem besonders harten Kampf um Marktanteile führt[76], der speziell für kleinere Spieler ein Kampf um das Überleben sein kann. Der Geschwindigkeit, mit der der Markt besetzt wird, d.h. neue Produkte entwickelt und auf den Markt gebracht werden – der sog. time to market – kommt daher vorrangige Bedeutung für den Unternehmenserfolg zu.

Zusätzlich drängen aufgrund der niedrigen Eintrittsbarrieren in der Softwareindurstrie (vgl. 2.2.2) ständig neue Konkurrenten auf den Markt. Um diesem Wettbewerb standzuhalten, müssen die bestehenden Unternehmen schnell und flexibel auf deren Produktangebot reagieren, was die Anforderungen an die Innovationsfähigkeit erhöht.

Neben der Wettbewerbsintensität spielen auch die Anforderungen des Kapitalmarkts eine Rolle. Die Erwartungen an Profitabilität und Wachstum von Softwareunternehmen sind überdurchschnittlich (vgl. [FSS99], 2.2.2) und führen zu einer gewissen Kurzatmigkeit in der Investitionspolitik. Angestrebt wird der kurzfristige Erfolg, was mittel- bis langfristige Investitionen unattraktiv erscheinen läßt.

Insgesamt ergibt sich aus diesen Bedingungen ein kurzer Zeithorizont für die Software-Industrie. Er ist generell charakteristisch für junge Branchen (vgl. [Por99], 283f) und insofern nicht außergewöhnlich, führt jedoch zu für systematische Wiederverwendung ungünstigen Rahmenbedingungen. Diese treten in drei konkreten Ausprägungen auf:

- Systematische Wiederverwendbarmachung von Software erfordert Zeit (vgl. 2.1.2). Bei der Nutzung wiederverwendbarer Software kann zwar eine Verkürzung der Entwicklungszeit erzielt werden, die Herstellung wiederverwendbarer Software erfordert jedoch zusätzliche Entwicklungszeit (siehe auch 4.2.3). Die große Bedeutung der time to market für den Markterfolg verhindert daher häufig die Entwicklung wiederverwendbarer Software.
- Systematische Wiederverwendung setzt zumindest mittelfristig kontinuierliche Planung voraus. Dies steht im Widerspruch zu der Anforderung, möglichst flexibel auf die Angebote der Konkurrenten reagieren zu können.

[76] Das kann so weit gehen, daß Unternehmen versuchen, ihre Konkurrenten zu eliminieren wie beispielsweise im Fall Microsoft ([MiQ96],

- Die Kosten der systematischen Wiederverwendbarmachung sind als Investition zu sehen, die sich in der Regel kurzfristig nicht rentieren. Demgegenüber erwartet der Kapitalmarkt stetige hohe Gewinne.

In Tabelle 7-2 fassen wir den hier beschriebenen grundlegenden Konflikt zwischen den Anforderungen systematischer Wiederverwendung und den Erfolgsvoraussetzungen in der Softwareindustrie in einem Überblick zusammen.

Anforderungen systematischer Wiederverwendung	Voraussetzungen für den Erfolg in der Softwareindustrie
Herstellung von Wiederverwendbarkeit erfordert Zeit	Time to market ist entscheidend
Kontinuierliche Planung notwendig	Flexible Reaktion auf Angebote der Konkurrenz notwendig
Investition in Wiederverwendbarkeit in der Regel kurzfristig unprofitabel	Stetige hohe Gewinne gefordert

Tabelle 7-2: Anforderungen systematischer Wiederverwendung und Erfolgsvoraussetzungen in der Softwareindustrie

7.2.3 Zusammenfassende Darstellung des Status quo

Die wesentlichen in dieser Arbeit gewonnenen Erkenntnisse zum Status quo der Wiederverwendung in der industriellen Softwareentwicklung fassen wir im folgenden kurz zusammen.

Generelle Einschätzung des Potentials für Wiederverwendung

Die allgemein vorherrschende, nahezu uneingeschränkt positive Einschätzung der Wiederverwendung (vgl. Kapitel 3) wird durch unsere Untersuchungen in Frage gestellt und weicht einer differenzierten Sichtweise. Entgegen der weitverbreiteten Überzeugung (vgl. 3.5.3) ist die Entwicklung wiederverwendbarer Software ökonomisch nicht in allen Fällen vorteilhaft. Die entsprechende Investition ist mit – zum Teil sehr hoher – Unsicherheit behaftet und erzielt Renditen, die eine Bandbreite zwischen sehr hohen Verlusten und sehr hohen Gewinnen einnehmen.

Daneben birgt sie auch ein erhebliches softwaretechnisches Risiko bis hin zur Gefährdung des ursprünglichen Entwicklungsprojekts bei A-Priori-Wiederverwendbarmachung. Insofern kann es durchaus sinnvoll sein, auf die Wiederverwendbarmachung von Software zu verzichten, auch wenn dies zur Folge hat, daß mehrfach für ähnliche Anforderungen neu entwickelt wird.

Hindernisse

Hauptsächliches Hindernis für Wiederverwendung ist die mangelnde Bereitschaft, Ressourcen in Form von Zeit und Personal für die Wiederverwendbarmachung bereitzustellen, d.h. in finanzieller Sicht eine Investition zu tätigen (vgl. 4.3.3). Wie wir in 7.2.2 ausführen, ist dies eine direkte Folge der

Industriestruktur und nicht nur auf eine verfehlte Einschätzung durch die Verantwortlichen zurückzuführen. Ein Mangel an Ressourcen hat den zusätzlichen negativen Effekt, daß kein Personal bereitgestellt werden kann, um Entwickler wiederverwendbarer Software zu unterstützen. Bedarf dafür besteht in der Regel bei den Maßnahmen, die der Nutzbarkeit für Entwickler dienen (z.B. Erstellung von Beispielanwendungen, Code Reviews, Dokumentation, Erstellung von Testfällen) und bei der Wartung. Diese Tätigkeiten werden häufig als abschreckend empfunden. Eine Unterstützung ist insofern besonders wichtig, damit der abschreckende Effekt vermieden wird, zumal für die Lösung der schwierigen softwaretechnischen Probleme bei der Entwicklung wiederverwendbarer Software gezielt die besten Entwickler eingesetzt werden sollten, ohne daß sich diese jedoch automatisch – und womöglich gegen ihren Willen – längerfristig für Wartungsaufgaben verpflichten müssen.

Bereitstellung von Ressourcen ist lediglich eine notwendige, nicht aber eine hinreichende Bedingung für den Erfolg. Zentrale Bedeutung hat die Architektur und hier insbesondere die Trennung der Zuständigkeiten. Während Kriterien dafür angegeben werden können, was eine gute Architektur im Allgemeinen auszeichnet (beispielsweise die Verwirklichung der Trennung der Zuständigkeiten), ist es nur in begrenztem Maß möglich, die Methode zu formulieren, die dorthin führt, ähnlich wie der Weg zu einem eleganten Beweis nicht formalisierbar ist. Menschliche Kreativität und Erfahrung sind in beiden Fällen nicht zu ersetzen. Insofern ist die Verpflichtung eines guten Chefdesigners indirekt ein entscheidender Einflußfaktor für die Qualität der Architektur (vgl. [Den91], 4).

Unterschiede nach Software-Kategorien

Die nach Software-Kategorien (vgl. 2.3.6) getrennte Untersuchung der Erfolgsaussichten für Wiederverwendung in softwaretechnischer (vgl. Kapitel 5) und ökonomischer (vgl. Kapitel 6) Hinsicht hat große Unterschiede zutage gefördert:

- Null-Software hat sowohl technisch als auch ökonomisch das höchste Erfolgspotential. Allerdings ist davon auszugehen, daß der Bedarf teilweise bereits durch kommerziell verfügbare Bibliotheken wie JGL für Java oder STL für C/C++ abgedeckt ist. Zudem ist der Anteil an Null-Software an der gesamten Softwareentwicklung nicht beliebig erweiterbar. Es ist aber davon auszugehen, daß die Möglichkeiten zur Maximierung dieses Anteils durch konsequente Trennung der Zuständigkeiten bei weitem noch nicht ausgeschöpft sind.
- T-Software hat ökonomisch großes Renditepotential, gleichzeitig ist jedoch die Investitionsunsicherheit hoch, so daß diese Kategorie insgesamt etwas schlechter dasteht als Null-Software. Die gute Standardisierbarkeit ist ein softwaretechnischer Vorteil bei der Wiederverwendbarmachung.
- A-Software schließlich fällt ökonomisch stark ab, auch softwaretechnisch ist die Wiederverwendbarmachung durch die mangelnde Standardisierbarkeit deutlich erschwert. Die hohen Erwartungen an wiederverwendbare Anwendungskomponenten (vgl. z.B. [HeS99]) erscheinen vor diesem Hintergrund ungerechtfertigt.

Nachteile

Die Entwicklung systematisch wiederverwendbarer Software birgt eine Reihe von Nachteilen, die in der Literatur bisher fast vollständig ignoriert wurden (vgl. Kapitel 3). Die wichtigsten sind:

- Die Entwicklung systematisch wiederverwendbarer Software ist schwieriger als die Entwicklung vergleichbarer nicht systematisch wiederverwendbarer Software. Daher muß die Durchführbarkeit eines geplanten Projekts zur Wiederverwendbarmachung sorgfältig geprüft werden, um das Risiko des Scheiterns zu minimieren. Bei A-Priori-Wiederverwendbarmachung fällt dieses Projekt mit dem ursprünglichen Entwicklungsprojekt zusammen. Insofern ist besondere Vorsicht geboten, zumal die Gefahr des Over-Engineerings in diesem Fall besonders groß ist.
- Durch die Entwicklung systematisch wiederverwendbarer Software wird zunächst die Geschwindigkeit der Entwicklung verringert; die time to market verzögert sich, was – speziell, aber nicht nur im Falle von Produktfirmen – ein gewichtiges Gegenargument sein kann.
- Investitionen in die Entwicklung systematisch wiederverwendbarer Software sind mit – teilweise sehr hoher – Unsicherheit behaftet, was sie angesichts der hohen Gewinnerwartungen von Seiten des Kapitalmarkts unattraktiv erscheinen läßt, zumal die unvermeidliche operative Unsicherheit (z.B. durch Fehleinschätzungen des Marktes oder Stornierungen von Kundenaufträgen) erheblich ist.

7.3 Verfeinerter Wiederverwendbarkeitsbegriff

In den bestehenden Veröffentlichungen wird eine differenzierte Betrachtung dessen, was Wiederverwendbarkeit ausmacht, oft vernachlässigt; außerdem werden Wechselwirkungen mit allgemeinen Entwicklungszielen nicht ausreichend untersucht (vgl. 3.5).

Daher verfeinern wir in diesem Abschnitt vor dem Hintergrund der Ergebnisse in den vorangegangenen Kapiteln den Wiederverwendbarkeitsbegriff und leisten so einen wissenschaftlichen Beitrag. Dazu analysieren wir Wiederverwendbarkeit zunächst getrennt nach Einzelmerkmalen und aus der Sicht der Investition. Dann gehen wir auf die Wechselwirkungen mit allgemeinen Entwicklungszielen ein. Schließlich führen wir eine systematische Analyse des durch Wiederverwendbarkeit erzielbaren Nutzens durch.

7.3.1 Das Wiederverwendbarkeitskontinuum

Wie in 2.1.2 ausgeführt, setzt sich Wiederverwendbarkeit aus drei Merkmalen zusammen: Nutzbarkeit für Entwickler, Adaptierbarkeit und Portierbarkeit, wobei die beiden letzteren unter dem Begriff Variabilität zusammengefaßt werden können. Nutzbarkeit für Entwickler faßt Verständlichkeit, Änderungseffizienz und Verfügbarkeit zu einem Begriff zusammen.

Die Merkmale der Wiederverwendbarkeit können von Komponente zu Komponente jeweils unterschiedlich ausgeprägt sein. Wenn wir sie uns zu den zwei Merkmalen Variabilität und Nutzbarkeit für Entwickler zusammengefaßt denken, können wir uns ein Kontinuum mit diesen zwei Dimensionen vorstellen, in dem jede Komponente einen Ort einnimmt, der dem Maß entspricht, in dem diese beiden Merkmale ausgeprägt sind. Eine beispielhafte Darstellung dieses *Wiederverwendbarkeitskontinuums* gibt Abbildung 7-1. Je stärker die Variabilität einer Komponente ausgeprägt ist, desto weiter rechts befindet sie sich im Koordinatensystem; analog dazu bedeutet höhere Nutzbarkeit für Entwickler eine Position weiter oben.

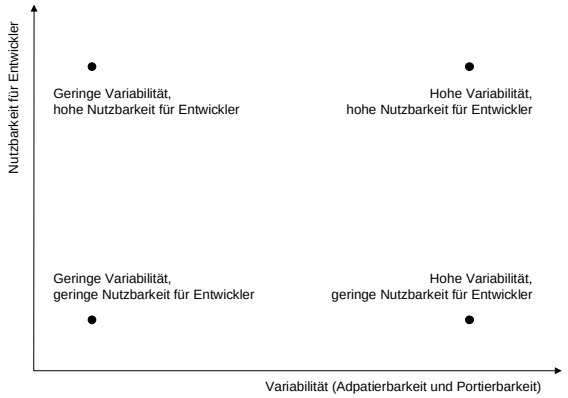

Abbildung 7-1: Wiederverwendbarkeitskontinuum mit Positionsbeispielen

Wir betrachten nun, wie sich Maßnahmen zur Steigerung der Wiederverwendbarkeit (vgl. 2.1.2) auf die Kosten der Entwicklung wiederverwendbarer Komponenten auswirken und welches Ausmaß an Expertise im Software-Entwurf notwendig ist. Dabei stellen wir fest, daß sich die beiden Dimensionen wesentlich im Ausmaß unterscheiden, in dem die Kosten aufgrund der Maßnahmen ansteigen, was durch die unterschiedlichen Pfeile in Abbildung 7-2 illustriert ist: während die Maßnahmen zur Steigerung der Variabilität einen geringen Kostenanstieg verursachen, ist der Kostenanstieg durch Maßnahmen zur Steigerung der Nutzbarkeit für Entwickler groß. Dasselbe gilt mit vertauschten Dimensionen für die geforderte Software-Entwurfs-Expertise, wie in Abbildung 7-3 gezeigt.

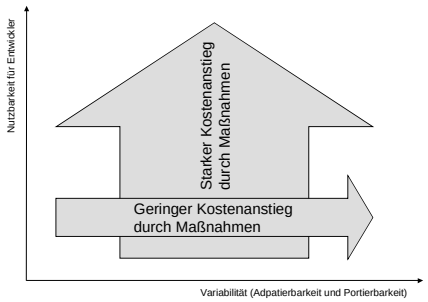

Abbildung 7-2: Kostenanstieg durch Maßnahmen

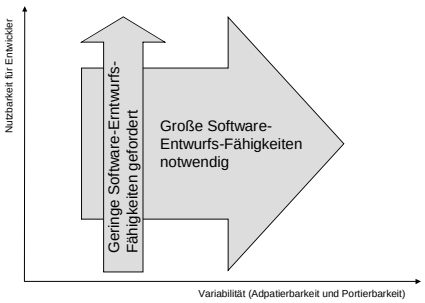

Abbildung 7-3: Geforderte Fähigkeiten im Software-Entwurf

Die Begründung für den Unterschied im Kostenanstieg ist wie folgt: Variabilität wird zunächst durch eine geeignete Architektur erreicht, die durch Zerlegung in entkoppelte Komponenten und Trennung der Zuständigkeiten das Geheimnisprinzip verwirklicht. Nebenbei wird so der Austausch von Schnittstellenimplementierungen ermöglicht, der einen sehr wichtigen Variabilitätsmechanismus darstellt. Dafür fällt grundsätzlich kein höherer Aufwand an als für eine Architektur, die diese Kritereien nicht erfüllt, sondern es wird lediglich anders vorgegangen. Das dadurch – ohne Mehrkosten – erreichbare Maß an Variabilität bezeichnen wir als *inhärente Variabilität der Architektur*. Darüber hinaus wird Variabilität durch den Einsatz weiterer Variabilitätsmechanismen (u.a. Parametrierung, Vererbung, Generierung; vgl. 2.3.5) erreicht, die in der Regel geringe Zusatzkosten verursachen. Eine Erhöhung der Nutzbarkeit für Entwickler hingegen erfordert substantiellen zusätzlichen Aufwand, sei es für zusätzliche Dokumentation (z. B. ein Tutorial), Bereitstellung von Beispielanwendungen oder Unterstützung der nutzenden Entwickler, beispielsweise durch eine Hotline.

Wenn der Kostenanstieg für eine Steigerung der Variabilität auch gering ist, ist es trotzdem keinesfalls einfach, sie zu realisieren. Vielmehr sind hierfür große Fähigkeiten im Software-Entwurf notwendig, die Urteilsvermögen und Kreativität einschließen. Der dafür entscheidende Erfolgsfaktor sind kompetente und erfahrene Software-Architekten; wir gehen darauf in 7.6 näher ein. Die Hauptschwierigkeit liegt darin, daß solche Software-Architekten knapp und überdies äußerst gefragt sind.[77] Die Steigerung der Nutzbarkeit für Entwickler verlangt hingegen lediglich ein Verständnis des Entwurfs, aber keine aktive Eigenleistung, so daß die entsprechenden Maßnahmen von Entwicklern mit durchschnittlicher Qualifikation durchgeführt werden können.

7.3.2 Wiederverwendbarkeit aus Sicht der Investition

Ein wichtiges Ergebnis unserer Untersuchungen ist, daß die Kosten der Wiederverwendbarmachung als Investition betrachtet und dementsprechend bewertet werden sollten (vgl. 7.2.1, [Invest]). In 2.1.6 schlagen wir daher eine Differenzierung des Begriffs der systematischen Wiederverwendbarkeit vor, wobei das Kriterium der Abgrenzung die Höhe der Investition in Wiederverwendbarkeit ist. Als Grenze definieren wir dabei eine Investition in Höhe der ursprünglichen Entwicklungskosten K_e. Liegt die Investition darunter, dann sprechen wir von Wiederverwendbarkeit im engeren Sinne, liegt sie darüber, dann handelt es sich um Mehrfachverwendbarkeit.

Im folgenden beschäftigen wir uns mit der Art der Investition, untersuchen also, für welche Maßnahmen der Investitionsaufwand im Einzelnen entsteht. Eine wesentliche Rolle bei der Betrachtung spielen dabei die in 7.3.1 beschriebenen unterschiedlichen Kostenanstiege, die durch verschiedene Maßnahmen zur Steigerung der Wiederverwendbarkeit hervorgerufen werden. In Abbildung 7-4 illustrieren wir das schematisch. Eingezeichnet ist dort die

[77] Da der Preis mit der Knappheit eines Gutes ansteigt, führt dies zu entsprechend hohen Gehältern. Diese können dazu beitragen, daß höhere Variabilität höhere Kosten verursacht.

Kurve konstanter Investition für einen Faktor der Investition in Wiederverwendbarkeit i_{wv} in Höhe von i_{wv}= 1, die die Trennlinie zwischen Wiederverwendbarkeit im engeren Sinne und Mehrfachverwendbarkeit darstellt (vgl. 2.1.6). Die Kurve hat folgende charakteristische Merkmale:

- Die mit derselben Investition erreichbare Steigerung der Variabilität ist deutlich größer als die der Nutzbarkeit für Entwickler, so daß der Schnittpunkt der Kurve mit der x-Achse (Variabilität) deutlich weiter außen liegt als der mit der y-Achse (Nutzbarkeit für Entwickler) – vergleichbare Maßstäbe vorausgesetzt.
- Die Steigerung der Variabilität aufgrund der realisierten inhärenten Variabilität der Architektur verursacht keine Mehrkosten, so daß die Investitionskurve in diesem Bereich flach verläuft.

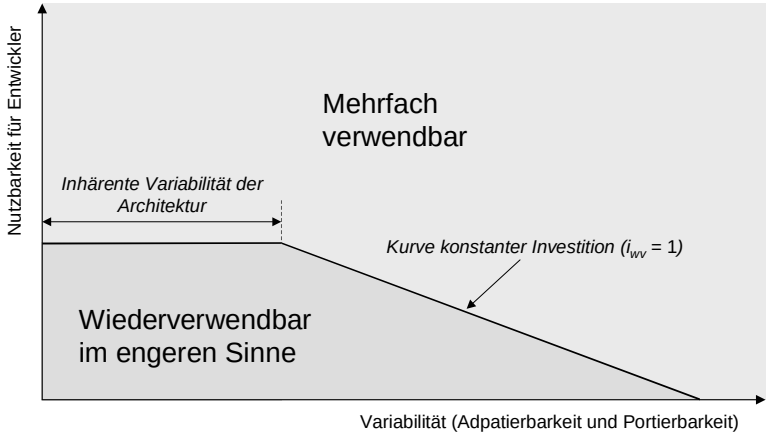

Abbildung 7-4: Varianten systematischer Wiederverwendbarkeit nach Art und Höhe der Investition (schematische Darstellung)

Eine Zusammenstellung wesentlicher Maßnahmen, die mit dem Ziel der Nutzbarkeit für Entwickler ergriffen werden, ist in Tabelle 7-3 zu sehen. Einige der Maßnahmen haben neben dem hauptsächlichen Effekt einen Nebeneffekt, beispielsweise dient die Bereitstellung von automatisierten Testfällen dazu, Änderungen effizient durchzuführen, indem der Testaufwand verringert wird. Daneben bewirkt sie aber in der Regel auch ein verbessertes Verständnis der jeweiligen wiederverwendbaren Komponente.

Auf die letzte der in Tabelle 7-3 angegebenen Maßnahmen wollen wir näher eingehen: Bereitstellung alternativer Implementierungen. Variabilität bedeutet die Bereitstellung einer Möglichkeit der Anpassung. Die Implementierung dieser Anpassung obliegt üblicherweise dem nutzenden Entwickler. Es besteht jedoch die Möglichkeit, von vornherein alternative Implementierungen für verschiedene, häufig auftretende Variationen der Anforderungen bereitzustellen, z.B. indem Treiber für unterschiedliche Systemumgebungen entwickelt werden. Dies bedeutet eine zusätzliche Investition, die nicht der Variabilität an sich, sondern der Erleichterung von Anpassungen dient, also der Änderungseffizienz und damit der Nutzbarkeit für Entwickler. Die damit verbundenen Kosten sind in der Regel erheblich. Zur Illustration wählen wir das Beispiel der Entkopplung einer Anwendung von der verwendeten Datenbank, z.B. durch Verwendung einer Datenbankzugriffsschicht wie des QDI (vgl. 5.2.1): dadurch

wird Variabilität im Sinne von Portierbarkeit bereitgestellt, d.h. es wird ermöglicht, die Anwendung an eine andere Datenbank anzupassen. Werden darüber hinaus Adapter (beim QDI 'API-Expert' genannt) für verschiedene Datenbanken implementiert und bereitgestellt, entspricht das alternativen Realisierungen.

Maßnahme	Verfügbarkeit	Verständlichkeit	Änderungs-effizienz
Einstellen in Bibliothek	**		
Publizität herstellen (Veröffentlichungen, Informationsveranstaltungen)	**	*	
Erstellen von Dokumentation		**	
Erstellen von Beispielen		**	
Unterstützung (Hotline)	*	**	
Schulungen	*	**	
Bereitstellung von Testfällen		*	**
Bereitstellung von Generatoren			**
Zentrale Wartung			**
Bereitstellung alternativer Implementierungen			**

**: hauptsächlicher Effekt, *: Nebeneffekt

Tabelle 7-3: Maßnahmen mit dem Ziel der Nutzbarkeit für Entwickler – Effekte im Detail

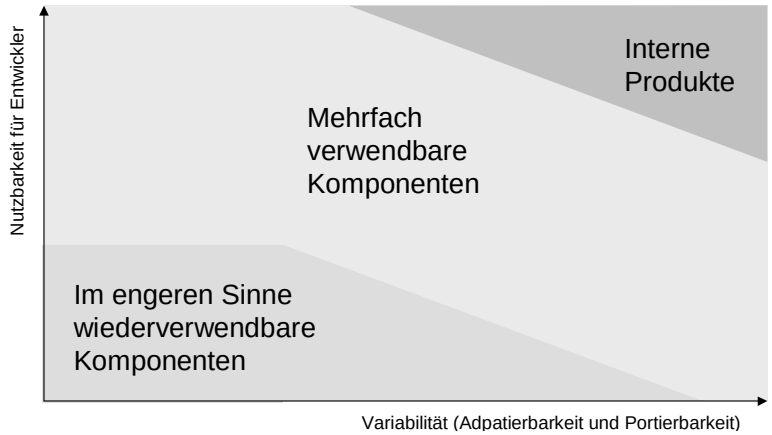

Abbildung 7-5: Internes Produkt im Wiederverwendbarkeitskontinuum (schematische Darstellung)

Überschreiten die Investitionskosten eine bestimmte Höhe, dann kann es sinnvoll erscheinen, von einem *internen Produkt* zu sprechen. Typische Kriterien für das Überschreiten der Grenze sind hohe Nutzungsraten in Verbindung mit professionellen Wartungs- und Supportleistungen, erheblichem internem Vermarktungsaufwand und der Bereitstellung alternativer Realisierungen. Eine schematische Darstellung der Position interner Produkte im Wiederverwendbarkeitskontinuum zeigt Abbildung 7-5.

Da die Kosten für interne Produkte besonders hoch sind und dementsprechend hohe Nutzungsraten erfordern, um die Wirtschaftlichkeit sicherzustellen, ist ihr Einsatzgebeit stark beschränkt: nach Erfahrung des Autors ist Potential dafür nur in seltenen Fällen bei großen Produktunternehmen zu erwarten.

7.3.3 Wechselwirkungen mit allgemeinen Zielen der Entwicklung

Die Ziele der Entwicklung können in der Regel auf drei grundlegende Zielgrößen zurückgeführt werden: Kosten und Zeit der Entwicklung sowie Qualität der entwickelten Software. Im folgenden untersuchen wir, wie Wiederverwendbarkeit das Erreichen dieser Ziele beeinflußt und welche Wechselwirkungen sich einstellen. Dabei greifen wir verschiedene Erkenntnisse auf, die wir im Verlauf der Arbeit gewonnen haben, und fassen sie systematisch zusammen. Die Ergebnisse dienen insbesondere als Grundlage für die in 7.4.1 beschriebene Analyse des Nutzens der Wiederverwendbarkeit.

Auswirkungen von Wiederverwendbarkeit auf Entwicklungskosten und Entwicklungszeit

Senkungen der Kosten und Verkürzungen der Entwicklungszeit ergeben sich beide aus reduziertem Aufwand (vgl. 2.2.3). Damit dieser sich in eine Verkürzung der Entwicklungszeit ummünzen läßt, müssen folgende Bedingungen erfüllt sein: die Größe des Entwicklungsteams darf nicht im selben Maß wie der Aufwand reduziert werden und die verbleibenden Aufgaben müssen ausreichend parallelisierbar sein.

Entscheidend ist dabei die Frage, wie sich Wiederverwendbarkeit auf den Aufwand auswirkt. Dabei ist zu unterscheiden zwischen kurzfristigen und mittel- bis langfristigen Auswirkungen. Dafür ist die Zeitverzögerung verantwortlich, mit der der Break-Even eintritt (vgl. 6.4.1): Kurzfristig steigen deshalb die Gesamtkosten der Entwicklung aufgrund der Investition in die Wiederverwendbarmachung zunächst an (auf den sich daraus ergebenden Konflikt mit den Erfolgsbedingungen in der Software-Industrie gehen wir in 7.2.2 und 7.2.3 ein). Im Erfolgsfall, d.h. wenn die Rendite dieser Investition ausreichend ist, sinken sie mittel- bis langfristig unter das Niveau ohne Wiederverwendung ab.

Entsprechend ergeben sich Vorteile hinsichtlich der Entwicklungskosten und Entwicklungszeit nur bei mittel- bis langfristiger Betrachtung.

Auswirkungen von Wiederverwendbarkeit auf die Qualität der entwickelten Software

Wenn wiederverwendbare Software genutzt wird, so hat das überwiegend – aber nicht nur – positive Auswirkungen auf die Qualität der entwickelten Software in den Nutzungsprojekten, zu denen bei A-Priori-Wiederverwendbarkeit auch das Herstellungsprojekt zählt. Die Qualität setzt sich nach unserer Definition (vgl. 2.1.2) aus folgenden Merkmalen zusammen: Nutzbarkeit für Entwickler, Adaptierbarkeit, Portierbarkeit, Spezifikationstreue der Funktionalität, Zuverlässigkeit, Benutzbarkeit für den Endbenutzer und Effizienz (zusammengesetzt aus Performance und Ressourcenverbrauch).

Da die Merkmale der Wiederverwendbarkeit Qualitätsmerkmale sind, steigt die Qualität der entwickelten Software dementsprechend entlang dieser Merkmale gemäß ihrer jeweiligen Ausprägung mit der Wiederverwendbarkeit der genutzten Komponenten. Wir sprechen von primären Auswirkungen, da sie sich direkt aus der Wiederverwendbarkeit ergeben. Zusätzlich können sich auch Zuverlässigkeit und Benutzbarkeit für den Endbenutzer verbessern (vgl. 2.2.3). Diese Auswirkungen nennen wir sekundär, da sie nur unter bestimmten Voraussetzungen eintreten. Im Fall der Zuverlässigkeit ist die Voraussetzung, daß ein Rückfluß der Information über entdeckte Fehler aus den Nutzungsprojekten sichergestellt ist und daß die Wartung zentral erfolgt, so daß die Zuverlässigkeit wiederverwendbarer Komponenten mit zunehmender Nutzung ansteigt, weil die Komponenten immer gründlicher getestet sind. Im Fall der Benutzbarkeit für den Endbenutzer eines bestimmten Systems ist die Voraussetzung, daß dieses System aufgrund mehrfachen Einsatzes bestimmter wiederverwendbarer Komponenten einheitlicher und dadurch leichter zu bedienen wird. Diese Komponenten können beispielsweise Elemente der Anwendungslogik oder der Oberfläche sein.

Die Nutzung wiederverwendbarer Software kann jedoch auch negative Auswirkungen auf die Qualität der entwickelten Software haben, denn Variabilität auf der einen Seite und Spezifikationstreue der Funktionalität sowie Effizienz auf der anderen Seite sind gegenläufige Qualitätsmerkmale. Wir behandeln zunächst die Spezifikationstreue: Steigt die Variabilität, dann sinkt tendenziell die Genauigkeit, mit der die funktionalen Anforderungen abgedeckt werden: Im Vergleich zu einer speziell für einen bestimmten Anwendungsfall entwickelten Komponente ist die durch eine wiederverwendbare Komponente bereitgestellte Lösung häufig aufwendiger als notwendig, was negative Folgen sowohl für Wartungsprogrammierer als auch für Endbenutzer nach sich ziehen kann. Die Ausprägung des Qualitätsmerkmals Spezifikationstreue der Funktionalität nimmt in diesen Fällen also ab. Analog ist es bei der Effizienz, für die entscheidend ist, daß die Lösung exakt auf die Anforderungen im jeweiligen Einzelfall abgestimmt ist, um einen optimalen Einsatz der technischen Ressourcen sicherzustellen.

Diese negativen Auswirkungen stellen sich nicht notwendigerweise ein, sind aber möglich. Da sie sich direkt aus der Wiederverwendbarkeit der Software ergeben, sind sie ebenfalls primärer Art im Sinne der o.g. Definition. Eine Übersicht über die Auswirkungen von Wiederverwendbarkeit auf die Qualität entwickelter Software insgesamt zeigt Abbildung 7-6. Sie kann dazu dienen,

im Einzelfall systematisch zu bestimmen, welche Auswirkungen Wiederverwendbarkeit auf bestimmte Qualitätsmerkmale hat, um diese in der Bewertung der Investition zu berücksichtigen.

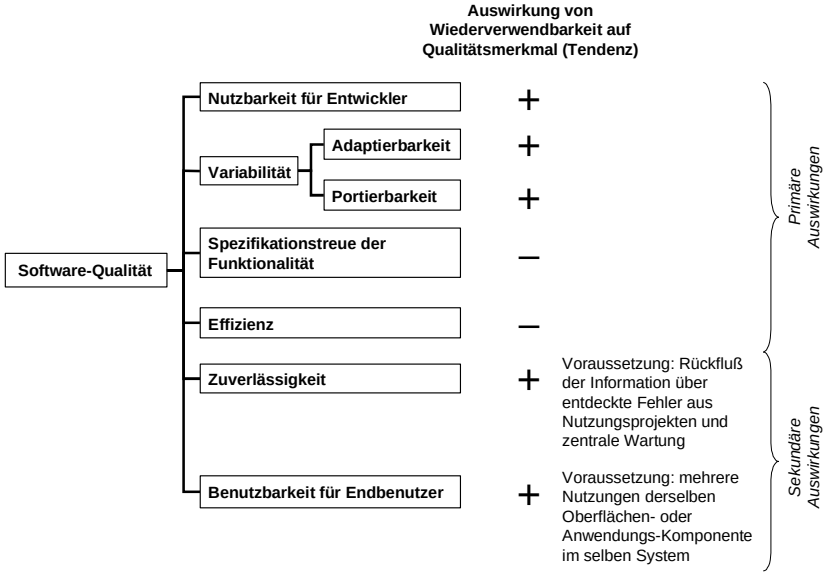

Abbildung 7-6: Auswirkung von Wiederverwendbarkeit auf die Qualität entwickelter Software

Wiederverwendbarkeit im Kontext der Änderbarkeit

Nutzbarkeit für Entwickler und Variabilität sind nicht nur bei Wiederverwendung von Bedeutung, sondern immer dann, wenn eine Komponente geändert werden soll. Das ist auch in folgenden Situationen der Fall:

- bei Wartung und Weiterentwicklung
- bei iterativem Vorgehen während der Entwicklung selbst
- beim Customizing eines Produkts

Speziell die Weiterentwicklung kann dabei als zeitlich versetzte Wiederverwendung im selben Kontext, aber unter geänderten Bedingungen, interpretiert werden.

Daraus folgt, daß durch die Wiederverwendbarmachung einer Komponente zusätzlicher Nutzen jenseits der Wiederverwendbarkeit entstehen kann. Dieser Nutzen muß bei einer umfassenden Bewertung der Investition berücksichtigt werden.

Ein umgekehrter Effekt kann sich bei der Nutzbarkeit für Entwickler einstellen: Wenn sie für eine bestimmte Komponente unzureichend ist, beispielsweise weil nicht ausreichend getestet oder dokumentiert wurde, so besteht darin eine besondere Gefahr, wenn diese Komponente a posteriori wiederverwendbar gemacht werden soll. Der Aufwand für den Ausgleich des Defizits wird in der Regel – unberechtigterweise – der Wiederverwendbarmachung zugerechnet, obwohl er eigentlich bei der ursprünglichen Entwicklung hätte erbracht werden müssen. Durch die Wiederverwendbarmachung erfolgt in diesem Fall eine Art von nachträglicher Quersubventionierung des ursprünglichen Entwicklungs-

projekts zum Vorteil aller Nutzer. Wenn die im Ergebnis erreichte Erhöhung der Nutzbarkeit für Entwickler auch aus Sicht des Unternehmens durchaus wünschenswert ist, so stellt der dafür anfallende Aufwand doch ein bedeutendes Hindernis für die Wiederverwendbarmachung dar, falls dieser Aspekt bei der Bewertung der Investition nicht ausreichend berücksichtigt wird.

7.4 Strategische Leitlinien

Ein dauerhafter Wettbewerbsvorteil gegenüber der Konkurrenz ist nicht durch operative Vorteile, sondern nur durch überlegene strategische Positionierung zu erreichen [Por96]. Daher analysieren wir in diesem Abschnitt zunächst, in Form welcher strategisch relevanter Verbesserungen sich der Nutzen auswirkt, der durch den Einsatz wiederverwendbarer Software entstehen kann. Dann befassen wir uns damit, wie strategisch relevante Verbesserungen dazu eingesetzt werden können, unterschiedliche strategische Ziele zu erreichen, wobei wir den Unterschied zwischen Produkt- und Projektgeschäft herausarbeiten.

Die Frage der strategischen Relevanz der Entwicklung wiederverwendbarer Software ist bisher unzureichend untersucht; unsere Ergebnisse tragen dazu bei, diese Lücke zu schließen (vgl. 3.5).

7.4.1 Systematische Nutzenanalyse

Die Grundlage der strategischen Leitlinien für die Entwicklung wiederverwendbarer Software ist eine systematische Analyse der Auswirkungen des Nutzens, der durch den Einsatz wiederverwendbarer Software entstehen kann. Das Ergebnis dieser Analyse beschreiben wir hier, wobei wir auf der in 2.2.3 (Abbildung 2-17) präsentierten Einteilung des Nutzens in drei grundsätzliche Arten aufbauen: reduzierter Aufwand für Entwicklung und Wartung, erhöhte Qualität der entwickelten Software und projektübergreifender Nutzen. Hier sei noch einmal darauf hingewiesen, daß der Nutzen erst durch die Nutzung wiederverwendbarer Software entsteht – Voraussetzung dafür ist die Entwicklung, die zunächst Aufwand verursacht. Zwischen Zeitpunkt der Investition und Wirksamkeit des Nutzens liegt also eine – je nach Intensität der Nutzung – unterschiedlich lange Verzögerungszeit.

In der Analyse unterscheiden wir zwischen operativ und strategisch relevanten Verbesserungen. Bei den operativ relevanten Verbesserungen konzentrieren wir uns auf diejenigen, die sich in Form einer Senkung der Gesamtentwicklungskosten E auswirken. Diese Senkung ergibt sich als Summe der Einsparungen über alle Nutzungsereignisse. Bei den strategisch relevanten Verbesserungen steht wegen ihrer besonderen Bedeutung (vgl. 7.2.2) die Markt- und Wettbewerbsposition im Zentrum. Sie wird von folgenden Größen beeinflußt: den Gesamtentwicklungskosten E, der time to market t_{tm}, der Kundenzufriedenheit Z und der Größe des erreichbaren Marktsegments M.

Auf die operativen Verbesserungen sind wir bereits in 2.2.3 eingegangen. Eine Senkung der Gesamtentwicklungskosten E entsteht vor allem aus dem gerin-

geren Anteil der Neuentwicklung. Eine zusätzliche Verringerung kann sich aus einer Erhöhung der Qualität in verschiedenen Ausprägungen ergeben: Verbesserte Nutzbarkeit für Entwickler und höhere Zuverlässigkeit helfen, die Nutzungskosten K_n zu senken, was die Einsparung pro Nutzungsereignis $\Delta K = K_e - K_n$ erhöht. Daneben kann sich eine verbesserte Variabilität in einer Steigerung der Nutzungsrate λ_0 auswirken, was wiederum E senkt. Der projektübergreifende Nutzen (Stärkung des Erfahrungsaustauschs der Entwickler und Vereinfachung von Rapid Prototyping) schließlich kann eine zusätzliche Senkung der Nutzungskosten K_n bewirken.

Die Markt- und Wettbewerbssituation wird durch eine Senkung der Gesamtentwicklungskosten E und durch eine Verkürzung der time to market t_{tm} entscheidend verbessert (vgl. 7.2.2); beide ergeben sich aus dem geringeren Anteil der Neuentwicklung. Auch eine Erhöhung der Nutzbarkeit für Entwickler kann sich in dieser Form auswirken. Die weiteren Qualitätsmerkmale wirken sich folgendermaßen aus: Gesteigerte Variabilität kann zu einer Vergrößerung des erreichbaren Marktsegments M führen, indem eine Erweiterung der Funktionalität möglich wird. Daneben führen Verbesserungen bei Zuverlässigkeit und Benutzbarkeit für den Endbenutzer in der Regel zu einer Erhöhung der Kundenzufriedenheit Z. Der projektübergreifende Nutzen in Form einer Vereinfachung des Rapid Prototyping kann ebenfalls zu einer Vergrößerung des erreichbaren Marktsegments beitragen.

Einen Überblick über die hier beschriebenen Effekte gibt Abbildung 7-7. Für die verschiedenen Arten des Nutzens ist jeweils dargestellt, welche Ausprägung jeweils operativ oder strategisch relevant ist. Die Pfeile hinter den Variablen zeigen an, ob sich eine Verbesserung für einen steigenden oder fallenden Wert ergibt. Wir erläutern dies an einem Beispiel: Die Nutzung wiederverwendbarer Software kann zu einem geringeren Anteil der Neuentwicklung führen. In der Ausprägung gesenkter Entwicklungskosten bedeutet dies eine Verbesserung sowohl der operativen Situation als auch der Markt- und Wettbewerbssituation des Unternehmens, in der Ausprägung einer verkürzten time to market bedeutet es eine Verbesserung der Markt- und Wettbewerbssituation.

Ausprägung des Nutzens / Relevanz der Verbesserung	Geringerer Anteil der Neuentwicklung	Höhere Qualität: Nutzbarkeit für Entwickler	Höhere Qualität: Variabilität	Höhere Qualität: Zuverlässigkeit	Höhere Qualität: Benutzbarkeit für Endbenutzer	Projektübergreifender Nutzen
Operativ (Kosten)	$E \downarrow$	$K_n \downarrow$	$\lambda_0 \uparrow$	$K_n \downarrow$	–	$K_n \downarrow$
Strategisch (Markt- und Wettbewerbsposition)	$E \downarrow$, $t_{tm} \downarrow$	$E \downarrow$, $t_{tm} \downarrow$	$M \uparrow$	$Z \uparrow$	$Z \uparrow$	$M \uparrow$

E: gesamte Entwicklungskosten
K_n: Nutzungskosten
λ_0: anfängliche Nutzungsrate p.a.
t_{tm}: time to market
M: Größe des erreichbaren Marktsegments
Z: Kundenzufriedenheit

Abbildung 7-7: Systematische Nutzenanalyse

7.4.2 Beitrag wiederverwendbarer Software zur Erreichung strategischer Ziele

Der Zweck strategischer Ziele ist die Erhaltung bestehender sowie die Suche und Schaffung neuer Erfolgspotentiale. Im Gegensatz dazu dienen operative Ziele dazu, den aktuellen Erfolg in Form von Gewinn und Wachstum sicherzustellen (vgl. [WeA01], 123). Verschiedene Ziele lassen sich in eine Zielhierarchie einordnen, an deren Spitze in der Regel als oberstes Ziel die langfristige Existenzsicherung steht ([WeA01], 117ff).

Wir untersuchen nun, welche Bedeutung die möglichen strategisch relevanten Verbesserungen für die Erreichung strategischer Ziele haben. Dazu tragen wir wesentliche strategische Ziele getrennt nach Produkt- und Projektgeschäft zusammen (vgl. [Stü99, Hoc99]; siehe auch 7.2.2, 7.2.3) und geben an, welche der in 7.4.1 angegebenen Verbesserungen für die verschiedenen Ziele jeweils relevant sind. Diese Analyse führen wir separat für Produkt- und Projektgeschäft durch. Es sei angemerkt, daß sie selbstverständlich nur hinsichtlich des Nutzens der Wiederverwendung vollständig ist. Viele weitere – sich nicht aus der Wiederverwendung ergebende – Einflußgrößen, die für die strategischen Ziele eine Rolle spielen, werden hier aufgrund unserer Fokussierung nicht betrachtet. Wir verwenden die oben eingeführte Schreibweise: t_{tm} steht für time to market, E für die Gesamtentwicklungskosten, M für die Größe des erreichbaren Marktsegments und Z für die Kundenzufriedenheit.

Produktgeschäft

Wichtige strategische Ziele im Produktgeschäft sind in Tabelle 7-4 dargestellt. Zu jedem Ziel geben wir an, welche der in 7.4.1 aufgeführten Ausprägungen des Nutzens für dieses Ziel hauptsächlich relevant ist.

Strategisches Ziel	**Hauptsächlich relevante Ausprägung des Nutzens der Wiederverwendung**
Marktführerschaft	t_{tm}
Innovationskraft	E, t_{tm}
Wachstum	M
Gutes Image im Markt	Z

Tabelle 7-4: Wichtige strategische Ziele im Produktgeschäft

Die Gewichtung der Ziele ist je nach Situation des Unternehmens unterschiedlich. Sie sind außerdem nicht entkoppelt, sondern hängen in vielfältiger Weise zusammen. Im folgenden erläutern wir das näher.

- Wer das zentrale Ziel der *Marktführerschaft* erreichen will, muß in der Lage sein, den Markt schnell zu besetzen (vgl. 7.2.2, [Stü99, Hoc99]). Dabei spielt die time to market t_{tm} die zentrale Rolle. Für den Erfolg muß sie soweit wie möglich abgesenkt werden, um die Konkurrenz zu schlagen.
- *Innovationskraft* erfordert Investitionen. Eine Senkung der Gesamtentwicklungskosten E setzt dafür notwendige Mittel frei und steigert so die Inno-

vationskraft. Daneben ist die Geschwindigkeit der Entwicklung wichtig, so daß auch die time to market t_{tm} eine Rolle spielt. Innovationskraft ist eine unabdingbare Voraussetzung dauerhafter Marktführerschaft, weil nur durch sie den Innovationen der Konkurrenz Paroli geboten werden kann. Zudem verbessert sie die Bedingungen für das Wachstum, weil Innovationen die Möglichkeit eröffnen, neue Marktsegmente zu besetzen.

- *Wachstum* ist auf zwei sich ergänzende Arten möglich: durch verbesserte Penetration des angestammten Marktsegments und durch Vergrößerung des erreichbaren Marktsegments *M*. Wachstum wird in der Regel von Investoren erwartet und sichert so die für die Innovationskraft entscheidende Kapitalversorgung. Daneben ist Wachstum auf dem Weg zum Erreichen der Marktführerschaft unabdingbar.
- Die Grundlage des Vertriebserfolgs sowohl im Sinne der Bestandssicherung als auch im Sinne des Wachstums ist ein *gutes Image im Markt*. Entscheidende Bedeutung dafür hat die Kundenzufriedenheit *Z*, denn Kaufinteressenten für ein Produkt messen den Erfahrungen große Bedeutung zu, die bestehende Kunden damit gemacht haben. Viele Produktanbieter veröffentlichen daher Listen von Referenzkunden. Dabei ist zu beachten, daß dies speziell für betriebliche Informationssysteme gilt, die einzeln an Unternehmen vertrieben werden und nicht als Massenware an Konsumenten. In letzterem Fall spielt die Kundenzufriedenheit eine weniger wichtige Rolle.

Projektgeschäft

In Tabelle 7-5 sind wichtige strategische Ziele im Projektgeschäft dargestellt. Auch hier geben wir zu jedem Ziel an, welche der in 7.4.1 aufgeführten Ausprägungen des Nutzens für dieses Ziel hauptsächlich relevant sind.

Strategisches Ziel	**Hauptsächlich relevante Ausprägung des Nutzens der Wiederverwendung**
Kontinuierlich hohe Auslastung	*Z, E*
Langfristige Kundenbeziehungen	*Z*
Innovationskraft	*E*, t_{tm}
Wachstum	*M*

Tabelle 7-5: Strategische Ziele im Projektgeschäft

- Zentrales strategisches Ziel im Projektgeschäft ist eine *kontinuierlich hohe Auslastung* (vgl. [Stü99, Hoc99]). Voraussetzung dafür ist ein ausreichender Auftragseingang, der auf erfolgreichen Akquisitionen beruht. Die Kundenzufriedenheit *Z* spielt dafür eine wichtige Rolle, sei es, weil bei einem bestehenden Kunden ein Nachfolgeprojekt akquiriert werden soll, sei es, weil die Referenzkundenliste eine wichtige Rolle für neue Kunden spielt. Daneben wird Wettbewerb immer zu einem gewissen Grad über den Preis ausgetragen. Für die Fähigkeit, wettbewerbsfähige Preise bei ausreichen-

der Marge anzubieten, spielen die Entwicklungskosten *E* eine entscheidende Rolle.

- Die *Langfristigkeit der Kundenbeziehungen* ist für den Akquisitionserfolg bei bestehenden Kunden essentiell. Sie dient somit dem Ziel der kontinuierlich hohen Auslastung, zumal Erstakquisitionen bei neuen Kunden in der Regel erheblich aufwendiger sind als die Akquisition von Folgeaufträgen. Entscheidende Einflußgröße ist die Kundenzufriedenheit *Z*. Ein Nebenziel ist auch hier das gute Image im Markt, aus denselben Gründen wie im Produktgeschäft. Da der Kunde das entwickelte System im Projektgeschäft allerdings erst im Nachhinein beurteilen kann, spielt das Vertrauen in die Leistungsfähigkeit des Anbieters eine wesentlich größere Rolle.
- Auch im Projektgeschäft erfordert *Innovationskraft* wie im Produktgeschäft Investitionen und Geschwindigkeit bei der Entwicklung, so daß die Gesamtentwicklungskosten *E* und die time to market t_{tm} eine Rolle spielen. Dies dient jedoch nicht dem vorgelagerten Ziel der Marktführerschaft, sondern der kontinuierlich hohen Auslastung: Zum einen erwarten die Kunden von einem Projektanbieter, daß er technisch auf der Höhe der Zeit ist. Zum anderen spielen Rapid Prototyping-Fähigkeiten bei der Akquisition regelmäßig eine Rolle.
- *Wachstum* hat im Projektgeschäft eine geringere Bedeutung als im Produktgeschäft. Doch auch hier erwarten Investoren in der Regel ein gewisses Wachstum, das somit – ähnlich wie im Produktgeschäft –Kapitalversorgung und somit auch die Innovationskraft sichert. Außerdem wirkt sich die durch Wachstum erreichbare Größe in zweierlei Hinsicht positiv aus: das Risiko eines Scheiterns bei der Akquisition wird breiter gestreut und es können größere Projekte durchgeführt werden, was ein wichtiges Kriterium für Kunden sein kann. In beiden Fällen dient das Wachstum dem Ziel der kontinuierlich hohen Auslastung. Die wesentliche Einflußgröße ist hierbei die Größe des erreichbaren Marktsegments *M*.

7.5 Leitlinien für die Bewertung von Projektvorhaben

Die Entwicklung wiederverwendbarer Software ist mit einer Reihe von Nachteilen und softwaretechnischen Risiken verbunden; darüber hinaus ist die ökonomische Rentabilität nicht sichergestellt (vgl. 7.2.3). Bevor mit der Entwicklung wiederverwendbarer Software begonnen wird, sollte daher eine Bewertung des Vorhabens nach softwaretechnischen und ökonomischen Kriterien durchgeführt werden, auf deren Basis über die Durchführung entschieden wird. Erreicht werden soll dadurch, daß nur Projekte durchgeführt werden, die sowohl softwaretechnisch als auch ökonomisch ein ausreichendes Erfolgspotential haben. Aufbauend auf den Ergebnissen aus Kapitel 4 bis Kapitel 6 geben wir im folgenden Leitlinien für diese Bewertung an, die das diesbezüglich bestehende Defizit in der Literatur mindern (vgl. 3.5). Dabei behandeln wir das Problem nicht erschöpfend, sondern konzentrieren uns auf die Spezifika von Projektvorhaben, in denen systematisch wiederverwendbare Software entwickelt werden soll.

Die Leitlinien bestehen in einem umfassenden Kriterienrahmen, der für die Bewertung von Projektvorhaben ergänzend zu den generell geltenden Kriterien herangezogen werden kann. Da die Bewertung für viele der Kriterien nicht formalisierbar ist, z.B. in Form einfacher Metriken, beschränken wir uns in diesen Fällen darauf, mögliche Ziele und Wege zu ihrer Erreichung anzugeben. Dabei gehen wir insbesondere auf mögliche Zielkonflikte ein.

7.5.1 Softwaretechnische Kriterien

Zunächst muß geprüft werden, ob das Projektvorhaben aus softwaretechnischer Sicht durchführbar ist (vgl. 5.3.3). Dabei geht es vor allem darum, Vorhaben auszusondern, bei denen das Risiko des Scheiterns zu groß ist. Wir geben zunächst allgemeine Kriterien an und gehen anschließend auf besondere Kriterien für den Fall der A-Priori-Wiederverwendbarkeit ein.

Allgemeine Kriterien

Die allgemeinen Kriterien beziehen sich auf drei Bereiche: Spezifikation, Architektur und Projektmanagement.

Spezifikation. Bei der Entwicklung wiederverwendbarer Software ist es besonders schwierig, die spezifizierten Anforderungen klar zu begrenzen, da Variabilität diesem Ziel grundsätzlich entgegensteht, indem sie eine Erweiterung des Spezifikationsumfangs bedeutet. Insofern ist die Versuchung besonders groß, auf eine Begrenzung zu verzichten. Die Begrenzung ist jedoch unabdingbar und daher sicherzustellen. Dies ist umso schwieriger, je geringer die Bereitschaft dazu ist, eine Standardisierung der Anforderungen zu akzeptieren. Diese Bereitschaft ist bei Anwendungsfunktionalität am geringsten[78], so daß die Begrenzung umso leichter wird, je geringer die Anforderungen von der Anwendungsfunktionalität beeinflußt sind (vgl. 5.3.3). Unter diesem Aspekt eignet sich daher A-Software tendenziell am wenigsten gut zur Wiederverwendbarmachung. Bei T-Software hingegen ist die grundsätzliche Bereitschaft zur Standardisierung deutlich höher, da viele Standards bereits bestehen – seien sie De-Facto-Standards oder von Gremien beschlossen (z.B. für verteilte Verarbeitung DCOM von Microsoft respektive CORBA).

Architektur. Die Architektur muß sich zur Wiederverwendbarmachung eignen. Dieses Kriterium hat große Bedeutung für a posteriori wiederverwendbare Software, da bestehende Architekturen nicht wesentlich geändert werden können. Geprüft werden muß hier, ob die notwendige Variabilität im Rahmen der bestehenden Architektur realisiert werden kann; eine Zerlegung in entkoppelte Komponenten hat hierfür große Bedeutung (auf die Variabilitätsmechanismen im Einzelnen gehen wir in 7.6 ein). Bei a priori wiederverwendbarer Software bezieht sich das Kriterium auf die geplante Architektur. Der Unterschied ist,

[78] Lediglich Produktunternehmen mit großer Marktmacht können erfolgreich eine Standardisierung von Anwendungsfunktionalität – teilweise auch gegen den Willen der Endbenutzer – durchführen, z.B. die Firma SAP AG mit ihrem Produkt R/3 in den Bereichen Buchhaltung und Controlling.

daß in diesem Fall in der Regel Änderungen zu vertretbaren Kosten möglich sind, da die Entwicklung in einem frühen Stadium ist.

Darüber hinaus wird die Nutzbarkeit für Entwickler zu einem gewissen Grad ebenfalls durch die Architektur bestimmt, was eine zusätzliche Prüfung erfordert: nur bei ausreichender Einfachheit und Klarheit ist sichergestellt, daß die Architektur für nutzende Entwickler verständlich ist.

Projektmanagement. Das erste Kriterium hinsichtlich des Projektmanagements ist trivial, aber schwer einzuhalten: es muß erreicht werden, daß ein ausreichendes Budget zur Verfügung steht, was wegen der geringen Bereitschaft zur Investition (vgl. 7.2.3) äußerst schwierig sein kann und voraussetzt, daß vorher eine Projektkalkulation durchgeführt wird, was nicht immer der Fall ist (vgl. 5.2.3). Außerdem ist die Projektgröße kritisch zu bewerten: die Schwierigkeit und damit die Gefahr des Scheiterns steigt generell mit dem Umfang des Projekts. Schließlich ist zu überprüfen, ob die Maßnahmen getroffen wurden, die für Projekte unter erschwerten Bedingungen – darum handelt es sich bei Projekten zur Entwicklung wiederverwendbarer Software (vgl. 7.2.3) – üblicherweise ergriffen werden müssen: Dazu gehören u.a. die Auswahl eines Projektleiters und eines Teams mit ausreichender Erfahrung sowie besondere Rigorosität beim Projektcontrolling.

Besondere Kriterien bei A-Priori-Wiederverwendbarkeit

Im Falle der A-Priori-Wiederverwendbarkeit ergeben sich Besonderheiten in allen drei Bereichen.

Spezifikation und Architektur. Die Gefahr des Over-Engineerings ist bei a priori wiederverwendbarer Software besonders groß, weil die Anforderungen nicht dadurch eingeschränkt sind, daß die Spezifikation ursprünglich auf einen konkreten Fall bezogen war (vgl. 7.2.3). Insofern ist es besonders wichtig, daß die Spezifikation auf den Anforderungsumfang überprüft wird.

Projektmanagement. Es muß dafür gesorgt werden, daß das ursprüngliche Entwicklungsprojekt nicht in Mitleidenschaft gezogen wird. Dies ist beispielsweise der Fall, wenn es durch den zusätzlichen Zeitaufwand für die Wiederverwendbarmachung unzulässig verzögert wird (vgl. das in 4.2.3 beschriebene Beispiel).

7.5.2 Softwareökonomische Kriterien

Die ökonomische Bewertung orientiert sich an der Einteilung des Nutzens (vgl. 7.4.1). Zunächst geben wir Kriterien hinsichtlich der operativen Auswirkungen und dann hinsichtlich der strategisch relevanten Auswirkungen an. Bei der Quantifizierung der ökonomisch relevanten Größen greifen wir auf das in Kapitel 6 vorgestellte ReValue-Modell zurück.

Kriterien bezüglich operativer Auswirkungen

Folgende Größen sind bei der Bewertung der operativen Auswirkungen abzuwägen:

- Höhe der Investition
- Zeitverzögerung, Höhe und Unsicherheit der erwarteten Kostenersparnis

Da Häufigkeit und zeitliche Verteilung der Nutzungsereignisse in der Regel im Vorhinein nicht feststehen, sondern nur statistische Aussagen darüber möglich sind, müssen die Kenngrößen für die Kostenersparnis ebenfalls statistisch beschrieben werden. Sie können je nach der – in den Modellparametern des ReValue Modells (vgl. 6.2) erfaßbaren – Konstellation hinsichtlich der Wiederverwendbarmachung und des Nutzungsprozesses in weiten Bereichen schwanken (vgl. 6.3). Die Zeitverzögerung, mit der die Amortisation der Investitionskosten eintritt, kann in der Größenordnung von Monaten oder Jahren liegen und ein großes Hindernis für Wiederverwendung darstellen (vgl. 4.2.3).

Die Kenngrößen müssen zunächst bestimmt werden, z.B. mit Hilfe des ReValue-Modells, um sie anschließend vor dem Hintergrund der operativen Ziele zu bewerten. Letztere werden von folgenden Größen beeinflußt:

- *Liquiditätssituation des Unternehmens*. Von der aktuellen Liquidität des Unternehmens hängt ab, ob Mittelabflüsse (negativer Cash Flow) grundsätzlich akzeptabel sind.
- *Konkurrenzsituation am Markt*. Die Härte des Wettbewerbs hat entscheidenden Einfluß auf das operative Vorgehen des Unternehmens. Verschiedene Faktoren spielen im Konkurrenzkampf am Markt eine Rolle: Preis, Verfügbarkeit des Produkts, Eigenschaften des Produkts (das Produkt kann dabei auch die Serviceleistung eines Projektunternehmens sein). Wenn der Wettbewerb über den Preis ausgetragen wird, so kann das dazu führen, daß zusätzliche Investitionskosten wegen eines Umsatzrückgangs nicht tragbar sind. Ist die Verfügbarkeit kritisch, so können keine Entwicklerkapazitäten von der unmittelbaren Produktentwicklung abgezogen werden. Dasselbe gilt, wenn das Produkt Defizite in seinen Eigenschaften hat.
- *Anforderungen von Banken und Investoren*. In seiner Geschäftspolitik muß sich das Unternehmen an den Anforderungen der Gläubigerbanken und Investoren orientieren. Diese sind in der Regel an einem positiven operativen Ergebnis und einem insgesamt positiven Cash Flow interessiert.

Die Investition in Wiederverwendbarkeit bewirkt zunächst eine Erhöhung der Kosten, was hinsichtlich der operativen Ziele folgende Auswirkungen hat:

- Die Kosten bedeuten einen Liquiditätsabfluß. Ob er toleriert werden kann, hängt von seiner Höhe ab und von der Zeitverzögerung, mit der die Amortisation der Investitionskosten eintritt. Diese Amortisation bedeutet einen Liquiditätszufluß.
- Durch die Investition verschlechtert sich die momentane Wettbewerbsposition des Unternehmens. Damit wird jedoch eine – je nach Konstellation unterschiedlich große – Chance auf eine zukünftige Verbesserung erwor-

ben. Die Ausgangsposition des Unternehmens entscheidet darüber, ob auf die vorübergehende Verschlechterung eingegangen werden kann.

- Die Investition bewirkt einen kurzfristigen Gewinnrückgang. Ob und in welchem Ausmaß er für Banken und Investoren wahrnehmbar ist, hängt von der Fristigkeit der Rechnungslegung und der Zeitverzögerung ab, mit der die Kostenersparnis durch die Nutzung eintritt. Es kann sein, daß – z.B. bei jährlicher Rechnungslegung und einer kurzen Zeitverzögerung in der Größenordnung eines Monats – diese Kosten zum Zeitpunkt der Veröffentlichung der Unternehmenszahlen längst kompensiert sind. Es ist aber auch möglich, daß sie – z.B. bei quartalsweiser Rechnungslegung und einer Verzögerung in der Größenordnung eines Jahres – sich mehrfach in den veröffentlichten Unternehmenszahlen auswirken, bevor die Ersparnis wirksam wird.

Grundsätzlich können alle Aussagen über die Kostenersparnis nur mit Unsicherheit gemacht werden. Welches Ausmaß an Unsicherheit akzeptabel ist, hängt von der generellen Risikobereitschaft des Unternehmens ab, die von den Erwartungen der Banken und Investoren beeinflußt wird. Börsennotierte Unternehmen sind einer strengen Publizitätspflicht unterworfen und daher zusätzlich eingeschränkt hinsichtlich der Risiken, die sie eingehen können.

Kriterien bezüglich strategisch relevanter Auswirkungen

Die strategisch wirksamen Nachteile der Entwicklung wiederverwendbarer Software (siehe 7.4.1) haben insgesamt nachrangige Bedeutung (vgl. 4.2.3). Insofern spielt vor allem die Abwägung mit eventuell entstehenden operativen Nachteilen eine Rolle.

Allgemeine strategische Ziele in Produkt- und Projektgeschäft und ihre typische Gewichtung sind in 7.4.2 angegeben. Wir gehen im folgenden auf Details ein und behandeln dabei die beiden Geschäfte getrennt.

Produktgeschäft

Bei der Bewertung des strategischen Nutzens im Produktgeschäft sind für den Einzelfall folgende Effekte zu beachten:

- Wie hoch die Bedeutung einer Senkung der time to market t_{tm} ist, hängt von zwei Faktoren ab: (i) Zunächst ist die aktuelle Wettbewerbsposition zu berücksichtigen. Ein Unternehmen, das bereits Marktführer ist, kann sich z.B. bemühen, durch Erhöhung der Kundenzufriedenheit sein Image im Markt zu verbessern, um sich eine Basis für die Eroberung neuer Märkte zu schaffen. (ii) Der Nutzen nimmt mit der Zeitverzögerung ab, mit der er eintritt, da diese zunächst die time to market verlängert.
- Die Größe des erreichbaren Marktes wird umso bedeutender, je höher die Durchdringung des aktuellen Marktes ist. Ein Unternehmen, das eine geringe Durchdringung im aktuell erreichbaren Markt hat, wird auf dessen Vergrößerung wenig wert legen, wogegen eine Unternehmen mit einer hohen Durchdringung für das Wachstum unbedingt darauf angewiesen ist.

- Wie eine Erhöhung der Kundenzufriedenheit zu bewerten ist, hängt wesentlich von deren Ausgangsniveau ab. Grundsätzlich ist davon auszugehen, daß es vor allem darum geht, ein Mindestniveau nicht zu unterschreiten.

Projektgeschäft

Im Einzelfall sind bei der Bewertung des strategischen Nutzens im Projektgeschäft folgende Effekte zu bedenken:

- Die Bedeutung verschiedener Einflußfaktoren für die Kundenzufriedenheit ist unterschiedlich und kann für denselben Kunden zeitlichen Schwankungen unterliegen. Die durch Nutzung wiederverwendbarer Software erreichbaren Steigerungen der Zuverlässigkeit und Benutzbarkeit für den Endbenutzer können dabei von kurzfristigen Kostensteigerungen oder Verlangsamung der Entwicklung konterkariert werden, die durch die Zeitverzögerung entstehen, mit der der Nutzen eintritt. Eine natürliches Korrektiv ist durch die Erfolgsbedingungen bei Akquisitionen vorgegeben. Gültig ist jedoch in allen Fällen, daß das vertretbare Höchstmaß an Kundenzufriedenheit angestrebt werden sollte.
- Hinsichtlich der time to market gilt folgende Überlegung analog zum Produktgeschäft: Der Nutzen nimmt mit der Zeitverzögerung ab, mit der der Nutzen eintritt, da diese Zeitverzögerung zunächst eine Verlängerung der time to market bedeutet.
- Auch für die Größe des erreichbaren Marktes gelten die Überlegungen für das Produktgeschäft analog, allerdings mit folgender Änderung: der Markt für Softwareprojekte ist in aller Regel wesentlich fragmentierter als der Markt für Produkte, so daß die Wahrscheinlichkeit wesentlich geringer ist, an die Grenze zu stoßen.

Abwägung der Kriterien

Die strategischen Ziele geben den Rahmen für das operative Geschäft vor (vgl. [Hun01], 41). Daher können Zugeständnisse hinsichtlich der Erreichung der operativen Ziele durch strategische Ziele gerechtfertigt sein. Dies darf jedoch nicht zu einer Gefährdung des operativen Geschäfts, z.B. durch zu hohe Liquiditätsabflüsse oder Verluste von Marktanteilenn führen. Das damit verbundene Problem der Abwägung, das in den oben ausgeführten Überlegungen immer wieder anklang, kann nur für den Einzelfall unter Berücksichtigung sämtlicher beschriebener Einflußgrößen gelöst werden. Der grundsätzliche Zusammenhang zwischen dem Risiko einer Investition und der damit verbundenen Chance gilt auch hier: je größer die Chance, desto höher das Risiko.

7.6 Softwaretechnische Leitlinien

In diesem Abschnitt behandeln wir das zentrale Problem des Vorgehens beim Entwurf wiederverwendbarer Software. Dies umfaßt zunächst die Frage nach der technisch und ökonomisch sinnvollen Realisierung von Variabilität, auf die wir im ersten Abschnitt eingehen. Im zweiten Abschnitt behandeln wir das Vorgehen bei der Zerlegung in Komponenten.

7.6.1 Kontrollierte Variabilität

Die Realisierung von Variabilität muß zunächst bei der Spezifikation der Anforderungen ansetzen. Bezogen darauf bedeutet Variabilität, daß es möglich ist, eine entwickelte Komponente an bestimmte Variationen in den Anforderungen anzupassen. Je höher die Variabilität, desto größer ist die Wiederverwendbarkeit der Komponente.

Trotzdem ist das Ziel nicht, die Variabilität zu maximieren, denn dem zusätzlichen Nutzen durch eine Erhöhung der Variabilität, sind die softwaretechnischen und softwareökonomischen Nachteile gegenüberzustellen. Die bei der Bewertung im Einzelnen anzuwendenden Kriterien sind in 7.5 angegeben. Die Bewertungsmethode wird also für den Vergleich unterschiedlicher Szenarien eingesetzt, die sich durch das angestrebte Ausmaß an Variabilität unterscheiden. Der Nutzen wird dabei mit der in 7.4.1 präsentierten Methode bestimmt, wobei die ökonomische Analyse mit Hilfe des ReValue-Modells durchgeführt wird. Das Ziel ist, das aufgrund der Abwägung optimale Maß an Variabilität zu erreichen, das wir *kontrollierte Variabilität* nennen.

Zwei Ziele sind für die Realisierung der Variabilität wichtig: Änderungseffizienz und Verständlichkeit. Sie gehen Hand in Hand, denn ein leicht verständlicher Variabilitätsmechanismus erzeugt bei der Änderung geringen Aufwand. Zu beachten ist darüber hinaus, daß die Aufgabe aufgrund mangelnder Verständlichkeit nicht mehr beherrschbar sein kann (vgl. 5.2.3). Die Untersuchungen in Kapitel 5 ergeben, daß eine Zerlegung des Systems in durch Schnittstellen entkoppelte Komponenten für die Variabilität eines Softwaresystems zentral ist. Auch die Art der Komponentenzerlegung spielt für die Variabilität eine wichtige Rolle. Es zeigt sich, daß die Anwendung der Software-Kategorien eine praktikable Methode dafür ist, die Trennung der Zuständigkeiten sicherzustellen.

Der wichtigste Variabilitätsmechanismus ist in diesem Zusammenhang der Austausch von Komponenten, die bestimmte Schnittstellen implementieren. Wie die empirischen Untersuchungen zeigen, ist die Verhaltensvererbung durch Schnittstellen der häufig emfohlenen Implementierungsvererbung (vgl. Kapitel 3) deutlich überlegen und sollte daher vorgezogen werden (vgl. 5.3.3). Eine wichtige Bestimmungsgröße für die Variabilität ist dabei Anzahl der Parameter in der Signatur der Schnittstelle: Zusätzliche Parameter bedeuten eine Erhöhung der Variabilität.

Wird die Trennung der Zuständigkeiten nach Software-Kategorien durchgeführt, so kommt als weiterer Variabiliätsmechanismus die Generierung zum Tragen: Komponenten, die stereotype Transformationen zwischen verschiedenen Software-Kategorien durchführen, können mit Hilfe von Metainformation generiert werden. Eine Änderung muß dann nur für die Metainformation, nicht aber für den Generator durchgeführt werden; der geänderte Code wird generiert, was die Effizienz wesentlich erhöht.

7.6.2 Zerlegung in Komponenten

Eine Formalisierung des Vorgehens bei der Zerlegung in Komponenten können wir nicht leisten. Wir geben jedoch Ziele und heuristische Verfahren an, mit deren Hilfe beim Entwurf wiederverwendbarer Software eine gute Lösung erreicht werden kann.

Die in 5.2.1 und 5.2.2 beschriebenen Erfolgsbeispiele zeigen, daß ein für die Zerlegung nutzbarer Begriff der Komponente auf einer höheren Granularitätsebene als der gebräuchlichen ansetzen muß, die einzelnen Objekten entspricht. Die Komponenten der Zerlegung sind vielmehr auf einer mittleren Granularitätsebene anzusiedeln und nach unserer Klassifikation als Baugruppen zu bezeichnen (vgl. 2.3.3). Zwei Ziele sind dabei wesentlich: Stabilität der Schnittstellen und Trennung der Zuständigkeiten.

Soll Variabilität durch Austausch von Schnittstellenimplementierungen erreicht werden, dürfen sich die Schnittstellen selbst nicht ändern. Daher ist es wichtig, die geforderte Variabilität genau im Sinne kontrollierter Variabilität zu spezifizieren. Dabei sollte ein iteratives Vorgehen gewählt werden.

Die Trennung der Zuständigkeiten kann durch die Regeln erreicht werden, die mit den Software-Kategorien verbunden sind. Die in 2.3.6 vorgestellten Kategorien stellen lediglich eine heuristische Annäherung an das im Einzelfall erzielbarer Optimum dar. Im Falle betrieblicher Informationssysteme sind die damit erzielten Ergebnisse jedoch bereits sehr gut (vgl. 5.3.3).

Kapitel 8

Zusammenfassung und Ausblick

Dieses Kapitel faßt den Inhalt der Arbeit in seinen Grundzügen zusammen und gibt einen Ausblick auf sich ergebende zukünftige Forschungsthemen.

Inhalt: **Seite**

8.1 Zusammenfassung

Ausgangspunkt dieser Arbeit ist die Diskrepanz zwischen den hohen Erwartungen an Wiederverwendung und dem geringen Grad der Umsetzung in der industriellen Praxis. Eine Vielzahl von – teilweise widersprüchlichen – Annahmen zu den Ursachen und notwendigen Maßnahmen bestehen, die empirisch jedoch unzureichend überprüft sind. Es zeigt sich, daß bisher kein dafür ausreichendes Datenmaterial zur Verfügung steht.

Daher entwickeln wir auf empirischer Basis systematisch eine differenzierte und umfassende Sicht der Wiederverwendung, welche sowohl die softwaretechnischen als auch die ökonomischen Aspekte berücksichtigt. Wir erklären den Status quo und leiten systematisch Leitlinien für ein erfolgreiches Vorgehen ab, die den Rahmenbedingungen in der industriellen Praxis Rechnung tragen.

Angesichts ihrer großen Bedeutung konzentrieren wir uns dabei auf betriebliche Informationssysteme, zumal diese in der Forschung bisher wenig beachtet wurden. Die Arbeit fokussiert auf die Entwicklung von Software-Komponenten und legt dabei den Schwerpunkt auf systematische Wiederverwendung innerhalb des Unternehmens.

Zunächst entwickeln wir ein konsistentes Begriffssystem für Wiederverwendung. Wir stellen den Forschungsbedarf fest und formulieren Hypothesen zu einer Reihe von wesentlichen und in der Literatur weitverbreiteten Annahmen, die wir systematisch anhand der Ergebnisse unserer Untersuchungen überprüfen.

In einer ersten Fallstudie bei einem großen internationalen Produktunternehmen erheben wir umfangreiche qualitative und quantitative Daten zu Rahmenbedingungen und Status quo der Wiederverwendung. Wir werten diese Daten aus und verwenden sie, um einen großen Teil der Hypothesen zu überprüfen, wobei wir einige davon falsifizieren. Daneben interpretieren wir die Ergebnisse im Zusammenhang und leiten allgemeine Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software ab. Es zeigt sich, daß die Entwickler der Wiederverwendung äußerst positiv gegenüberstehen – insbesondere spielt das häufig angeführte Not-Invented-Here-Syndrom keine Rolle – und großes Potential für einen Ausbau sehen. Als wesentliches Hindernis kristallisiert sich die mangelnde Bereitschaft zur Investition in die Entwicklung von und der daraus folgende Mangel an wiederverwendbarer Software heraus.

In weiteren Fallstudien untersuchen wir detailliert die softwaretechnischen Aspekte von drei Individual-Entwicklungsprojekten der sd&m AG, wobei wir den Schwerpunkt auf die Architektur legen. Aus der systematischen vergleichenden Analyse der Projekte gewinnen wir drei softwaretechnische Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software. Der erste ist die Sicherstellung der Durchführbarkeit, da es sich um Projekte unter erschwerten Bedingungen handelt, in denen insbesondere die Gefahr herrscht, bei der Spezifikation das Maß des Machbaren zu überschreiten. Die richtige Realisierung der Variabiliät stellt den zweiten Erfolgsfaktor dar. Im Zentrum steht dabei die Zerlegung des

Systems in durch Schnittstellen entkoppelte Komponenten und die Trennung der Zuständigkeiten auf Basis der Software-Kategorien, wobei ein wichtiges Ziel ist, den Anteil der Null-Software zu maximieren. Der dritte Erfolgsfaktor ist schließlich die Investition in die Nutzbarkeit für Entwickler, u.a. in Form von Generalisierung des Codes, strukturierter Dokumentation und Beispielanwendungen. Zusätzliches Ergebnis ist eine systematische Bestandsaufnahme von Nachteilen der Entwicklung wiederverwendbarer Software, zu denen der erhöhte Schwierigkeitsgrad, das ensprechende Risiko des Scheiterns und die zunächst entstehende Verzögerung der Entwicklung insgesamt gehören.

Schließlich entwickeln wir ein umfassendes Modell – das ReValue-Modell – für die ökonomische Bewertung der Entwicklung wiederverwendbarer Software. Es beschreibt die relevanten Einflußgrößen durch Parameter und bestimmt die Rendite der Investition in die Entwicklung wiederverwendbarer Software nach der Kapitalwertmethode. Da in der Praxis keine ausreichenden Daten zur Verfügung stehen, gewinnen wir diese durch Monte-Carlo-Simulation auf Basis des ReValue-Modells. Für verschiedene Szenarien, die Komponenten unterschiedlicher Software-Kategorien entsprechen, ermitteln wir so den Erwartungswert der Rendite und die Unsicherheit der Investition in deren Wiederverwendbarmachung. Es zeigt sich, daß die Entwicklung wiederverwendbarer Software keinesfalls – wie weithin angenommen – immer sehr rentabel ist. Vielmehr schwankt der mögliche finanzielle Ertrag je nach Konstellation zwischen hohen Verlusten und hohen Gewinnen. Bei entsprechender Berücksichtigung dieser Unsicherheit reduziert sich die zu erwartende Rendite außerdem, teilweise um mehr als eine Größenordnung. Daneben ist die Verzögerungszeit zwischen Investition und deren Amortisation unterschiedlich lang und kann deutlich über ein Jahr betragen, wenn auch dabei die Unsicherheit der Aussage in Betracht gezogen wird. Für die simulierten Szenarien ergeben sich folgende grundsätzliche Aussagen, wobei die relative Positionierung über die konkreten Szenarien hinaus von Interesse ist: wiederverwendbare Null-Software erwirtschaftet hohe Renditen bei vergleichsweise geringer Unsicherheit, T-Software bringt hohe Renditen bei hoher Unsicherheit, die Retabilität der Investition in A-Software ist gering bei hoher Unsicherheit.

Aus den Ergebnissen der verschiedenen Untersuchungen entwickeln wir Leitlinien für die Entwicklung systematisch wiederverwendbarere Software. Wir zeigen zunächst, wie sich der Status quo der Wiederverwendung aus der Industriestruktur erklären läßt und entwickeln, aufbauend auf den Ergebnissen der Untersuchungen, einen differenzierten Begriff der Wiederverwendbarkeit, indem wir sie aus Sicht der Investition betrachten und im Kontext der Softwareentwicklung untersuchen. Auf Grundlage einer systematischen Nutzenanalyse formulieren wir strategische Leitlinien, indem wir die Auswirkung des Nutzens für die Erreichung verschiedener strategischer Ziele bestimmen. Wir geben – in Form softwaretechnischer und ökonomischer Kriterien – Leitlinien für die Bewertung von Projektvorhaben an, wobei wir die Ergebnisse der Fallstudien synthetisieren und für die ökonomische Bewertung auf das ReValue-Modell zurückgreifen. Schließlich formulieren wir softwaretechnische Leitlinien für die Realisierung von Variabilität und die Zerlegung eines Systems in Komponenten.

Wissenschaftliche Beiträge leisten wir mit den empirischen Untersuchungen und den daraus gezogenen Schlußfolgerungen inklusive der überprüften Hypothesen, mit dem ReValue-Modell samt Simulationsergebnissen, mit der Verfeinerung des Wiederverwendbarkeitsbegriffs, mit den Leitlinien zur Bewertung von Projektvorhaben sowie mit den strategischen und softwaretechnischen Leitlinien.

8.2 Ausblick

Die Arbeit fokussiert auf betriebliche Informationssysteme, die Ergebnisse sind jedoch nicht notwendigerweise darauf beschränkt. Die vorgestellten Methoden lassen sich zum Teil – direkt oder in angepaßter Form – auf andere Domänen übertragen.

Das ReValue-Modell kann durch entsprechende Wahl der Parameter in unterschiedlichsten Kontexten Anwendung finden. Die Software-Kategorien, die sich als äußerst nützliche Richtschnur für Analyse und Konstruktion erwiesen haben, versprechen ebenfalls Nutzen für andere Domänen, müssen vorher jedoch daran angepaßt werden. Dies ist das Ziel der Arbeit an deren Verallgemeinerung und Formalisierung unter Mitwirkung des Autors; erste Ergebnisse (vgl. [SiS02]) stehen demnächst zur Veröffentlichung an.

Daneben scheint eine Fortsetzung der empirisch fundierten Erforschung der Wiederverwendung vielversprechend, die ebenfalls auf andere Domänen ausgedehnt werden kann. Insbesondere das systematische Sammeln und Analysieren ökonomischer Daten bietet bei entsprechender Kooperationsmöglichkeit mit Software-Unternehmen ein lohnendes Betätigungsfeld für die Validierung der mit dem ReValue-Modell erzielten Simulationsergebnisse.

Fortschritte im Bereich der Software-Wiederverwendung können nicht nur durch Verbesserungen des Vorgehens, sondern auch durch geänderte Rahmenbedingungen erzielt werden. Mit zunehmender Reife der immer noch jungen Software-Industrie sind Abmilderungen der Erwartungen des Kapitalmarkts möglich, die sich positiv auf die Bereitschaft zu längerfristigen Investitionen und damit auf die Rahmenbedingungen der Entwicklung wiederverwendbarer Software auswirken würden. Sollte sich zudem die technische Entwicklung im Bereich von Hard- und Software verlangsamen – eine seriöse Prognose hierzu ist schwierig – und damit die Innovationszyklen verlängern, würde sich dies äußerst positiv auf die Renditeerwartungen auswirken. In jedem Fall gilt, daß sich die zukünftige Forschung an den Rahmenbedingungen der industriellen Praxis der Softwareentwicklung orientieren und deren Änderungen beobachten sollte.

Anhang Fragebogen zu Kapitel 4

Wir drucken hier den Fragebogen, der bei der in beschriebenen Untersuchung verwendet wurde, in seiner Originalfassung ab.

Einführung

Ziel dieser Umfrage ist, herauszufinden, wie Reuse (Mehrfachverwendung) in der Entwicklung momentan betrieben wird und welche Verbesserungsmöglichkeiten bestehen. Aus den Ergebnissen der Umfrage sollen Maßnahmen abgeleitet werden, die einen effektiveren Einsatz von Reuse ermöglichen. Dadurch soll z.B. die Reuse Library noch besser an die Bedürfnisse der Entwickler angepasst werden.

Gegenstand der Untersuchung sind

- der gesamte Entwicklungsprozess (von der Spezifikation über Design und Implementierung bis hin zu Test und Benutzer-Dokumentierung) und
- alle Zwischen- und Endprodukte, die im Verlauf des Prozesses entstehen und wiederverwendet werden können – im Folgenden sämtlich ***Reusables*** genannt:

– Teile des an den Kunden ausgelieferten Produkts, also neben Codestrecken oder Softwaremodulen auch die Dokumentierung

– Alle Arten von Zwischenprodukten, die im Verlauf des Entwicklungsprozesses entstehen und die man unverändert oder in modifizierter Form wiederverwendet. Dies können Templates sein (z.B. für Spezifikation, Design oder Codierung) oder modellhafte Beschreibungen (z.B. in UML oder als Entity Relationship-Diagr.), Testdaten, aber auch Geschäftsprozessbeschreibungen oder use cases.

– Prozess- oder Vorgehensbeschreibungen, z.B. zur Durchführung von Tests.

Oft ist es sinnvoll, zwischen Nutzern von Reusables und Herstellern von Reusables zu unterscheiden. Nutzer betreiben Entwicklung mit Reuse während Hersteller Entwicklung für Reuse machen.

Die Reusables können ganz unterschiedliche Granularität (vergleichen Sie z.B. einen einzelnen Funktionsbaustein mit einem abgeschlossenen Modul) und verschiedene Allgemeinheitsgrade (gemessen als Anzahl potentieller oder tatsächlicher Verwendungen, z. B. zwischen drei und mehreren Hundert) haben.

Bitte beantworten Sie alle Fragen aus Ihrer persönlichen Sicht, da nur so ein repräsentatives Bild gewonnen werden kann. Konstruktive Verbesserungsvorschläge sind sehr willkommen.

Um Ihnen detailliertere Aussagen zu ermöglichen, können Sie im Fragebogen an verschiedenen Stellen „Fließtext“ eintragen. Bitte machen Sie von dieser Möglichkeit Gebrauch, denn dadurch werden die Ergebnisse konkreter und aussagekräftiger! Natürlich sind auch mündliche Auskünfte im Interview sehr willkommen.

Vielen Dank für Ihre Teilnahme an der Umfrage!

1. Vorgehen und Systemunterstützung bei Reuse

1.1 **Nutzt Ihre Abteilung Reusables** auf den verschiedenen Stufen des Entwicklungsprozesses?
Falls ja: Wie hoch ist der **Grad der Nutzung**?
Bitte geben Sie auch an, wie hoch Sie das **Potential für mehr Reuse** einschätzen.

	Nutzung?		Grad der Nutzung?					Potential für mehr Reuse?				
	ja	nein	sehr hoch	hoch	mittel	niedrig	sehr niedrig	ja	nein	hoch	mittel	niedrig
• Spezifikation	O	O	O	O	O	O	O	O	O	O	O	O
• Design	O	O	O	O	O	O	O	O	O	O	O	O
• Implementierung												
– Codierung	O	O	O	O	O	O	O	O	O	O	O	O
– Alpha-Test (eig. Tests)	O	O	O	O	O	O	O	O	O	O	O	O
– Techn. Dokumentierg.	O	O	O	O	O	O	O	O	O	O	O	O
• Beta-Test (Module Inte-, gration T., Acceptance T.)	O	O	O	O	O	O	O	O	O	O	O	O
• Benutzer-Dokumentierg.	O	O	O	O	O	O	O	O	O	O	O	O

Bemerkungen: _____

_____

_____

_____

1.2 **Bitte kreuzen Sie an**, welche Reusables **genutzt werden** (Mehrfachnennungen möglich).
Unterstreichen Sie bitte zusätzlich diejenigen, die **darüber hinaus potentiell genutzt werden könnten**.
Ergänzen Sie bitte nicht aufgeführte Reusables.

- Spezifikation O Templ. O Modelle O Geschäftsprozessbeschreibungen O Use cases
  O Andere: _____
- Design O Templ. O Modelle O Andere: _____
- Implementierung
  - Codierung O Funktionsbausteine O Starre Kopiervorlagen O Demo-Programme O Generatoren
    O Templates O Klassen O Andere: _____
  - Alpha-Test O Templ. O Testdatenbank O Andere: _____
  - Techn. Doku. O Templ. O Vorh. Text O Andere:_
- Beta-Test O Templ. O Testdatenbank O Andere: _____
- Benutzer-Doku. O Templ. O Vorh. Text O Andere:_

Bemerkungen: _____

_____

_____

1.3 Reuse Library

1.3.1 Kennen Sie die Reuse Library? O Ja O Nein *(falls Nein: direkt zu 1.4 springen)*

1.3.2 Wissen Sie, wie sie bedient wird? O Ja O Nein

1.3.3 Wissen Sie, wie sie bestückt wird? O Ja O Nein

1.3.4 Wird sie genutzt? O Ja O Nein → **Falls ja**: Wie hoch ist der Grad der Nutzung?

sehr hoch	hoch	mittel	niedrig	sehr niedrig
O	O	O	O	O

1.3.5 Würde sie mehr genutzt werden, wenn daran etwas verbessert würde? O Ja O Nein

Falls ja: was? O Bedienung O Aufbau O Dokumentation O Verfügbarkeit lokaler Bausteine

O Anderes: _____

Fragen zur Nutzung *(wenn die Reuse Library nicht genutzt wird, können Sie direkt zu 1.4 springen)*

1.3.6 Welche Inhalte werden in Ihrer Abteilung genutzt?

O Programmcode O Beispiele O Dokumentation O Andere:_____

1.3.7 Wieviel Prozent der benötigten Komponenten sind in der Reuse Library enthalten? __%

1.3.8 Fehlen wichtige Standardbausteine in der Reuse Library? O Ja O Nein

Falls ja: welche? _____

1.4 Andere Tools für die Suche und Verwaltung von Reusables

1.4.1 Werden andere Tools für die Suche und Verwaltung von Reusables verwendet? O Ja O Nein

Falls ja: welche? _____

_____

1.4.2 Hätten Sie gerne andere/weitere zur Verfügung? O Ja O Nein

Falls ja: welche? _____

_____

1.6 Überblick über Quellen für Reuse: Bitte bringen Sie die folgenden Quellen für Reuse in eine Rangordnung. Falls für Sie andere Quellen als die aufgeführten relevant sind, ergänzen Sie diese bitte unten.

Rang

_____. Eigene Entwicklung

_____. Persönliche Kontakte innerhalb der Gruppe/Abteilung

_____. Persönliche Kontakte außerhalb der Gruppe/Abteilung

_____. Reuse Library

_____. _____

2. Erfahrungen mit Reuse

2.1 Stellenwert und Potential von Reuse

Sollte Ihrer Meinung nach in Zukunft weniger, gleich viel oder mehr Reuse betrieben werden?

viel weniger	weniger	gleich viel	mehr	viel mehr
O	O	O	O	O

2.2 Vor- und Nachteile von Reuse

2.2.1 Bitte bewerten Sie die **Vorteile** von Reuse. *Gehen Sie dabei bitte wie folgt vor*:

- ***Kreuzen Sie** zunächst die aus Ihrer Sicht **relevanten Vorteile an** (Kreis unter „Relev.“)*
- ***Bewerten Sie** anschließend die **Bedeutung der angekreuzten Vorteile** auf folgender Skala:*
  1 = sehr hoch, 2 = hoch, 3 = mittel, 4 = niedrig, 5 = sehr niedrig ("Bedeutg.")
- *Falls für Sie andere Vorteile als die aufgeführten relevant sind, **ergänzen** Sie diese bitte unten!*

Relev.	Bedeutg.	
O	_____	Höhere Produktivität bei Nutzung von Reusables/Entwicklung mit Reuse (dadurch, dass weniger neu entwickelt werden muss)
O	_____	Insgesamt geringerer Wartungsaufwand durch einmalige, zentrale Wartung
O	_____	Höhere Qualität (Fehlerfreiheit), da Reusables häufiger getestet sind
O	_____	Leichtere Erlernbarkeit der Bedienung für den Benutzer (durch Einheitlichkeit)
O	_____	Geringere Entwicklungszeit (time to market)
O	_____	_____
O	_____	_____

2.2.2 Bitte bewerten Sie die **Nachteile** von Reuse. *Gehen Sie dabei bitte analog zu 2.2.1 vor.*

Relev.	Bedeutg.	
O	_____	Zusätzlicher Aufwand für Abstimmung, Generalisierung, Dokumentierung etc. bei Herstellung von Reusables/Entwicklung für Reuse
O	_____	Entwicklung mit Reuse führt zu schlechterer Performance (Ballast wird mitgeschleppt)
O	_____	Entwicklung mit Reuse führt zu suboptimalen, nicht maßgeschneiderten Lösungen
O	_____	_____
O	_____	_____

2.2.3 Was überwiegt?

Nachteile überwiegen deutlich	Nachteile über-wiegen	Vor- und Nach-teile sind ausgeglichen	Vorteile über-wiegen	Vorteile überwiegen deutlich
O	O	O	O	O

2.3 Anreize und Hindernisse für Reuse

2.3.1 Bitte geben Sie an, was **aus Ihrer Sicht Reuse verhindert**.
*Gehen Sie bei der Beantwortung bitte analog zu **2.2.1** vor.*

Relev.	Bedeutg.	
O	_____	Mangel an Zeit erlaubt keinen Zusatzaufwand
O	_____	Personelle Ressourcen für Zusatzaufwand stehen nicht zur Verfügung
O	_____	Höherer Dokumentierungs- und Wartungsaufwand schreckt von Entwicklung für Reuse ab
O	_____	Reuse führt zu schlechteren Produkten (schlechtere Performance und/oder suboptimale, nicht maßgeschneiderte Lösungen)
O	_____	Entwicklung für Reuse wird bestraft/nicht ausreichend belohnt
O	_____	Entwicklung mit Reuse wird bestraft/nicht ausreichend belohnt
O	_____	Entwickler sind nicht ausreichend informiert
O	_____	Systemunterstützung ist nicht ausreichend
O	_____	Es fehlt ein zentraler Ansprechpartner/Koordinator
O	_____	Abteilungsdenken erschwert übergreifende Zusammenarbeit
O	_____	Spezifikation ist am Anfang noch zu unklar
O	_____	Identifikation lohnender Komponenten ist schwierig
O	_____	Bei Entwicklung mit Reuse nimmt die Kontrolle über das eigene Produkt ab
O	_____	Es wird zu wenig objektorientiert entwickelt

2.3.2 Durch **welche Maßnahmen** könnte Reuse Ihrer Meinung nach **am effektivsten gefördert** werden? Bitte nennen Sie – falls möglich – bis zu drei mögliche Maßnahmen.

1. _____
2. _____
3. _____

2.3.3 **Welche Anreize sollte man Abteilungsleitern geben**, damit sie mehr Reuse betreiben?
*Gehen Sie bei der Beantwortung bitte analog zu **2.2.1** vor.*

Relev.	Bedeutg.	
O	_____	Keine – vorrangig sollten Hindernisse abgebaut werden
O	_____	Direkte finanzielle Anreize (z.B. fester Betrag für jede Verwendung einer Komponente)
O	_____	Förderung durch bessere Aufstiegsmöglichkeiten
O	_____	Förderung durch – ggf. selbst gewählte - interessante Aufgaben
O	_____	Sichtbares Honorieren der Leistung
O	_____	Erfolgreiche Reuse-Arbeit zur Voraussetzung für das weitere Vorankommen machen
O	_____	_____

3. Wirtschaftliche Aspekte

3.1 Entwicklungs-Mehraufwand

3.1.1 Um welchen Faktor ist Ihrer Schätzung nach der Aufwand bei der Entwicklung für Reuse höher als bei Einmal-Entwicklung?

Durchschnitt: _____ Minimum: _____ Maximum: _____

3.1.2 Wie oft muss Ihrer Meinung nach eine Komponente wiederverwendet werden, bis der Break-even erreicht wird (d.h. insgesamt der Mehraufwand durch Einsparungen aufgewogen wird)?

Durchschnitt: _____ Minimum: _____ Maximum: _____

3.2 Komponentengröße/Granularität

Gibt es Ihrer Meinung nach eine optimale Komponentengröße/Granularität für Reuse?

O Ja O Nein -> Falls ja: welche ist optimal?

klein (z.B einzelner Funktionsbaustein)	eher klein	mittel	eher groß	groß (z.B. R/3-Modul)
O	O	O	O	O

4. Organisation und Strategie

4.1.1 Zentrale vs. dezentrale Reuse-Organisation: Sollten die Aufgaben im Zusammenhang mit Reuse schwerpunktmäßig zentral oder dezentral wahrgenommen werden?

zentral	eher zentral	ausgeglichen	eher dezentral	dezentral
O	O	O	O	

4.1.2 Wie sollten die folgenden Aufgaben im Einzelnen verteilt werden?

	dezentral	zentral
• Sammeln und screenen der Reuse-Bausteine	O	O
• Aufbereiten der Reuse-Bausteine		
– Generalisierung	O	O
– Dokumentierung, Bereitstellung von Beispielen	O	O
– Erstellung von Test-Cases	O	O
• Verwalten der Infrastruktur (Reuse Library, Entwicklgsumg.)	O	O
• Verwalten der Reuse-Prozesse	O	O
• Eigentliche Entwicklung der Reuse-Bausteine	O	O
• _____	O	O

4.2 Organisation übergreifender Vorhaben: Wie können Reuse-Vorhaben, die über eine Abteilung hinausgehen, am besten organisiert werden?

4.3 Bedeutung einer unternehmensweiten Reuse-Strategie: Wie groß ist Ihrer Meinung nach die Implementierung einer breit kommunizierten unternehmensweiten Reuse-Strategie?

sehr klein	klein	mittel	groß	sehr groß
O	O	O	O	O

5. Reuse in konkreten Projekten

5.1 Projekte mit erfolgreich praktiziertem Reuse

5.1.1 Kennen Sie Projekte, in denen besonders erfolgreich Reuse praktiziert wurde? O Ja O Nein

Falls ja: welche?

5.1.2 Kennen Sie Projekte, in denen Neuentwicklungen aufgrund ähnlicher Anforderungen in verschiedenen Bereichen (z.B. mehrere IBUs) von Beginn an abgestimmt wurden, um ein Maximum an Gleichteilen zu erreichen, die gemeinsam entwickelt und genutzt wurden (a priori-Reuse)? O Ja O Nein

Falls ja: welche?

5.2 Potential für Reuse

5.2.1 Kennen Sie Projekte, die ein **großes, bisher nicht ausgeschöpftes Reuse-Potential** haben?

O Ja O Nein

Falls ja: welche?

5.2.2 Was müsste in diesen Projekten getan werden, um das Reuse-Potential auszuschöpfen?

6. Identifikation/Reuse Hunting und Metriken

6.1 Wie würden Sie vorgehen, um für Reuse geeignete Bausteine zu identifizieren?

6.1.2 Eine Möglichkeit der Identifikation von Reusables ist, die Entwicklungsprojekte, in denen sie entstanden sind, auf einer von den technischen Details abstrahierten Ebene zu beschreiben. Eine Möglichkeit dafür würde ein "elektronischer Marktplatz" für Entwicklungsprojekte bieten. Wie würden Sie den Nutzen eines solchen Marktplatzes einstufen?

sehr hoch	hoch	mittel	niedrig	sehr niedrig
O	O	O	O	O

6.1.3 Welche Funktionen sollte ein solcher Marktplatz haben?

6.1.4 Der elektronische Marktplatz könnte neben bereits abgeschlossenen Entwicklungsprojekten auch Entwicklungsvorhaben von der Planungsphase an umfassen, um eine frühzeitige Koordination verwandter Vorhaben zu ermöglichen. Wie würden sie den Nutzen dieser Maßnahme einstufen?

sehr hoch	**hoch**	**mittel**	**niedrig**	**sehr niedrig**
O	O	O	O	O

6.2 Wie würden Sie den Erfolg der Reuse-Bemühungen messen?

Angaben zum Befragten

Name, Vorname: _____

Telefon: _____

Bereich: _____
Abteilung: _____

- Wie viele Gruppenleiter hat Ihre Abteilung? _____
- Wie viele Entwickler hat Ihre Abteilung? _____
- Wie lange arbeiten Sie schon beim Unternehmen?_____ Jahre
- Wird in Ihrer Abteilung objektorientiert entwickelt? O Ja O Nein
  - **Falls ja:** welchen Anteil hat die OO-Entwicklung? _____%
  - Wie ist die Tendenz? O steigend O gleichbleibend O fallend

Zusätzliche persönliche Informationen, die für die wissenschaftliche Auswertung sehr hilfreich wären:

- Welche Ausbildung haben Sie? O Uni O FH O andere: _____
  Fach: _____

Vielen Dank für Ihre Teilnahme an der Umfrage!

Literatur

[Ald00] Aldrich, J. (2000): *Challenge problems for separation of concerns*, Minneapolis: OOPSLA Workshop on Advanced Separation of Concerns 2000

[AHU74] Aho, A.V.; Hopcroft, J.E.; Ullman, J.D. (1974): *The design and analysis of computer algorithms*, Reading, Mass.: Addison-Wesley 1974.

[AKK00] Asundi, J.; Kazman, R.; Klein, M. (2000): *An architectural approach to software cost modeling*, in: Proc. EDSER 2 – Second Workshop on Economics-Driven Software Engineering Research, 2000.

[Amb98] Ambler, S.W. (1998): *Process patterns: building large-scale systems using object technology*, Cambridge, UK: Cambridge University Press 1998.

[AnG01] Anastasopoulos, M.; Gacek, C. (2001*): Implementing product line variabilities*, Toronto, Canada: Symposium on Software Reusability (SSR) 2001.

[AOS02] Aspect-Oriented Software Development website, Januar 2002, http://aosd.net

[ArG97] Arnold, K.; Gosling, J. (1997): *The Java™ programming language*, second edition, 7th printing February 2000, Reading, MA: Addison-Wesley 1997.

[AsK01] Asundi, J.; Kazman, R. (2001): *A foundation for the economic analysis of software architectures*, in: Proc. EDSER 3 – Third Workshop on Economics-Driven Software Engineering Research, 2001.

[Att95] Atteslander, P. (1995): *Methoden der empirischen Sozialforschung*, 8. Auflage, Berlin; New York: de Gruyter 1995.

[Bab97] Baber, R.L. (1997): *Comparison of electrical 'engineering' of Heaviside's times and software 'engineering' of our times*, in: IEEE Annals of the History of Computing, 1997, 19/4, S. 5-17.

[BaC95] Basili, V.; Caldiera, G. (1995): *Improve software quality by reusing knowledge and experience*, in: Sloan Management Review, 1995, 37/1, S. 55-64.

[BFK99] Bayer, J.; Flege, O.; Knauber, P.; Laqua, R.; Muthig, D.; Schmid, K.; Widen, T.; DeBaud, J.-M. (1999): *PuLSE: a methodology to develop software product lines*, in: Proc. SSR Symposium on Software Reusability, 1999, S. 122-131.

[Bai99] Bailin, S.C. (1999): *Like excrement... reuse occurs*, Austin, Texas: 9th Workshop in Institutionalizing Software Reuse WISR 1999.

[Bal96] Balzert, H. (1996): *Lehrbuch der Software-Technik - Band 1: Software-Entwicklung*, Heidelberg: Spektrum 1996.

[Bal98] Balzert, H. (1998): *Lehrbuch der Software-Technik - Band 2: Software-Management*, Heidelberg: Spektrum 1998.

[Bal99] Balzert, H. (1999): *Lehrbuch Grundlagen der Informatik*, Heidelberg: Spektrum 1999.

[Bar99] Bartsch, H.J. (1999): *Taschenbuch mathematischer Formeln*, 18. Auflage, München: Carl Hanser Verlag 1999.

[Bas97] Bassett, P.G. (1997): *Framing software reuse: lessons from the real world*, Upper Saddle River: Yourdon Press/Prentice Hall 1997.

[BBa01] Bachmann, F.; Bass, L. (2001*): Managing variability in software architectures*, Toronto, Canada: Symposium on Software Reusability (SSR) 2001.

[BBo91] Barnes, B.; Bollinger, T. (1991): *Making reuse cost-effective*, in: IEEE Software, 1991, Januar, S. 13-24.

[BCJ99] Butler, S.; Chalasani, P.; Jha, S.; Raz, O.; Shaw, M. (1999): *The potential of portfolio analysis in guiding software decisions*, in: Proc. EDSER 1 – First Workshop on Economics-Driven Software Engineering Research, 1999.

[BCK98] Bass, L.; Clements, P.; Kazman, R. (1998): *Software architecture in practice*, Reading, MA: Addison-Wesley 1998.

[BDH98] Broy, M.; Deimel, A.; Henn, J.; Koskimies, K.; Plasil, F.; Pomberger, G.; Pree, W.; Stal, M.; Szyperski, C. (1998): *What characterizes a software component*?, in: Software – Concepts & Tools, 1998, 19/1, S. 49-56.

[BeA01] Bergmans, L.M.J.; Aksit, M. (2001): *How to deal with encapsulation in aspect-orientation*, in: Proc. OOPSLA Workshop on Advanced Separation of Concerns, 2001.

[Ben01a] Beneken, G. (2001): *Quasar – Erweiterbarkeit*, München: Internes Dokument sd&m AG 2001.

[Ben01b] Beneken, G. (2001): *Release Letter QDI 1.0*, München: Internes Dokument sd&m Research GmbH 2001.

[BeS01a] Beneken, G.; Siedersleben, J. (2001): *Quasar – QDI Tutorial*, München: Internes Dokument sd&m Research GmbH 2001.

[BeS01b] Beneken, G.; Siedersleben, J. (2001): *Quasar Datenverwaltung (QDI) – Programmers Reference*, München: Internes Dokument sd&m Research GmbH 2001.

[BeS97] Berson, A.; Smith, S.J. (1997): *Data Warehousing, Data Mining, & OLAP*, New York: McGraw-Hill 1997.

[Beu99] Beugnard, A. (1999): *How to make aspects reusable, a proposition*, in: Proc. ECOOP Aspect-Oriented Programming Workshop, 1999, S. 1-4.

[BJR98] Booch, G.; Jacobson, I.; Rumbaugh, J. (1998): *The unified software development process*, Reading, MA: Addison-Wesley 1998.

[BMR00] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. (2000): *Pattern-orientierte Softwarearchitektur: ein Pattern-System*, 1. korr. Nachdruck, München: Addison-Wesley/Pearson Education 2000.

[BoB01] Boehm, B.; Basili, V. (2001): *The CeBASE framework for strategic software development and evolution*, in: Proc. EDSER 3 – Third Workshop on Economics-Driven Software Engineering Research, 2001.

[Boe81] Boehm, B. (1981): *Software Engineering Economics*, Englewood Cliffs, NJ: Prentice-Hall 1981.

[Boe84] Boehm, B.W. (1984): *Software Engineering Economics*, in: IEEE Transactions on Software Engineering, 1984, Vol. SE-10 (1), S. 4-21.

[Bos92] Bosch, K. (1992): *Elementare Einführung in die angewandte Statistik*, 4. Auflage, Braunschweig: Vieweg 1992.

[Box98] Box, D. (1998): *Essential COM*, Reading, MA: Addison-Wesley 1998.

[Bra01] Brav, A. (2001): *Budgeting using Internal Rate of Return*, Vorlesungsunterlagen Duke University: http://www.duke.edu/ ~brav/TEACHING/MATERIALS/ Topic5_budgeting.html 2001.

[BrM95] Brealey, R.A.; Myers, S.C. (1995): *Principles of corporate finance*, fourth edition, New York: McGraw-Hill 1995.

[Bro87] Brooks, F.P. (1987): *No silver bullet: Essence and accidents of software engineering*, in: IEEE Computer, 1987, April, S. 10-19.

[Bro95] Brooks, F. (1995): *The mythical man-month*, 20th anniversary edition, 9th printing 1998, Reading, Mass.: Addison-Wesley 1995.

[BrS02] Broy, M.; Siedersleben, J. (2002): *Objektorientierte Programmierung & Softwareentwicklung: eine kritische Einschätzung*, in: Informatik-Spektrum, 2002, 1

[BSa97] Bandinelli, S.; Sagarduy, G. (1996): *A unifying framework for reuse economic models*, European Software Institute: Technical Report No. ESI-1996-REUSE03 1996.

[BSe01] Bronstein, I.N.; Semendjajew, K.A.; Musiol, G.; Mühlig, H. (2001): *Taschenbuch der Mathematik*, 5. Auflage, Thun: Verlag Harri Deutsch 2001.

[BSi00] Brössler, P.; Siedersleben, J. (2000) (Hrsg.): *Softwaretechnik*, München: Carl Hanser 2000.

[Buc99] Buck-Emden, R. (1999): *Die Technologie des SAP R/3-Systems*, 4. Auflage, Reading, Mass.: Addison-Wesley 1999.

[Bun00] Bundesaufsichtsamt für das Kreditwesen (2000) (Hrsg.): *Grundsatz I über die Eigenmittel der Institute*, http://www.bakred.de 2000.

[Btl99] Butler, G. (1999): *Developing frameworks by aligning requirements, design, and code*, Austin, Texas: 9th Workshop in Institutionalizing Software Reuse WISR 1999.

[But99] Butler, J. (1999): *Components: reuse in action*, in: Tinnirello, P.C. (Hrsg.): Systems development handbook, Boca Raton: Auerbach 1999, S. 147-156.

[CaE95] Carroll, M.D.; Ellis, M.A. (1995): *Designing and coding reusable C++*, Reading, MA: Addison-Wesley 1995.

[CCo94] Card, D.; Comer, E. (1994): *Why do so many reuse programs fail?*, in: IEEE Software, 1994, September, S. 114-115.

[ChD00] Cheesman, J.; Daniels, J. (2000): *UML components: a simple process for specifying component-based software*, Boston: Addison-Wesley 2000.

[ChN95] Chau, C.T.; Nordhauser, F. (1995): *Adding dynamics to the master budget: flexible budgeting with micro-computer simulation procedures*, in: Journal of Accounting and Computers, 1995, XI (Fall)

[CKK01] Clements, P.; Kazman, R.; Klein, M. (2001): *Evaluating software architectures: methods and case studies*, Boston, Mass.: Addison-Wesley 2001.

[Cle99] Clements, P.C. (1999): *Essential product line practices*, Austin, Texas: 9th Workshop in Institutionalizing Software Reuse WISR 1999.

[Coh99] Cohen, S. (1999): *From use cases to domains to architecture*, Austin, Texas: 9th Workshop in Institutionalizing Software Reuse WISR 1999.

[CEP96] Computational Science Education Project (1996): *Introduction to Monte Carlo methods*, Elektronisches Buch: http://csep1.phy.ornl.gov/mc/mc.html 1996

[CHW98] Coplien, J.; Hoffman, D.; Weiss, D. (1998): *Commonality and variability in software engineering*, in: IEEE Software, 1998, November, S. 37-45.

[CKM94] Copeland, T.; Koller, T.; Murrin, J. (1994): *Valuation – measuring and managing the value of companies*, Second edition, New York: John Wiley & Sons 1994.

[Coc02] Cockburn, A. (2002): *Agile software development*, Boston, Mass.: Pearson Education 2002.

[Cou97] Coulange, B. (1997): *Software reuse*, Berlin: Springer 1997.

[Cox87] Cox, B.J. (1987): *Building malleable systems from software 'chips'*, in: Computerworld, 1987, 21(13), S. 59-62, 64-8.

[Cox90] Cox, B. (1990): *There is a silver bullet*, in: Byte, 1990, October, S. 209-18.

[CuS95] Cusumano, M.; Selby, R. (1995): *Microsoft secrets*, New York: The Free Press 1995.

[DeM78] DeMarco, T. (1978): *Structured analysis and system specification*, New York: Yourdon 1978.

[Den91] Denert, E. (1991): *Software-Engineering*, 1. korr. Nachdruck 1992, Berlin: Springer 1991.

[Den01] Denert, E. (2001): *persönliches Gespräch mit Prof. Dr. Enrst Denert*, München, 5. Juli 2001.

[Dij76] Dijkstra, E. W. (1976): *A discipline of programming*, Englewood Cliffs, NJ: Prentice Hall 1976.

[DRK76] DeRemer, F.; Kron, H.H. (1976): *Programming-in-the-small versus Programming-in-the-large*, in: IEEE Transactions on Software Engineering, 1976, 2(2), S. 80-86.

[DSW98] D'Souza, D.F.; Wills, A.C. (1998): *Objects, components, and frameworks with UML: the Catalysis approach*, Reading, MA: Addison-Wesley 1998.

[DvK95] Dusink, L.; van Katwijk, J. (1995): *Reuse dimensions*, in: Proc. SSR Symposium on Software Reusability, 1995, S. 137-149.

[Eis88] Eisenhardt, K.; Bourgeois, I.J. (1988): *Politics of strategic decision making in high velocity environments: Toward a mid range theory*, in: Academy of Management Journal, 1988, 31, S. 737-770.

[Eis89] Eisenhardt, K.M. (1989): *Building theories from case study research*, in: Academy of Management Review, 1989, 4, S. 532-550.

[End01] Endres, A. (2001): *Der Informatikermangel und seine Folgen*, in: Informatik-Spektrum, 2001, 24/3, S. 148-152.

[Erd99] Erdogmus, H. (1999): *Comparative evaluation of software development strategies based on net present value*, in: Proc.

EDSER 1 – First Workshop on Economics-Driven Software Engineering Research, 1999.

[Faf94] Fafchamps, D. (1994): *Organizational factors and reuse*, in: IEEE Software, 1994, September, S. 31-41.

[Fav99] Favaro, J. (1999): *Strategic analysis of component-based development*, Austin, Texas: 9th Workshop in Institutionalizing Software Reuse WISR 1999.

[FFa99] Favaro, J.M.; Favaro, K.K. (1999): *Strategic analysis of application framework investments*, in: Fayad, M.E.; Schmidt, D.C.; Johnson, R.E. (Hrsg.): Building application frameworks: object-oriented foundations of framework design, New York: Wiley 1999, S. 567-596.

[FFF97] Favaro, J.M.; Favaro, K.R.; Favaro, P.F. (1997): *Value based software reuse investment*, in: Annals of Software Engineering, 1998, 5, S. 5-52.

[FFi01] Farbey, B.; Finkelstein, A. (2001): *Evaluation in software engineering: ROI, but more than ROI*, in: Proc. EDSER 3 – Third Workshop on Economics-Driven Software Engineering Research, 2001.

[FHB00] Fayad, M.E.; Hamu, D.S.; Brugali, D. (2000): *Enterprise Frameworks: characteristics, criteria, and challenges*, in: Communications of the ACM, 2000, Vol. 43, No. 10 (October), S. 39-46.

[Fis98] Fischer, T. (1998): *Client-DV-Konzept: Anwendungskern am Client (interne Vorabversion)*, München: Internes Dokument sd&m AG 1998.

[FKP99] Fahrmeir, L.; Künstler, R.; Pigeot, I.; Tutz, G. (1999): *Statistik: der Weg zur Datenanalyse*, Berlin: Springer 1999.

[Fow99] Fowler, M. (1999): *Analysemuster*, Bonn: Addison-Wesley Longman 1999.

[FPG94] Fenton, N.; Pfleeger, S.L.; Glass, R. (1994): *Science and substance: a challenge to software engineers*, in: IEEE Software, 1994, 11 (4), S. 86-95.

[Fra94] Frakes, W.; Isoda, S. (1994): *Success factors of systematic reuse*, in: IEEE Software, 1994, September, S. 14-19.

[Frc95] Frick, A. (1995): *Der Software-Entwicklungsprozeß – ganzheitliche Sicht*, München: Hanser 1995.

[FrF95] Frakes, W.; Fox, C. (1995): *Sixteen questions about software reuse*, in: Communications of the ACM, 1995, 38(6), S. 75-87, 112.

[Fri94] Friedman, J.P. (1994) (Hrsg.): *Dictionary of business terms*, Hauppauge, NY: Barron's 1994.

[FrP94] Frakes, W.B.; Pole, T.P. (1994): *An empirical study of representation methods for reusable software components*, in:

IEEE Transactions on Software Engineering, 1994, August, 20(8), S. 617-30.

[FSJ99] Fayad, M.E.; Schmidt, D.C.; Johnson, R.E. (1999) (Hrsg.): *Building application frameworks: object-oriented foundations of framework design*, New York: Wiley 1999.

[FSS99] Funke, C.; Sauder, A.; Schmitgen, S.; Stützle, R. (1999): *Impact of the e-business craze on the enterprise applications software industry*, Internes Dokument: McKinsey&Company, Inc. 1999.

[GAO95] Garlan, D.; Allen, R.; Ockerbloom, J. (1995): *Architectural mismatch: why reuse is so hard*, in: IEEE Software, 1995, 12(6), S. 17-26.

[GBS01] van Gurp, J.; Bosch, J.; Svahnberg, M. (2001): *On the notion of variability in software product lines*, in: Proc. WICSA Working IEEE/IFIP Conference on Software Architecture, 2001.

[GHJ94] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (1994): *Design patterns: elements of reusable object-oriented software*, Reading, Mass.: Addison-Wesley 1994.

[Gol95] Goldberg, A.; Rubin, K.S. (1995): *Succeeding with objects*, Reading, Ma.: Addison-Wesley 1995.

[Grf98] Griffel, F. (1998): *Componentware*, Heidelberg: dpunkt-Verlag 1998.

[Gri94] Griss, M.L. (1994): *Software Reuse Experience at Hewlett-Packard*, in: Proc. ICSE international conference on software engineering, 1994, S. 270.

[Gri98] Griss, M.L. (1998): *Architecting for large-scale systematic component reuse*, Santa Barbara, CA: TOOLS conference – Technology of object-oriented languages and systems 1998.

[Gri99] Griss, M.L. (1999): *Reuse 2001-2004: What next, now that we have solved all reuse problems?*, Austin, Texas: 9th Workshop in Institutionalizing Software Reuse WISR 1999

[GrW95] Griss, M.; Wosser, M. (1995): *Making reuse work at Hewlett-Packard*, in: IEEE Software, 1995, Januar, S. 105-107.

[Gur00] van Gurp, J. (2000): *Variability in software systems: the key to software reuse*, Blekinge Institute of Technology: Licentiate thesis 2000.

[Gut77] Guttag, J. (1977): *Abstract data types and the development of data structures*, in: Communications of the ACM, 1977, Vol. 20 (6), S. 396-404.

[HaH67] Hammersley, J.M.; Handscomb, D.C. (1967): *Monte Carlo Methods*, Norwich: Fletcher & Son 1967.

[Han01] Harrison, Wa. (2001): *What does a software engineer need to know about economics and finace and why*, in: Proc. EDSER 3 – Third

Workshop on Economics-Driven Software Engineering Research, 2001.

[HaP97] Hallsteinsen, S.; Paci, M. (1997) (Hrsg.): *Experiences in software evolution and reuse*, Berlin: Springer 1997.

[Har01] Harrison, Wi. (2001): *Composition and multiple-inheritance in OO design (where in the madness is the method?)*, in: Proc. OOPSLA Workshop on Advanced Separation of Concerns, 2001.

[HaU01] Hanenberg, S.; Unland, R. (2001): *Using and reusing aspects in AspectJ*, in: Proc. OOPSLA Workshop on Advanced Separation of Concerns, 2001.

[HaV99] Hawawini, G.; Viallet, C. (1999): *Finance for executives*, Cincinnati: South-Western College Publishing 1999.

[HeB01] Heiman, R.V.; Byron, D. (2001): *IDC's 2001 software market taxonomy*, Document No. 23998: IDC, www.idc.com 2001.

[Hei95] Heinrich, L. (1995): *Ergebnisse empirischer Forschung*, in: Wirtschaftsinformatik, 1995, 37, S. 3-9.

[Hel01] Helfrich, C. (2001): *Praktisches Prozess-Management*, München: Hanser 2001.

[Hen94] Henninger, S. (1994): *Using iterative refinement to find reusable software*, in: IEEE Software, 1994, September, S. 48-59.

[HMS92] Hering, E.; Martin, R.; Stohrer, M. (1992): *Physik für Ingenieure*, 4. Auflage, Düsseldorf: VDI Verlag 1992.

[Her68] Hertz, D.B. (1968): *Investment policies that pay off*, in: Harvard Business Review, 1968, January-February, S. 96-107.

[HeS99] Herzum, P.; Sims, O. (1999): *Business Component Factory*, New York: Wiley Computer Publishing 1999.

[Hey00] Heymer, V. (2000): *Beratungsmuster Anwendungslandschaft*, Internes Dokument: sd&m AG 2000.

[Hoc99] Hoch, D.J.; Lindner, S.; Purkert, G.; Roeding, C. (1999): *Secrets of software success*, Boston, Massachusetts: Harvard Business School Press 1999.

[Hop00] Hopkins, J. (2000): *Component primer*, in: Communications of the ACM, 2000, Vol. 43, No. 10 (October), S. 27-30.

[HöS99] Hömberg, H.; Schauer, U. (1999): *Wiederverwendung oder Wiedererfindung? – Wiederverwendung aus der Sicht des Wiederverwenders*, in: sTeam (Mitarbeiterzeitschrift der sd&m AG), 1999, 3, S. 52

[Hun01] Hungenberg, H. (2001): *Strategisches Management in Unternehmen*, 2. Auflage, Wiesbaden: Gabler 2001.

[Ill99] Illback, J.H. (1999): Software reuse: data conversion experiences and issues, in: Proc. SSR Symposium on Software Reusability, 1999, S. 10-16.

[JGJ97] Jacobson, I.; Griss, M.; Jonsson, P. (1997): *Software Reuse*, New York, NY: ACM Press/Addison-Wesley 1997.

[JGJ97a] Jacobson, I.; Griss, M.; Jonsson, P. (1997a): *Making the reuse business work*, in: IEEE Computer, 1997, October, S. 36-42.

[JoF88] Johnson, R.E.; Foote, B. (1988): *Designing reusable classes*, in: Journal of Object-Oriented Programming, 1988, 1(2), S. 22-35.

[Joh01] Johnson, R. (2001): *Introduction to 'On the design and development of program families'*, in: Hoffman, D.M.; Weiss, D.M. (Hrsg.): Software Fundementals: Collected papers by David L. Parnas, 2001, S. 193-194.

[Jon94] Jones, C. (1994): *Economics of software reuse*, in: IEEE Computer, 1994, July, S. 106-107.

[Joo94] Joos, R. (1994): *Software reuse at Motorola*, in: IEEE Software, 1994, September, S. 42-47.

[Kaf98] Kafka, F. (1998): *Der Prozeß*, München: DTV 1998.

[Kar95] Karlsson, E.-A. (1995) (Hrsg.): *Software reuse: a holistic approach*, Chichester: Wiley 1995.

[Kar98] Karlsson, E.-A. (1999): *Experiences from an industrial software architecture course*, Austin, Texas: 9th Workshop in Institutionalizing Software Reuse WISR 1999.

[KaW86] Kalos, M.H.; Whitlock, P.A. (1986): *Monte Carlo methods*, Volume I: Basics, New York: Wiley 1986.

[Kel97] Keller, W. (1997): *Mapping objects to tables: a pattern language*, in: Proceedings of the 1997 European Pattern Languages of Programming Conference.

[KeM99] Keepence, B.; Mannion, M. (1999): *Using patterns to model variability in product families*, in: IEEE Software, 1999, July/August, S. 102-108.

[KHH01] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W.G. (2001): *An overview of AspectJ*, in: Proc. ECOOP Aspect-Oriented Programming Workshop, 2001, S. .

[KHL98] Krey, S.; Harnack, J.; Levin, C. (1998): *Management Data Base: Entwicklungsumgebung*, München: Projektdokument sd&m AG 1998.

[KIL96] Kiczales, G.; Irwin, J.; Lamping, J.; Loingtier, J.-M.; Videria Lopes, C.; Maeda, C.; Mendhekar, A. (1996): *Aspect-oriented programming*, in: ACM Computing Surveys, 1996, 28 (4).

[Kin99] Kind, I.; Kind, R. (1999): *Babel: a glossary of computer oriented abbreviations and acronyms*, http://www.geocities.com/ikind_babel/ babel/babel.html 1999.

[KLM97] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Videira Lopes, C.; Loingtier, J.-M.; Irwin, J. (1997): *Aspect-oriented*

programming, in: Proc. ECOOP Aspect-Oriented Programming Workshop, 1997.

[Koh72] Kohlas, J. (1972): *Monte-Carlo-Simulation im Operations Research*, Berlin: Springer 1972.

[Kre99] Krey, S. (1999): *The making of software in re-use*, in: sTeam (Mitarbeiterzeitschrift der sd&m AG), 1999, 3, S. 50.

[Krg92] Krueger, C.W. (1992): *Software reuse*, in: ACM Computing Surveys, 1992, 24 (2), S. 131-83.

[KRi88] Kernighan, B.W.; Ritchie, D.M. (1988): *The C programming language*, second edition, Englewood Cliffs, NJ: Prentice Hall 1988.

[KrL98a] Krey, S.; Levin, C. (1998): *Data Warehouse Loader: Systemkonzept*, Version 1.0, München: Projektdokument sd&m AG 1998.

[KrL98b] Krey, S.; Levin, C. (1998): *Data Warehouse Loader: Programmierer-Handbuch*, Version 1.0, München: Projektdokument sd&m AG 1998.

[Kru95] Kruchten, P. (1995): *Architectural blueprints – the '4+1' view model of software architecture*, in: IEEE Software, 1995, 12 (6), S. 42-50.

[KRW01] Kamm, C.; Reine, F.; Wördehoff, H. (2001): *Basisarchitektur E-Business*, in: Jahrestagung E-Business-Engineering von GI und OCG, Tagungsband, 2001, II, S. 683ff.

[KRy97] Karlsson, J.; Ryan, K. (1997): *A cost-value approach for prioritizing requirements*, in: IEEE Software, 1997, Sept., S. 67-74

[Lat99] Latour, L. (1999): *Reuse education is not about reuse*, in: Proc. WISR Workshop on Institutionalizing Software Reuse, 1999.

[LEO02] *Deutsch-englisches Wörterbuch des LEO Online-Service der Informatik der Technischen Universität München*, http://dict.leo.org, 2002.

[Lim94] Lim, W. (1994): *Effects of reuse on quality, productivity, and economics*, in: IEEE Software, 1994, September, S. 23-30.

[Lim98] Lim, W. (1998): *Managing software reuse*, Upper Saddle River: Prentice-Hall 1998.

[Lim98a] Lim, W.C. (1998): *Strategy-driven reuse: bringing reuse from the engineering department to the executive boardroom*, in: Annals of Software Engineering, 1998, 5, S. 85-103.

[Lut99] Lutz, R.R. (1999): *Toward safe reuse of product family specifications*, in: Proc. SSR Symposium on Software Reusability, 1999, S. 17-26.

[MaR92] Margono, J.; Rhoads, T.E. (1992): *Software reuse economics: cost-benefit analysis on a large-scale Ada project*, in: Proc. ICSE

international conference on software engineering, 1992, S. 338-348.

[MaW93] Malan, R.; Wentzel, K. (1993): *Economics of software reuse revisited*, Palo Alto, CA: Hewlett-Packard Laboratories, Technical Report HPL-93-31, April 1993.

[McC97] McClure, C. (1997): *Software reuse techniques*, Upper Saddle River: Prentice-Hall 1997.

[MCD00] Monday, P.; Carey, J.; Dangler, M. (2000): *SanFrancisco component framework*: an introduction, Reading, Mass.: Addison-Wesley 2000.

[McI68] McIlroy, M. D. (1968): *Mass produced software components*, in: Naur, P.; Randell, B. (Hrsg.): Software Engineering; Report on a conference by the NATO Science Committee (Garmisch, Germany, Oct.), 1968, S. 138-150.

[MCo96] McConnell, S. (1996): *Rapid Development*, Redmond, Washington: Microsoft Press 1996.

[MCo98] McConnell, S. (1998): *Software project survival guide*, Redmond, Wash.: Microsoft Press 1998.

[MET99] Morisio, M.; Ezran, M.; Tully, C. (1999): *Introducing reuse in companies: a survey of European experiences*, in: Proc. SSR Symposium on Software Reusability, 1999, S. 3-9.

[MeL95] Meyer, M.H.; Lopez, L. (1995): *Technology strategy in a software products company*, in: Journal of Product Innovation Management, 1995, Volume 12, Summer, S. 294-306.

[Mey97] Meyer, M. (1997): *Ringvorlesung Architektur: Einführung und Überblick*, Internes Dokument: sd&m AG 1997.

[MFG00] Mili, A.; Fowler Chmiel, S.; Gottumkkala, R.; Zhang, L. (2000): *An integrated cost model for software reuse*, in: Proc. ICSE international conference on software engineering, 2000.

[MiQ96] Mintzberg, H.; Quinn, J.B. (1996): *The strategy process*, 3rd edition, Upper Saddle River: Prentice-Hall 1996.

[Mon01] Monson-Haefel, R. (2001): *Enterprise Java Beans*, third edition, Sebastopol, CA: O'Reilly 2001.

[MSG96] Macala, R.; Stuckey, L. Jr.; Gross, D. (1996): *Managing domain-specific, product-line development*, in: IEEE Software, Vol. 1996, May, p. 57-67.

[MSR99] Morisio, M.; Stamelos, I.; Romano, D.; Moiso, C. (1999): *Framework based software development: learning as an investment factor*, in: Proc. EDSER 1 – First Workshop on Economics-Driven Software Engineering Research, 1999.

[NEC01] NEC Research Institute (2001): *ResearchIndex – the NEC scientific digital library*, NEC corporation: http://citeseer.nj.nec.com/cs 2001.

[Neu01] Neufville, R. de (2001): *Real options: dealing with uncertainty in systems planning and design*, in: 5th International Conference on Technology Policy and Innovation, Delft, Netherlands, 2001, June 29

[NoK01] Noda, N.; Kishi, T. (2001): *Implementing design patterns using advanced separation of concerns*, in: Proc. OOPSLA Workshop on Advanced Separation of Concerns, 2001.

[OhH01] Oh, E.; van der Hoek, A. (2001): *Challenges in using an economic cost model for software engineering simulation*, in: Proc. EDSER 3 – Third Workshop on Economics-Driven Software Engineering Research, 2001.

[Orl01] Orleans, D. (2001): *Separating behavioral concerns with predicate dispatch – or: If statement considered harmful*, in: Proc. OOPSLA Workshop on Advanced Separation of Concerns, 2001.

[OsT01] Ossher, H.; Tarr, P. (2001): *Using multidimensional separation of concerns to (re)shape evolving software: simplifying development, evolution, and integration of Java software using Hyper/J*, in: Communications of the ACM, 2001, 44/10, S. 43-50.

[PaC85] Parnas, D.L.; Clements, P.C. (1985): *A rational design process: how and why to fake it*, Berlin, 25.-29. März 1985: TAPSOFT Joint Conference on Theory and Practice of Software Development 1985.

[Pad99] Padberg, F. (1999): *A fresh look at cost estimation, process models and risk analysis*, in: Proc. EDSER 1 – First Workshop on Economics-Driven Software Engineering Research, 1999.

[Par72] Parnas, D.L. (1972): *On the criteria to be used in decomposing a system into modules*, in: Communications of the ACM, 1972, Vol. 15, No. 12 (December), S. 1053-8.

[Par76] Parnas, D.L. (1976): *On the design and development of program families*, in: IEEE Transactions on Software Engineering, 1976, SE-2, No. 1, S. 1-9.

[Par79] Parnas, D.L. (1979): *Designing software for ease of extension and contraction*, in: IEEE Transactions on Software Engineering, 1979, SE-5, No. 2 (March), S. 128-38.

[Par85] Parnas, D.L. (1985): *The modular structure of complex systems*, in: IEEE Transactions on Software Engineering, 1985, SE-11, No. 3/March, S. 259-266.

[Par85a] Parnas, D.L. (1985): *Software aspects of strategic defense systems*, in: Communications of the ACM, 1985, Vol. 28, No. 12, S. 1326-35.

[Par94] Parnas, D.L. (1994): *Einführung zu 'On the criteria to be used in decomposing systems into modules'*, in: Laplante, P. (Hrsg.): Great papers in computer science, 1996, S. 433.

[PCW83] Parnas, D.L.; Clements, P.C.; Weiss, D.M. (1983): *Enhancing reusability with information hiding*, Stratford, CT: Proceedings of ITT Workshop on Reusability in Programming, September 7-9, 1983.

[Pfl96] Pfleeger, S.L. (1996): *Measuring reuse: a cautionary tale*, in: IEEE Software, 1996, Juli, S. 118-27.

[Pfl01] Pfleeger, S.L. (2001): *Software engineering: theory and practice*, Upper Saddler River: Prentice-Hall 2001.

[PoC93] Poulin, J.S.; Caruso, J.M. (1993): *Determining the value of a corporate reuse program*, in: Proc. IEEE Computer Society Software Metrics Symposium, Baltimore, MD, 1993, 21-22 May, S. 16-27.

[Por96] Porter, M.E. (1996): *What is strategy?*, in: Harvard Business Review, 1996, November-December, S. 61-78.

[Por99] Porter, M.E. (1999): *Wettbewerbsstrategie*, 10. Auflage, Frankfurt/Main: Campus 1999.

[Pou97] Poulin, J.S. (1997): *Measuring software reuse*, Reading, MA: Addison-Wesley 1997.

[Pou99] Poulin, J.S. (1999): *The foundation of reuse*, in: Proc. WISR Workshop on Institutionalizing Software Reuse, 1999.

[Pre97] Pressman, R. (1997): *Software Engineering*, international edition, 4. Auflage, New York: McGraw-Hill 1997.

[Pri99] Price, E. (1999): *Production models drive reuse readiness*, in: Proc. WISR Workshop on Institutionalizing Software Reuse, 1999.

[Ran99] Ran, A. (1999): *Software isn't built from Lego blocks*, in: Proc. SSR Symposium on Software Reusability, 1999, S. 164-169.

[Rei97] Reifer, D.J. (1997): *Practical software reuse*, New York: Wiley Computer Publishing 1997.

[Rid01] Ridder, H. (2001): *WebLog: Projektdarstellung/Einsatz des Quasar Database Interface (QDI)*, Ratingen: Internes Dokument sd&m AG 2001.

[RiU01a] Ridder, H.; Unland, L. (2001): *WebLog: Pflichtenheft/ Entwicklerdokumentation*, München: Internes Dokument sd&m AG 2001.

[RiU01b] Ridder, H.; Unland, L. (2001): *Lastenheft WebLog*, München: Projektdokument sd&m AG 2001.

[Rom99] Roman, E. (1999): *Mastering Enterprise JavaBeans and the Java2 platform, Enterprise Edition*, New York: Wiley 1999.

[Sal98] Salzberger, T. (1998): *Client-DV-Konzept: Grobüberblick (interne Vorabversion)*, München: Internes Dokument sd&m AG 1998.

[Sam97] Sametinger, J. (1997): *Software engineering with reusable components*, Berlin: Springer 1997.

[SBF96] Sparks, S.; Benner, K.; Faris, C. (1996): *Managing object-oriented framework reuse*, in: IEEE Computer, 1996, September, S. 53-61

[Sci99] Schmid, K. (1999): *An economic perspective on product line software development*, in: Proc. EDSER 1 – First Workshop on Economics-Driven Software Engineering Research, 1999.

[SDD99] Sitaraman, M.; Davis, M.; Devanbu, P.; Poulin, J.; Ran, A.; Weide, B. (1999): *Reuse research: contributions, problems and non-problems*, in: Proc. SSR Symposium on Software Reusability, 1999, S. 178-180.

[SGB01] Svahnberg, M.; van Gurp, J.; Bosch, J. (2001): *A taxonomy of variability realization techniques*, http://www.xs4all.nl/~jgurp/publications/Variability_taxonomy.pdf: submitted for publication 2001.

[She97] Scheer, A.-W. (1997): *Wirtschaftsinformatik: Referenzmodelle für industrielle Geschäftsprozesse*, 7. Auflage, Berlin: Springer 1997.

[Shm99] Schmidt, D.C. (1999): *Why software reuse has failed and how to make it work for you*, in: C++ Report Magazine, 1999, January.

[Shö98a] Schöckle, M. (1998): *Client-DV-Konzept: Dialogablaufsteuerung (interne Vorabversion)*, München: Internes Dokument sd&m AG 1998.

[Shö98b] Schöckle, M. (1998): *Client-DV-Konzept: Querschnittsfunktionen am Client (interne Vorabversion)*, München: Internes Dokument sd&m AG 1998.

[Shö98c] Schöckle, M. (1998): *Client-DV-Konzept: Benutzerschnittstelle (interne Vorabversion)*, München: Internes Dokument sd&m AG 1998.

[ShG96] Shaw, M.; Garlan, D. (1996): *Software architecture: perspectives on an emerging discipline*, Upper Saddle River: Prentice-Hall 1996.

[SiD00] Siedersleben, J.; Denert, E. (2000): *Wie baut man Informationssysteme? Überlegungen zur Standardarchitektur*, in: Informatik-Spektrum, 2000, Band 23 Heft 4 (August), S. 247-57.

[Sie96] Siedersleben, J. (1996): *Die wirklichen Schwierigkeiten liegen nicht in der Technik – Wiederverwendung objektorientierter Software*, in: Computerwoche Extra, 1996, 1, S. 12-15, 43.

[Sie00] Siedersleben, J. (2000): *Ringvorlesung Softwarearchitektur*, Internes Dokument: sd&m AG 2000.

[Sie01] Siedersleben, J. (2002): *Architekturbegriffe für betriebliche Informationssysteme*, zur Veröffentlichung eingereicht 2002.

[Sie01a] Siedersleben, J. (2001): *Sidestep: Die Informatik-Initiative von sd&m*, 7. Workshop SEUH (Software Engineering im Unterricht der Hochschulen) 2001.

[Sie01b] Siedersleben, J.: *Persönliches Gespräch*, München 2001

[SiF01] Siedersleben, J.; Foitzik, T. (2001): *Was ist Quasar?*, München: Internes Dokument sd&m Research GmbH 2001.

[SiS02] Siedersleben, J.; Stützle, R. (2002): *Software categories: designing software for change*, Arbeitspapier sd&m Research; zur Veröffentlichung vorgesehen 2002.

[Sny86] Snyder, A. (1986): *Encapsulation and inheritance in object-oriented programming languages*, in: Proc. OOPSLA Conference on Object-Oriented Programming Systems, Languages, and Applications, 1986, S. 38-45.

[Sob94] Sobol, I.M. (1994): *A primer for the Monte Carlo method*, Boca Raton: CRC Press 1994.

[Som96] Sommerville, I. (1996): *Software Engineering*, 5th edition, Reading, Mass.: Addison-Wesley 1996.

[SoS99] Sodhi, J.; Sodhi, P. (1999): *Software reuse: domain analysis and design process*, New York: McGraw-Hill 1999.

[Spi90] Spiegel, M.R. (1990): *Statistik*, Hamburg: McGraw-Hill 1990.

[SSR02] Schmidt, D.; Stal, M.; Rohnert, H.; Buschmann, F. (2002): *Pattern-orientierte Software-Architektur: Muster für nebenläufige und vernetzte Objekte*, Heidelberg: dpunkt-Verlag 2002.

[Ste01] Steger, A. (2001): *Diskrete Strukturen 1: Kombinatorik, Graphentheorie, Algebra*, Berlin: Springer 2001.

[StS00] Stützle, R.; Siedersleben, J. (2000): *Erfolgsfaktoren für die Entwicklung wiederverwendbarer Software*, in: Computerwoche Extra, 2000, 8, S. 38-39.

[Stü99] Stützle, R. (1999): *Wesentliche Unterschiede zwischen Produkt- und Projektgeschäft in der Softwareindurstrie*, Internes Dokument: McKinsey&Company, Inc. 1999.

[Stü00a] Stützle, R. (2000): *Buchbesprechung: Jacobson/Griss/Jonsson, Software Reuse*, in: sTeam (Mitarbeiterzeitschrift der sd&m AG), 2000, 3, S. 38.

[Stü00b] Stützle, R. (2000): *Buchbesprechung: Franz Kafka, Der Prozeß*, in: sTeam (Mitarbeiterzeitschrift der sd&m AG), 2000, 3, S. 39.

[Stü01] Stützle, R. (2001): *Kontrollierte Variabilität*, Beitrag zum Jahresbericht 2000/2001: sd&m Research GmbH, München 2001.

[SuC01] Sullivan, K.; Cai, Y.; Hallen, B.; Griswold, W.G. (2001): *The structure and value of modularity in software design*, in: Proc. EDSER 3 – Third Workshop on Economics-Driven Software Engineering Research, 2001.

[SuR01] Sutton, S.M.; Rouvellou, I. (2001): *Applicability of categorization theory to multidimensional separation of concerns*, in: Proc. OOPSLA Workshop on advanced separation of concerns, 2001.

[sww] Website der sd&m AG: http://www.sdm.de

[Szy99] Szyperski, C. (1999): *Component Software*, New York: ACM Press 1999.

[Tau01] Taubner, D. (2001): *Software-Entwicklung im industriellen Maßstab*, in: Desel, J. (Hrsg.): Das ist Informatik, Berlin: Springer 2001, S. 85-104.

[TOH99] Tarr, P.; Ossher, H.; Harrison, W. (1999): *N degrees of separation: multi-dimensional separation of concerns*, in: Proc. ICSE international conference on software engineering, 1999, S. 107-119.

[Tra95] Tracz, W. (1995): *Confessions of a used program salesman*, Reading, Mass.: Addison-Wesley 1995.

[Vet98] Vetter, M. (1998): *Aufbau betrieblicher Informationssysteme*, 8. Auflage, Stuttgart: Teubner 1998.

[Vir01] Virtamo, J. (2001): *Queueing theory*, Lecture notes: Helsinki University of Techology 2001.

[WeA01] Welge, M.K.; Al-Laham, A. (2001): *Strategisches Management*, 3. Auflage, Wiesbaden: Gabler 2001.

[WeL99] Weiss, D.M.; Lai, C.T.R. (1999): *Software Product-Line Engineering*, Reading, MA: Addison-Wesley 1999.

[WiB98] Wiles, E.; Bott, F. (1998): *Eight steps to your own economic model of software reuse*, in: Proc. European Reuse Workshop, 1998, European Software Institute, S. 123 - 134.

[Wil99] Wiles, E. (1999): *Economic models of software reuse: a survey, comparison and partial validation*, Technical Report Nr. UWA-DCS-99-032: Department of Computer Science, University of Wales 1999.

[Wir71] Wirth, N. (1971): *Program development by stepwise refinement*, in: Communications of the ACM, 1971, 14 (4), S. 221-227.

[Wit96] Withey, J. (1996): *Investment analysis of software assets for product lines*, Technical Report CMU/SEI-96-TR-010: Carnegie-Mellon University 1996.

[Wöh96] Wöhe, G. (1996): *Einführung in die allgemeine Betriebswirtschaftslehre*, 19. Auflage, München: Vahlen 1996.

[WöS00] Wördehoff, H.; Spanner, C. (2000): *GO Java-Framework 2.0: Systemdokumentation*, München: Projektdokument sd&m AG 2000.

[Wol02] Wolfe, J. (2002): *IBM San Francisco: lots of promise, but little payoff*, in: The Software Practitioner, 2002, 12(1), S. 1, 5.

[Wri01] Wright, J.F. (2001): *Monte Carlo risk analysis*, www-Artikel: http://www.drjfwright.com/d/tramc.html 2001.

[www2] http://www.sternstewart.com

[www3] http://nts4.oec.uni-osnabrueck.de/absatz/download/jurdef.pdf

[www5] http://api.hq.faa.gov/apo3/tofc.htm

[Yin90] Yin, R (1990): *Case study research: design and methods*, Fifth printing/revised edition, Newbury Park: Sage 1990.

[You93] Yourdon, E. (1993): *Die westliche Programmierkunst am Scheideweg*, München, Wien: Hanser 1993.

[You02] Yourdon, E. (2002): *Notes on 'The Mythical Man-Month anniversary edition*, http://www.yourdon.com/books/coolbooks/notes/brooks.html 2002.

[ZAD97] Zand, M.; Arango, G.; Davis, M.; Johnson, R.; Poulin, J.S.; Watson, A. (1997): *Reuse R&D: is it on the right track*, in: Proc. SSR Symposium on Software Reusability, 1997, S. 212-216.

[Zan99] Zand, M. (1999): *Organizational and management issues: are we there yet?*, in: Proc. WISR Workshop on Institutionalizing Software Reuse, 1999.

[ZBB99] Zand, M.; Basili, V.; Baxter, I.; Griss, M.; Karlsson, E.-A.; Perry, D. (1999): *Reuse R&D: gap between theory and practice*, in: Proc. SSR Symposium on Software Reusability, 1999, S. 172-177.

[ZBe00] Zimmermann, J.; Beneken, G. (2000): *Verteilte Komponenten und Datenbankanbindung*, München: Addison-Wesley 2000.

[ZBP01] Zand, M.; Bassett, P.; Prieto-Diaz, R. (2001): *Reuse: where are we standing? – Can we say 'reuse is dead, long live reuse' or is it too soon?*, in: Proc. SSR Symposium on Software Reusability, 2001, S. 173-175.

[Zim92] Zimmermann, H.J. (1992): *Methoden und Modelle des Operations Research*, 2. Auflage, Braunschweig: Vieweg 1992

[ZME01] Zeh, U.; Mieth, A.; Engelkamp, S.; Lemmer, I. (2001): *QMS Verfahrenshandbuch*, München: sd&m AG 2001.