Spezifikation und Auswahl von Services

Beiträge zur komponenten- und serviceorientierten Entwicklung betrieblicher Anwendungssysteme

Dissertation
zur Erlangung des Doktorgrades an der
Wirtschaftswissenschaftlichen Fakultät
der Universität Augsburg

vorgelegt von

Oliver Skroch

2009

Erstgutachter:	Professor Dr. Klaus Turowski
Zweitgutachter:	Professor Dr. Robert Klein
Vorsitzender der mündlichen Prüfung:	Professor Dr. Axel Tuma
Tag der mündlichen Prüfung:	14. Dezember 2009

Inhaltsverzeichnis

Anmerkung:
Die Literaturverzeichnisse finden sich am Ende eines jeden Beitrags bzw. Kapitels.

Verzeichnis der Beiträge

In dieser Dissertationsschrift werden die folgenden, jeweils nach wissenschaftlichen Begutachtungsverfahren veröffentlichten oder zur Veröffentlichung angenommenen, oder zu wissenschaftlichen Begutachtungsverfahren eingereichten Beiträge vorgestellt:

B1: Veröffentlichter Beitrag der Kategorie B
Skroch, O.; Turowski, K. (2009), „A theory of software reuse strategies in ideal type stable and turbulent market environments“, *Proceedings of the Fifteenth Americas Conference on Information Systems*, Association for Information Systems, 6.-9. August 2009, San Francisco, USA: #272.

B2: Veröffentlichter Beitrag der Kategorie B
Skroch, O. (2006), „Integration assessment of an individually developed application vs. software packages from the market – an experience report“, *Integration, Informationslogistik und Architektur: Tagungsband DW 2006*, Lecture Notes in Informatics P-90, Gesellschaft für Informatik, 21.-22. September 2006, Friedrichshafen: 329-340.

B3: Angenommener Beitrag der Kategorie B
Pruß, M.; Skroch, O. (zur Veröffentlichung angenommen am 27. Juli 2009), „Die Bedeutung der Anforderungsspezifikation für erfolgreiche IT-Projekte“, *HMD – Praxis der Wirtschaftsinformatik*.

B4: Eingereichter Beitrag
Overhage, S.; Skroch, O.; Turowski, K. (in Begutachtung seit 14. September 2009), „A case study on requirements specifications and critical compensation factors in offshore application development“.

B5: Angenommener Beitrag der Kategorie A
Skroch, O.; Turowski, K. (zur Veröffentlichung angenommen am 30. September 2009), „Optimal stopping for the run-time self-adaptation of software systems“, *Journal of Information and Optimization Sciences*.

B6: Veröffentlichter Beitrag ohne Kategorie
Skroch, O.; Turowski, K. (2007), „Reducing domain level scenarios to test component-based software“, *Journal of Software*, 2 (5): 64-73.

Die zu den Beiträgen vermerkten Ranking-Kategorien bestimmen sich nach der *Zeitschriften und Ranking* Tabelle der wirtschaftswissenschaftlichen Fakultät der Universität Augsburg vom 25. September 2006 und nach den fachspezifischen *WI-Orientie-*

rungslisten in der am 27. Februar 2008 in München von der Wissenschaftlichen Kommission Wirtschaftsinformatik im Verband der Hochschullehrer für Betriebswirtschaft e. V. (WKWI) und dem Fachbereich Wirtschaftsinformatik der Gesellschaft für Informatik e. V. (GI-FB WI) verabschiedeten Fassung.

Für Publikationsorgane, die in mehr als einem Ranking erscheinen, wurde nach dem „best of"-Prinzip jeweils die beste Kategorie eingesetzt.

I Einleitung

I.1 Motivation und Problemstellung

„Better! Cheaper! Faster!“ ist eine herrschende Zielvorgabe im heutigen Wirtschaftsgeschehen, die quer durch viele Branchen von den grünen Tischen der Investoren über die Vorstandsetagen in die operativen Projekte weitergetragen wird (Brandon 2006, S. 4). Die drei grundsätzlichen unternehmerischen Herausforderungen dieser Zielvorgabe sind in der Struktur des in Abbildung I-1 illustrierten „triple constraint“ Dreiecks wiedergegeben (Rosenau 1981, S. 15-18). Die Schwierigkeit liegt dabei in der Dynamik des „triple constraint“, also in der gleichzeitigen Erfüllung aller drei voneinander abhängigen Bedingungen: „Unfortunately, the Triple Constraint is very difficult to satisfy because most of what occurs during a project conspires to pull the performance below specification and to delay the project so it falls behind schedule, which makes it exceed the budget.“ (Rosenau 1981, S. 15).

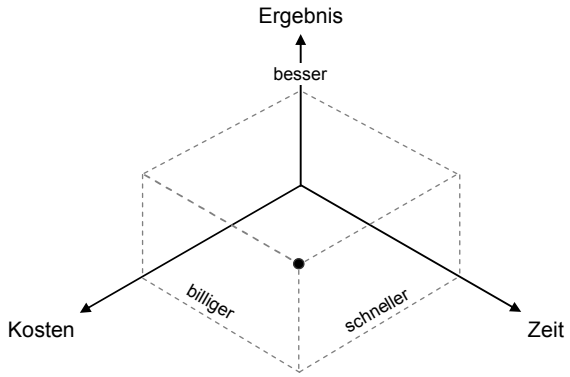

Abbildung I-1: „Triple Constraint“. Vgl. Rosenau (1981, S. 16).

Der aus dem „triple constraint“ erklärbare Wunsch nach besseren, billigeren und schnelleren Ergebnissen hat die IT-Entwicklungsprozesse mit als Erste erreicht (Voas 2001, S. 96; Brandon 2006, S. 4-6), denn die hohe Bedeutung der IT in der Unternehmenswelt macht das sogenannte „business-IT alignment“ – die enge, wechselseitige Abstimmung zwischen geschäftlichen Zielvorgaben und IT-Potenzialen – in vielen Märkten zur notwendigen Voraussetzung wirtschaftlicher Konkurrenz- und Überlebensfähigkeit (Henderson & Venkatraman 1993, S. 476; Teubner 2006, S. 368f).

Unterscheidet man in der IT zwischen Hardware und Software, so fällt zunächst auf, dass bei der Entwicklung von IT-Elektronik und Computer-Hardware schon seit Jahrzehnten extreme und sehr zuverlässig planbare Leistungssteigerungen erzielt werden. Moore (1965, S. 114-117) formulierte schon bald nach der Erfindung der integrierten Halbleiterschaltung den Zusammenhang, dass sich die Komplexität integrierter Schalt-

kreise mit minimalen Komponentenkosten etwa alle ein bis zwei Jahre verdoppelt. Die zur damaligen Zeit fantastisch anmutende und unbewiesene Expertenmeinung des Intel Co-Gründers Moore gilt heute als bestätigt, man spricht sogar vom Mooreschen Gesetz im Sinne einer Gesetzmäßigkeit.

Dagegen fange die Entwicklung von Software typischerweise scheinbar harmlos und einfach an, enttäusche dann aber regelmäßig die – von den Stakeholdern analog zur Hardwareentwicklung entsprechend hoch angesetzten – Erwartungen mit schlechten Ergebnissen, gesprengten Budgets und nicht eingehaltenen Terminen (Brooks 1987, S. 10; DeMarco 1997, S. 1-6; Glass 2006, S. 15). Spätestens seit in den 1960er Jahren die Kosten für Software erstmals die Hardwarekosten übertrafen – und in der Folge der Begriff der Softwarekrise geprägt wurde – ist die Softwareentwicklung dem speziellen Vorwurf ausgesetzt, sie könne nicht die angemessenen Fortschritte erzielen, die den Entwicklungen im Bereich der Elektronik und Computer-Hardware entsprechen (Naur & Randell 1969, S. 13f, S. 65ff; Dijkstra 1972, S. 866). Hierfür sind neben technischen Gründen auch planerisch-organisatorische Ursachen diskutiert worden (Martin & Chang 1994, S. 14f; Glass 1996, S. 183f). Chatzoglou (1997, S. 627) betont beispielsweise „inadequate project management caused by a lack of recognising and understanding what the real problems are in carrying out software development."

Es entstand damit der Wunsch nach neuen Technologien und Management-Methoden, die deutliche Verbesserungen bei der Softwareentwicklung bewirken, um auch dort eine ähnliche Fortschrittsgeschwindigkeit wie bei der Hardwareentwicklung zu erreichen.

Die drei deutlichsten Produktivitätsfortschritte, die in der Geschichte der Softwareentwicklung bisher erzielt wurden, werden von Brooks (1987, S. 12f) geschildert. Durch die *höheren Programmiersprachen* wurden in den 1950er Jahren die Probleme eliminiert, die in der Bauweise des Computers liegen („Fortran" der Firma IBM). Höhere Programmiersprachen abstrahieren Software von grundsätzlichen Eigenschaften der physischen Maschine wie Bit Endians, Register, usw. Sie gelten als größter Fortschritt bisher. Durch *Mehrbenutzerumgebungen* wurden in den 1960er Jahren die Probleme eliminiert, die in der sequenziellen Benutzung des Computers liegen („PDP-6" der Firma DEC). Mehrere Entwickler konnten nun gleichzeitig über Terminals mit einem Computer arbeiten und ihre Programme selbst kompilieren und ausführen. Durch *integrierte Entwicklungsumgebungen* wurden in den 1970er Jahren die Probleme eliminiert, die aus der Integration verschiedener Einzelwerkzeuge in einen Gesamtzusammenhang entstehen, in welchem aus Quellcode eine ausführbare Software erzeugt wird („Interlisp" der Firma Xerox). Entwickler hatten nun ihre vordefinierten Entwicklungs-

umgebungen, ihre „Werkzeugkästen“ mit Editor, Bibliotheksfunktionen, Compiler, Linker, Binder, Debugger, usw. zur Verfügung.

Mit der in diesem Zusammenhang oft diskutierten Objektorientierung (Dahl & Nygaard 1966; Meyer 1990; Rumbaugh et al. 1993) lässt sich Software auf höheren Abstraktionsebenen darstellen. Dabei ist ein wichtiger Vorteil, dass die zusätzlichen, objektorientierten Abstraktionskonzepte die Entwicklung noch stärker systematisieren – sie allerdings auch nicht einfacher machen. Die essentiellen Entwicklungsprobleme bei Software liegen aber nicht in der Darstellung der Lösung, sondern in der Lösung selbst. Durch objektorientierte Methoden wurden daher auch keine erheblichen Produktivitätssteigerungen erzielt (Brooks 1987, S. 14; Potok, Vouk & Rindos 1999, S. 844; Glass 2005, S. 18).

Diese bisher größten Produktivitätssprünge bei der Entwicklung von Software, die in den implementierungsnahen Phasen erreicht worden sind, haben viel dazu beigetragen, die folgenden, essentiell wesenhaften Schwierigkeiten der Disziplin deutlicher erkennbar werden zu lassen (Brooks 1987, S. 11f):

- *Extreme Komplexität*. Es gibt in einer Software, oberhalb der Ebene des Quellcodes, keine zwei Teile, die sich gleichen. Falls doch, so sind sie als dieselbe Komponente, dasselbe Objekt, dasselbe Modul, dieselbe Routine, usw. realisiert. Es gibt keine anderen Systeme außer Software mit dieser Eigenschaft. Software kann darüber hinaus extrem viele Zustände annehmen. Eine Software mit 300 booleschen Variablen gilt als klein, sie kann aber 2^{300} (entspricht ungefähr zweimal 10^{90}) verschiedene Zustände annehmen. Das sind etwa hundert Milliarden mal mehr Systemzustände als die Anzahl der Atome im beobachtbaren Universum, die derzeit in der Größenordnung um 10^{79} geschätzt wird (Wikipedia 2009).
- *Willkürliche Konformität*. Software, und ganz besonders betriebliche Anwendungssoftware, muss an ihren Schnittstellen Konformität zu anderen Softwaresystemen und zu weiteren, ebenfalls von Menschen gestalteten Institutionen, Systemen, Produkten usw. herstellen. Dabei handelt es sich um willkürlich bestimmte Kulturartefakte, die eben nicht allgemein gültigen und stets rationalen Regeln und Gesetzen unterliegen – im Unterschied etwa zur Physik, die es zwar auch mit hoher Komplexität zu tun hat, die es aber insofern einfacher hat, als sie „auf der anderen Seite ihrer Schnittstellen“ auf feste Naturgesetze trifft.
- *Ständige Änderbarkeit*. Im Unterschied zu allen anderen technischen Systemen wird Software regelmäßig auch nach ihrer Inbetriebnahme fundamental verändert. Obwohl der hohe Aufwand für Änderungen an fertig gestellten technischen Produkten in anderen Bereichen verstanden wird, beispielsweise bei Erweiterungen und

Umbauten an Gebäuden, hält man Änderungen an Software für vergleichsweise unproblematisch. Ein Grund mag sein, dass Software nicht greifbar (physisch anfassbar) ist und sich dadurch dem intuitiven Verständnis entzieht.

- *Unsichtbarkeit*. Softwaresysteme können nicht angemessen visualisiert werden, denn für die Realität von Software ist keine einfache Entsprechung im Raum bekannt. Visualisierungsansätze führen, ebenso wie Beschreibungsmodelle ohne grafische Notationen, zu mehrdimensionalen, von einander abhängigen Beschreibungs- und Diagrammebenen, und werfen schwierige Frage nach den genauen Abhängigkeiten und Beziehungen zwischen den einzelnen Beschreibungsebenen und den in ihnen spezifizierten Inhalten auf. Vgl. weiterführend etwa Overhage (2006, S. 125-129).

Die essentiellen Schwierigkeiten bei der Entwicklung von Software machen nicht an den implementierungsnahen Aufgaben halt. Parnas (1985, S. 1327f) beschreibt in diesem Zusammenhang fundamentale Unterschiede zwischen dem Software Engineering und der Entwicklung in anderen Ingenieurwissenschaften, und legt Gründe für die prinzipielle Unzuverlässigkeit von Software dar. Er geht dabei von dem grundsätzlichen Unterschied zwischen analogen und diskreten Systemen aus.

Analoge Systeme haben unendlich viele Zustände (Beispiele sind etwa Lautsprecher, Motoren, Heizungen). Ihr Verhalten kann durch stetige Funktionen angemessen beschrieben werden. Derartige Systeme sind Gegenstand klassischer Ingenieursdisziplinen, die entsprechende Mathematik der stetigen Funktionen wird sehr gut verstanden. Analoge Systeme bergen innerhalb ihres Betriebsbereichs keine versteckten Überraschungen: kleine Eingabeänderungen bedingen immer auch entsprechend kleine Ausgabeänderungen. Zuverlässiges Verhalten analoger Systeme kann garantiert werden durch mathematische Beschreibung und Analyse innerhalb des Betriebsbereichs sowie durch Testen, dass der definierte Betriebsbereich nicht verlassen wird. *Hybride Systeme* bestehen aus Komponenten, die eine kleine Menge von diskreten Zuständen haben und zwischen den verschiedenen Zuständen durch stetige Funktionen beschrieben sind (wie die Diode). *Diskrete Systeme* haben endlich viele Zustände. Ihr Verhalten ist außerhalb der definierten stabilen Zustände unerheblich bzw. nicht definiert. Die ersten diskreten Systeme vor der Zeit heutiger Computer hatten sehr wenige Zustände, konnten daher vollständig getestet werden und wurden so auch ohne analytische Beschreibung komplett verstanden (beispielsweise Bahnstellwerke). Die ersten diskreten Systeme im Computerbereich hatten bereits extrem viele Zustände, bestanden aber aus vielen identischen Kopien sehr weniger untereinander verschiedener Subsystemtypen. Sie konnten deshalb ebenfalls vollständig getestet und damit komplett verstanden werden (zum Beispiel der Halbleiterspeicher).

Eine entscheidende Konsequenz für die Systemplanung und -entwicklung ergibt sich aus der Beschreibbarkeit der Systeme als Modell. Bei analogen Systemen wird dies durch stetige mathematische Funktionen gelöst, die aber auf diskrete Systeme nicht anwendbar sind. Für Softwaresysteme gilt:

- sie sind diskret und können nicht wie traditionelle analoge Technik durch stetige Funktionen und die Methoden der Infinitesimalrechnung beschrieben werden,
- es ist derzeit kein entsprechend geeignetes mathematisches oder logisches Analyse-Instrumentarium zur einfachen Beschreibung diskreter Systeme bekannt,
- und ihr Verhalten kann auch nicht wie bei Computerhardware „brute force“ durch vollständiges Durchtesten verstanden werden, weil sie extrem viele Zustände einnehmen können und keine repetitive Struktur aufweisen, wodurch eine solche Prüfung am Aufwand scheitert.

Parnas (1985, S. 1328) erkennt diese Bedingungen als „fundamental difference that will not disappear with improved technology“. Diese grundlegenden technischen Eigenschaften, die im Wesen von Software liegen, sind bei der strategischen Zielfindung, bei der taktischen Planung und in der operativen Umsetzung der Entwicklung von Softwareanwendungen maßgeblich zu berücksichtigen. Zu den wenigen Fundamentalansätzen, in denen das auch ausdrücklich der Fall ist, die somit am ehesten zur deutlichen Verbesserung der Entwicklung betrieblicher Anwendungssysteme in Richtung der „triple constraint“ Wünsche beitragen können, gehören:

- Die *Ausbildung von Experten* (Brooks 1987, S. 18; Parnas 1985, S. 1328; Wissenschaftliche Kommission Wirtschaftsinformatik 2003). Geht man davon aus, dass die Unterschiede zwischen herausragender und durchschnittlicher Arbeit (nicht nur) in der Softwarebranche etwa eine Zehnerpotenz betragen (Sackman, Erikson & Grant 1968, S. 5f; Boehm 1986, S. 596), so ist das Finden, Ausbilden, Unterstützen und Fördern motivierter und fähiger Personen der wohl am meisten versprechende Ansatz überhaupt.
- Das strategische Konzept der *Wiederverwendung* (Biggerstaff & Richter 1987; Mili, Mili & Mili 1995; Rost 1997). Die „buy versus make“ Lösungsansätze möchten Entwicklungsaufgaben minimieren, indem bereits verfügbare, fertige (Teil-)Lösungen wiederverwendet werden, wo immer das möglich ist. Ein Kernproblem hierbei ist die Einsetzbarkeit fremdbezogener Zwischenergebnisse im neuen Kontext, die derzeit unter anderem mangels allgemein anerkannter Standards in der Softwarebranche nicht ohne weiteres vorausgesetzt werden kann. Poulin (1997, S. 145) stellt daher fest: „to achieve real results, we must institutionalize reuse“, und Mili, Mili

und Mili (1995, S. 529) beschreiben das Konzept der Wiederverwendung sogar als den einzig realistischen Lösungsweg: „That leaves us with software reuse as the only realistic, technically feasible solution: We could reuse the processes and products of previous development efforts in order to develop new applications."

- Die *Komponenten- und Serviceorientierung*. Diese auf dem „divide and conquer" Prinzip basierenden Lösungsansätze möchten eine große Aufgabe so lange in immer kleinere Teile zerlegen, bis die Teilaufgaben einzeln gelöst und wieder zu einer großen, lose gekoppelten Gesamtlösung zusammengesetzt werden können. Ein Kernproblem hierbei ist es, die richtigen Einzelteile (Komponenten bzw. Services) zu finden. Für diese in der Softwareentwicklung schon aus der strukturierten Analyse (als Modulabgrenzung) und aus der Objektorientierung (als Objektfindung) bekannte Frage werden heute bereits systematische, optimierende Verfahren diskutiert. Dabei sind komponenten- und serviceorientierte Lösungsansätze von Anfang an eng mit dem Konzept der Wiederverwendung verzahnt (Neighbors 1984, S. 567f; Sametinger 1997, S. 9ff, S. 67ff). Komponentenorientierung (Wassermann & Gutz 1982; Szyperski 1998; Brown 2000) und Serviceorientierung (Schulte & Natis 1996; Schulte 1996; Atkinson et al. 2002; Fröschle & Reinheimer 2007) unterscheiden sich untereinander vorwiegend im Gegenstand der Wiederverwendung. Im ersten Fall werden Komponenten selbst, im zweiten Fall die durch die Komponenten implementierten Services wiederverwendet.

Die vorliegende Arbeit motiviert sich daher, im strategischen Rahmen der Software-Wiederverwendung, über die zu erwartenden Vorteile des komponenten- und serviceorientierten Entwicklungsparadigmas und möchte in diesem Kontext vorwiegend gestalterische Beiträge zum Fortschritt bei der Entwicklung betrieblicher Anwendungssysteme beisteuern.

I.2 Zielsetzung und fokussierte Forschungsfragen

Als wissenschaftstheoretisch begründete Hauptaufgaben der Wirtschaftsinformatik gelten die Erklärung und die Gestaltung des Untersuchungsgegenstandes, zusätzlich lassen sich als Ergänzungsaufgaben die Beschreibung und die Prognose nennen; der Gestaltungsaufgabe kommt dabei ein besonders hoher Stellenwert zu (Mertens et al. 2005, S. 4f; Heinrich, Heinzl & Roithmayr 2007, S. 21). Ziel der Gestaltungsaufgabe – die auf Beschreibung, Erklärung und Prognose aufbaut – ist es, einen erwünschten Sollzustand zu bewirken.

Das Ziel wissenschaftlicher Untersuchungen in der Wirtschaftsinformatik kann als die Gewinnung von Theorien, Methoden, Werkzeugen und nachprüfbaren Erkenntnissen zu

Mensch-Aufgabe-Technik-Systemen und -Infrastrukturen der Information und Kommunikation in Wirtschaft und Verwaltung beschrieben werden, wobei langfristig die „sinnhafte Vollautomation“ angestrebt wird (Wissenschaftliche Kommission Wirtschaftsinformatik 1994, S. 81; Mertens et al. 2005, S. 4; Heinrich, Heinzl & Roithmayr 2007, S. 16, S. 21). Gerade in der Wirtschaftsinformatik sind zudem praxisorientierte Arbeiten zur Gewinnung und Validierung von Kenntnissen wünschenswert und notwendig (Wissenschaftliche Kommission Wirtschaftsinformatik 1994, S. 81).

Die im Hauptteil dieser Arbeit vorgestellten Beiträge verfolgen praxisrelevante, gestalterische Ziele. Die Beiträge möchten anwendbare wissenschaftliche Erkenntnisse für die praktische Unterstützung von realen Software-Entwicklungsprozessen liefern, und damit die Grundlagen für Planung und Durchführung entsprechender Entwicklungsaufgaben für betriebliche Anwendungssysteme verbessern. In den einzelnen Kapiteln und Beiträgen werden die folgenden spezifischen Ziele verfolgt und die jeweils daraus abgeleiteten, fokussierten Forschungsfragen untersucht:

Kapitel II, Beitrag B1: A theory of software reuse strategies in ideal type stable and turbulent market environments.

Die wiederverwendungsgetriebene Entwicklung von Software geht über die Grenzen traditioneller Entwicklungsprojekte hinaus und bezieht in ihrer Wertschöpfungskette ausdrücklich globale Märkte ein, etwa für Beschaffung und Absatz wiederzuverwendender Artefakte. Marktbedingungen sind daher bei der strategischen Ausrichtung der Wiederverwendung mit zu berücksichtigen. Beitrag B1 verfolgt das Ziel, die strategische Wahl von Wiederverwendungsansätzen zu unterstützen und eine Theorie zu deren Präferenz nach idealtypischen Marktbedingungen vorzuschlagen. Es werden hierzu die folgenden, fokussierten Forschungsfragen untersucht:

- Welche prinzipiellen Wiederverwendungsansätze sind bekannt?
- Welche idealtypischen Marktmilieus lassen sich mit Hinblick auf die wiederverwendungsgetriebene Softwareentwicklung abgrenzen?
- Welche strategischen Präferenzen lassen sich gegebenenfalls für bestimmte Wiederverwendungsansätze in Abhängigkeit von dem jeweiligen Marktumfeld herleiten?

Kapitel II, Beitrag B2: Integration assessment of an individually developed application vs. software packages from the market – an experience report.

Die Frage, ob hochkomplexe betriebliche Anwendungssysteme mit sehr vielen, stark individuellen Anforderungen aus vorgefertigten, am Markt erhältlichen Teilprodukten zusammengesetzt werden sollen, oder doch besser individuell zu entwickeln sind, stellt

sich in der Praxis immer wieder. Beitrag B2 bezweckt, die Strategie der Individualentwicklung im Vergleich zur Beschaffung betrieblicher Anwendungskomponenten am Markt zu untersuchen, wozu Ergebnisse aus entsprechenden Praxisprojekten vorgestellt und analysiert werden. Es werden hierzu die folgenden, fokussierten Forschungsfragen untersucht:

- Wie umfassend werden spezifische Anforderungen bei der Individualentwicklung eines sehr großen und komplexen Anwendungssystems erfüllt?
- Wie umfassend werden diese Anforderungen im Vergleich dazu von einer Kollektion von Softwarepaketen erfüllt, die am Markt angeboten werden?
- Welches Bild ergibt die Analyse von mehreren „make-or-buy"-Referenzprojekten für die Frage, ob Individualentwicklung oder Softwarepakete zu bevorzugen sind?

Kapitel III, Beitrag B3: Die Bedeutung der Anforderungsspezifikation für erfolgreiche IT-Projekte.

Die ohnehin schon anspruchsvolle Anforderungsspezifikation gehört, aufgrund ihrer zentralen Bedeutung in arbeitsteiligen Entwicklungsprozessen, zu den besonders erfolgskritischen Phasen der Anwendungs- und Systementwicklung. Beitrag B3 untersucht Grundlagen für die taktische Planung dieser Phase durch die Darstellung der Erfolgsfaktoren für die Anfertigung von hochwertigen Anforderungsspezifikationen, sowie durch die Beschreibung der Risiken, die aus unzureichenden Spezifikationen erwachsen können. Es werden die folgenden, fokussierten Forschungsfragen untersucht:

- Welche Bedeutung haben Anforderungsspezifikationen in der betrieblichen Praxis der Softwareentwicklung?
- Welche kritischen Erfolgsfaktoren lassen sich feststellen, die die Anfertigung von hochwertigen Leistungsbeschreibungen in der Softwareentwicklung ermöglichen?
- Welche praktischen Konsequenzen können in der arbeitsteiligen Softwareentwicklung durch unzureichende Leistungsbeschreibungen entstehen?

Kapitel III, Beitrag B4: A case study on requirements specifications and critical compensation factors in offshore application development.

Für die global arbeitsteilige Entwicklung betrieblicher Anwendungssoftware mit den typischen Offshore-Anteilen in Design und Programmierung ist die Qualität der zugrunde liegenden Anforderungsspezifikationen von hoher Bedeutung. Beitrag B4 befasst sich anhand einer großen, realen Fallstudie mit der Eignung von Spezifikationen

für die Offshore-Durchführung nachgelagerter Entwicklungsschritte. Es werden die folgenden, fokussierten Forschungsfragen untersucht:

- Wie können Anforderungsspezifikationen systematisch hinsichtlich ihrer Eignung für nachgelagerte Offhore-Entwicklungsschritte beurteilt werden?
- Welche Handlungsoptionen lassen sich gegebenenfalls aus den Ergebnissen einer solchen Beurteilung ableiten?
- Welche Erkenntnisse lassen sich bei der Durchführung einer solchen Beurteilung in einem realen Kontext gewinnen?

Kapitel IV, Beitrag B5: Optimal stopping for the run-time self-adaptation of software systems.

Manche Anwendungssysteme werden bereits heute als Mashups durch frei im Internet verfügbare (Web-)Services ergänzt. Dabei werden Funktionsaufrufe opportunistisch zur Laufzeit an Dienste delegiert, die im offenen Internet verfügbar sind. Die potenziell große Anzahl an möglichen Diensten und die Unkontrollierbarkeit des Internets machen eine solche Auswahl operativ schwierig. Beitrag B5 bezweckt, die dynamische Auswahl von wiederzuverwendenden (Web-)Services mit Mitteln der mathematischen Statistik operativ zu verbessern. Es werden im Beitrag die folgenden, fokussierten Forschungsfragen untersucht:

- Welche Besonderheiten zeichnet die dynamische, opportunistische Suche nach geeigneten Services in offenen Netzen aus?
- Unter welchen Annahmen und mit welchen Methoden kann die Suche unter solchen Umständen verbessert werden?
- Welche Vorteile sind zu erwarten?

Kapitel IV, Beitrag B6: Reducing domain level scenarios to test component-based software.

Theoretische Arbeiten zum Testen von Software konzentrieren sich traditionell auf formal-syntaktische Probleme. Vergleichsweise wenig untersucht wurden dagegen bislang Fragen der Softwareprüfung auf semantische und pragmatische Eignung in einem betrieblichen Ablauf („higher-order“ Tests auf Anwenderseite). Es handelt sich um eine schwierige operative Herausforderung, die aber hohe praktische Bedeutung besitzt. Beitrag B6 zielt auf die methodische Unterstützung der Auswahl pragmatisch geeigneter Komponenten und Dienste, wofür eine Methode zur frühzeitigen Prüfung von Spezifi-

kationen gegen Geschäftsprozessmodelle vorgeschlagen wird. Es werden im Beitrag die folgenden, fokussierten Forschungsfragen untersucht:

- Auf welche Weise kann geprüft werden, ob sich Softwarelösungen nicht nur formal-syntaktisch, sondern auch semantisch und besonders aus der pragmatischen Sicht zur Unterstützung eines durchgängigen betrieblichen Ablaufs eignen?
- Auf welcher Grundlage können solche Prüfungen möglichst früh im wiederverwendungsgetriebenen Entwicklungszyklus erfolgen?
- Wie können komplexe Geschäftsprozessmodelle so reduziert werden, dass sie sich als Grundlage für die Erstellung einfach auswertbarer „higher-order“ Prüfszenarien eignen?

I.3 Fachliche Einordnung und Aufbau

Für die Gliederung der Wirtschaftsinformatik in Teilgebiete existieren heute verschiedene Ansätze. Zu den wesentlichen Strukturierungsansätzen gehören neben dem wissenschaftstheoretischen vor allem der betriebswirtschaftlich orientierte und der inhaltlich ausgerichtete (Heinrich, Heinzl & Roithmayr 2007, S. 21f).

Für die fachliche Einordnung der Beiträge dieser Arbeit ist zunächst der inhaltliche Ansatz gut geeignet, da er sich stark an Gestaltungszielen orientiert. Die Beiträge dieser Arbeit können inhaltlich einerseits durch ihren Software Engineering Bezug in die (wiederverwendungsgetriebene) *Entwicklung betrieblicher Anwendungssysteme* eingeordnet werden, welche zu den zentralen Themen der Wirtschaftsinformatik gehört (Alpar et al. 2008, S. 287ff; Ferstl & Sinz 2008, S. 457ff; Heinrich, Heinzl & Roithmayr 2007, S. 23; Mertens et al. 2005, S. 153ff; Turowski 2003, S. 5f, S. 99ff). Die Beiträge können andererseits auch als Planungsthemen im Sinn eines gestaltend-zielgerichteten Leitungshandelns verstanden werden und dann inhaltlich, aufgrund ihres Information Engineering Bezugs, innerhalb der Wirtschaftsinformatik dem *Informationsmanagement* zugerechnet werden (Ferstl & Sinz 2008, S. 433f; Hansen & Neumann 2009, S. 240; Heinrich & Lehner 2005, S.7f).

In der betriebswirtschaftlich orientierten Sichtweise auf die Wirtschaftsinformatik werden, ebenso wie in der allgemeinen Betriebswirtschaftslehre, Aufgaben regelmäßig nach ihrer strategischen, taktischen bzw. operativen Reichweite gegliedert (Ferstl & Sinz 2008, S. 79, S. 438f; Hansen & Neumann 2009, S. 242; Heinrich & Lehner 2005, S. 22f; Heinrich, Heinzl & Roithmayr 2007, S. 216f). Diese betriebswirtschaftlich orientierte Gliederung gibt die übergeordnete Struktur für den Hauptteil der Arbeit vor. Abbildung I-2 illustriert den Aufbau der Arbeit.

Auf der strategischen Ebene der *langfristigen* Rahmenbedingungen werden umfassende, übergreifende Prinzipien und Konzepte mit einem Horizont von mindestens drei oder vier Jahren formuliert. Durch diese längerfristige, eher abstrakte und übergeordnete Betrachtungsweise – für die selbst der Begriff der Planung in mancher Hinsicht zu eng gefasst ist – sollen geeignete Grundsätze, Leitlinien, Normen und Standards entstehen, die zu erfolgskritischen Wettbewerbsvorteilen in der Grundkonfiguration der Entwicklungsvorhaben führen. Ein wichtiger Gestaltungspunkt sind dabei Vorgaben für nachgelagerte taktische Aufgaben, speziell hinsichtlich der Prioritäten bei mehreren Stakeholdern mit unterschiedlichen Interessenslagen.

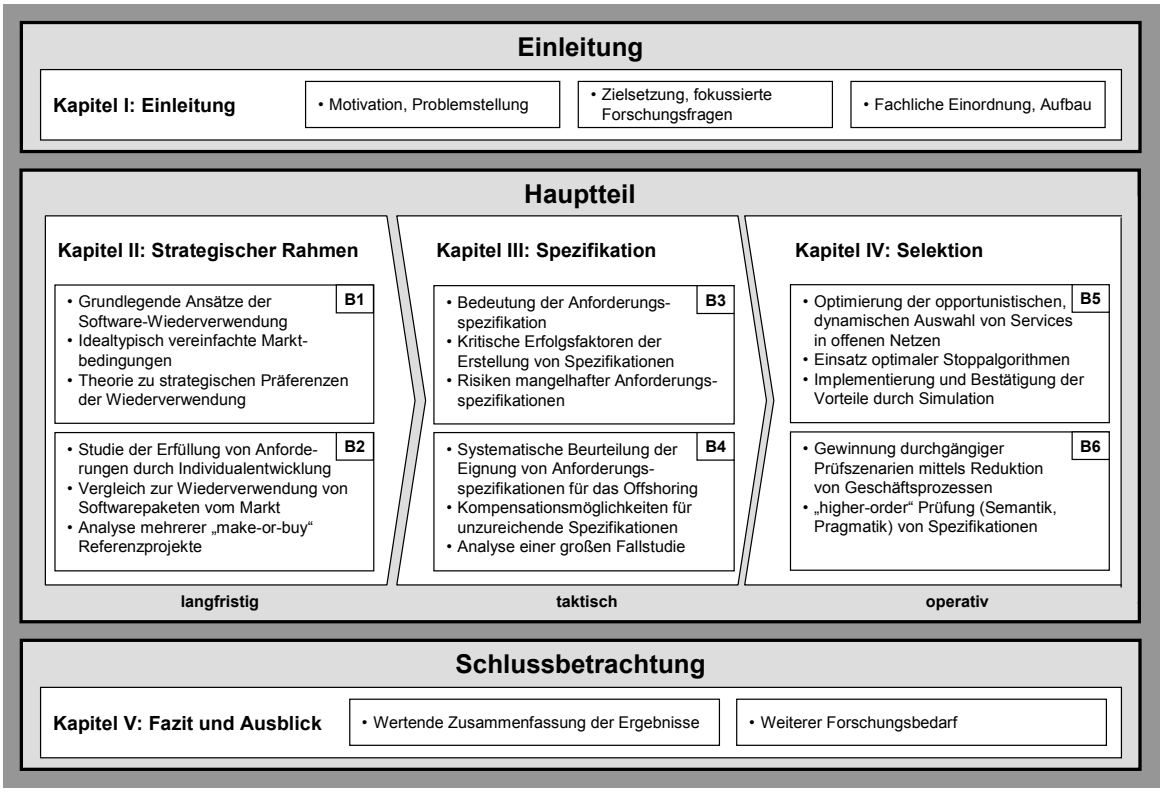

Abbildung I-2: Aufbau der Arbeit.

Die *taktischen* Aspekte (es werden in der Literatur auch andere Bezeichnungen verwendet, z.B. administrative Aspekte) betreffen die kurz- und mittelfristige Umsetzung der strategischen Vorgaben, die beispielsweise in einem Geschäfts-, Programm- oder Projektplan zusammengefasst werden können. Die Vorgaben, die im strategischen Rahmen gesetzt wurden, werden hier zu strukturierten Einzelzielen und -aufgaben heruntergebrochen und umgesetzt, um in der Folge die Strategie zu erfüllen. Wesentlicher Aspekt ist damit die Planung, Steuerung und Kontrolle der konkreten Durchführungsprozesse, darunter beispielsweise die Ressourcensteuerung, die Kommunikation und das Berichts- und Eskalationswesen. *Operative* Methoden und Techniken unterstützen schließlich konkret und detailliert die Durchführung der einzelnen Entwicklungsaufgaben im situa-

tionsabhängigen Tagesgeschäft. Die taktischen Vorgaben helfen hier bei der Beurteilung von Handlungsspielräumen und bilden den Korridor, innerhalb dessen man sich auf operativer Ebene bewegt.

Die Entwicklung betrieblicher Anwendungssysteme wird als komplexe Gesamtaufgabe nicht auf einen Streich gelöst, sondern in einzelne Teilschritte untergliedert. Ein Kernelement der strukturierten Entwicklung sind dabei die Vorgehensmodelle, d.h. bestimmte Phasenschemata des Entwicklungsablaufs, von denen im Laufe der Zeit mehrere entstanden sind. Einen Überblick über verschiedene, etablierte Vorgehensmodelle geben etwa Hansen und Neumann (2009, S. 364-383) oder Sametinger (1997, S. 151-158), speziell für die Komponentenorientierung etwa Turowski (2003, S. 112ff). Nicht zuletzt durch den Erfolg verteilter Open-Source Entwicklungsprojekte (wie das Linux Betriebsystem oder das Open Office Programmpaket) werden, spätestens seit Raymond (1998), die Vorgehensmodelle auch hinsichtlich ihrer impliziten, fundamentalen Grundannahmen hinterfragt und erweitert. Eine Erweiterung in Richtung der komponenten- und serviceorientierten Konstruktionsansätze wurde mit dem Multipfad-Vorgehensmodell (Ortner 1998, S. 332; Overhage 2006, S. 136) vorgeschlagen. Das Multipfad-Modell kann als Meta-Vorgehensmodell bezeichnet werden und zeigt sich als Erweiterung und Vereinheitlichung bestehender Vorgehensmodelle in Richtung der Software-Wiederverwendung auf strategischer Ebene. Der strategische Rahmen in Kapitel II dieser Arbeit lässt sich insofern im Multipfad-Vorgehensmodell einordnen.

Vorgehensmodelle geben als Phasenschemata Rahmenbedingungen für taktische Planungen und Entscheidungen vor. Vorgehensmodelle der Softwareentwicklung schreiten, allgemein und grob vereinfacht, von Lasten (Anforderungen) über Pflichten (Architektur, Design) zu Implementierung und (Abnahme-)Test voran, wobei die Arbeitsschritte in verschiedenster Weise iterativ, verteilt und von Maßnahmen der Qualitätssicherung aller Art begleitet sein können. Die Spezifikation von Anforderung ist dabei – in unterschiedlichen Ausprägungen – als Kernelement in allen wichtigen Vorgehensmodellen der Softwareentwicklung enthalten und gilt als besonders kritischer Einzelschritt im Software-Entwicklungszyklus (Alpar et al. 2008, S. 294; Sommerville 2001, S. 107). Anforderungsspezifikationen sind darüber hinaus ein fundamentales Grundprinzip jeder Art von Konstruktionsprozess, vgl. etwa Pahl et al. (2003, S. 9f). Durch die im komponenten- und serviceorientierten Ansatz propagierte Blackbox-Methode der Wiederverwendung mit ausschließlich expliziten Abhängigkeiten der verwendeten Einzelteile (Garlan, Allen & Ockerbloom 1995, S. 25) erhöht sich die Bedeutung von Anforderungsspezifikationen noch weiter, ebenso wie durch den zunehmenden Trend zu einer global arbeitsteiligen Entwicklung mit Offshore-Entwicklungsanteilen, die derzeit verstärkt in nachgelagerten Entwicklungsschritten wie der Implementierung eingesetzt

werden. Innerhalb der Vielzahl möglicher taktischer Fragestellungen der (wiederverwendungsgetriebenen) Entwicklung komponenten- und serviceorientierter betrieblicher Anwendungssysteme befassen sich die Beiträge zu taktischen Aspekten im Kapitel III dieser Arbeit daher mit der Spezifikation.

Nimmt man in den Leitszenarien der Komponentenorientierung nach Turowski (2003, S. 9-15) den Standpunkt der Nachfrageseite ein, die die Absicht hat, bereits existierende Komponenten und Services extern zu beziehen und im eigenen Kontext einzusetzen, so stellt sich die Identifikation und Auswahl geeigneter, wiederzuverwendenden Komponenten und Services als Kernproblem dar. Zwei unterschiedliche Ansätze zur Verbesserung der Komponenten- und Serviceauswahl in operativen Wiederverwendungssituationen werden in Kapitel IV dieser Arbeit vorgeschlagen.

Die vorliegende Arbeit ordnet sich also inhaltlich innerhalb der Wirtschaftsinformatik in das Leitbild der komponenten- und serviceorientierten Entwicklung betrieblicher Anwendungssysteme (Turowski 2003, S. 9-15) und in die Prozessbausteine seiner Konstruktionslehre (Overhage & Turowski 2008, S. 112f) ein. Das strategische Thema der Wiederverwendung, welche Hand in Hand mit dem komponenten- und serviceorientierten Konstruktionsleitbild einhergeht, bildet dabei den Ausgangspunkt und den Rahmen des Hauptteils der Arbeit.

Die übergeordnete Gliederung der Beiträge im Hauptteil folgt weiter der betriebswirtschaftlich orientierten Unterscheidung von strategischen, taktischen und operativen Sichtweiten. Der strategische Rahmen der Arbeit orientiert sich am Multipfad-Vorgehensmodell der komponenten- und serviceorientierten Entwicklung, die taktischen Beiträge behandeln die Spezifikation als besonders zentralen und kritischen Teil der Entwicklung, und die operativen Methoden sind, vom Standpunkt der Nachfrage, an der Selektion von Komponenten bzw. Services ausgerichtet.

Literatur (Einleitung)

Alpar, P.; Grob, H.; Weimann, P.; Winter, R. (2008), *Anwendungsorientierte Wirtschaftsinformatik: Strategische Planung, Entwicklung und Nutzung von Informations- und Kommunikationssystemen*, 5. Auflage, Vieweg, Wiesbaden.

Atkinson, C.; Bunse, C.; Groß, H.; Kühne, T. (2002), „Towards a General Component Model for Web-Based Applications“, *Annals of Software Engineering*, 13 (1): 35-69.

Biggerstaff, T.; Richter, C. (1987), „Reusability Framework, Assessment, and Directions“, *IEEE Software*, 4 (2): 41-49.

Boehm, B. (1986), *Wirtschaftliche Software-Produktion*, Forkel, Wiesbaden.

Brandon, D. (2006), *Project Management for Modern Information Systems*, IRM Press, Hershey, USA.

Brooks, F. (1987), „No Silver Bullet: Essence and Accidents of Software Engineering“, *IEEE Computer*, 20 (4): 10-19.

Brown, A. (2000), *Large-Scale, Component-Based Development*, Prentice Hall, Upper Saddle River, USA.

Chatzoglou, P. (1997), „Factors affecting completion of the requirements capture stage of projects with different characteristics“, *Information and Software Technology*, 39 (9): 627-640.

DeMarco, T. (1997), *Warum ist Software so teuer? Und andere Rätsel des Informationszeitalters*, Hanser, München.

Dahl, O.; Nygaard, K. (1966), „SIMULA – an ALGOL-based Simulation Language“, *Communications of the ACM*, 9 (9): 671-678.

Dijkstra, E. (1972), „The Humble Programmer“, *Communications of the ACM*, 15 (10): 859-866.

Ferstl, O.; Sinz, E. (2008), *Grundlagen der Wirtschaftsinformatik*, 6. Auflage, Oldenbourg, München.

Fröschle, H.; Reinheimer, S. (Hrsg.) (2007), „Serviceorientierte Architekturen“, *HMD – Praxis der Wirtschaftsinformatik*, 253.

Garlan, D.; Allen, R.; Ockerbloom, J. (1995), „Architectural Mismatch: Why Reuse Is So Hard“, *IEEE Software*, 12 (6): 17-26.

Glass, R. (1996), „Study Supports Existence of Software Crisis: Management Issues Appear to be Prime Cause“, *Journal of Systems and Software*, 32 (3): 183-184.

Glass, R. (2005), „’Silver Bullet’ Milestones in Software History“, *Communications of the ACM*, 48 (8): 15-18.

Glass, R. (2006), „Looking into the Challenges of Complex IT Projects“, *Communications of the ACM*, 49 (11): 15-17.

Hansen, H.; Neumann, G. (2009), *Wirtschaftsinformatik 1: Grundlagen und Anwendung*, 10. Auflage, Lucius & Lucius, Stuttgart.

Heinrich, L.; Heinzl, A.; Roithmayr, F. (2007), *Wirtschaftsinformatik: Einführung und Grundlegung*, 3. Auflage, Oldenbourg, München.

Heinrich, L.; Lehner, F. (2005), *Informationsmanagement*, 8. Auflage, Oldenbourg, München.

Henderson, J.; Venkatraman, N. (1993), „Strategic Alignment: Leveraging information technology for transforming organizations“, *IBM Systems Journal*, 32 (1): 4-16.

Martin, R.; Chang, C. (1994), „How to Solve the Management Crisis“, *IEEE Software*, 11 (6): 14-15.

Mertens, P.; Bodendorf, F.; König, W.; Picot, A.; Schumann, M.; Hess, T. (2005), *Grundzüge der Wirtschaftsinformatik*, 9. Auflage, Springer, Berlin.

Meyer, B. (1990), *Objektorientierte Softwareentwicklung*, Hanser, München.

Mili, H.; Mili, F.; Mili, A. (1995), „Reusing Software: Issues and research directions“, *IEEE Transactions on Software Engineering*, 21 (6): 528-562.

Moore, G. (1965), „Cramming More Components onto Integrated Circuits“, *Electronics Magazine*, 38 (8): 114-117.

Naur, P.; Randell, B. (Hrsg.) (1969), *Software Engineering: Report on a conference sponsored by the NATO Science Committee*, NATO Scientific Affairs Division, Brüssel, Belgien.

Neighbors, J. (1984), „The Draco Approach to Constructing Software from Reusable Components“, *IEEE Transactions on Software Engineering*, 10 (5):564-574.

Ortner, E. (1998), „Ein Multipfad-Vorgehensmodell für die Entwicklung von Informationssystemen – dargestellt am Beispiel von Workflow-Management Anwendungen“, *Wirtschaftsinformatik*, 40 (4): 329-337.

Overhage, S. (2006), „Vereinheitlichte Spezifikation von Komponenten: Grundlagen, UNSCOM Spezifikationsrahmen und Anwendung“, Dissertation, Universität Augsburg.

Overhage, S.; Turowski, K. (2008), „Ingenieurmäßige Entwicklung von Komponenten, Services und Anwendungssystemen: Zum Aufbau einer Konstruktionslehre für die

(Wirtschafts-)Informatik“, in Heinemann, E. (Hrsg.), *Anwendungsinformatik: Die Zukunft des Enterprise Engineering*, Nomos, Baden-Baden: 105-119.

Pahl, G.; Beitz, W.; Feldhusen, J.; Grote, K. (2003), *Konstruktionslehre: Grundlagen erfolgreicher Produktentwicklung: Methoden und Anwendung*, 5. Auflage, Springer, Berlin.

Parnas, D. (1985), „Software Aspects of Strategic Defense Systems“, *Communications of the ACM*, (28) 12: 1326-1335.

Potok, T.; Vouk, M.; Rindos, A. (1999), „Productivity Analysis of Object-Oriented Software Developed in a Commercial Environment“, *Software – Practice and Experience*, 29 (10): 833-847.

Poulin, J. (1997), *Measuring Software Reuse: Principles, Practices, and Economic Models*, Addison Wesley, Reading, USA.

Raymond, E. (1998), „The Cathedral and the Bazaar“, *First Monday*, 3 (3).

Rosenau, M. (1981), *Successful Project Management: A Step-by-Step Approach With Practical Examples*, Van Nostrand Reinhold, New York, USA.

Rost, J. (1997), „Wiederverwendbare Software“, *Wirtschaftsinformatik*, 39 (4): 357-365.

Rumbaugh, J.; Blaha, M.; Premerlami, W.; Eddy, F.; Lorensen, W. (1993), *Objektorientiertes Modellieren und Entwerfen*, Hanser, München.

Sackman, H.; Erikson, W.; Grant, E. (1968), „Exploratory Experimental Studies Comparing Online and Offline Programming Performance“, *Communications of the ACM*, (11) 1: 3-11.

Sametinger, J. (1997), *Software Engineering with Reusable Components*, Springer, Berlin.

Schulte, R. (1996), „’Service Oriented’ Architectures, Part 2“, Gartner Research ID Number SPA-401-069, Gartner, Stamford, USA.

Schulte, R.; Natis, Y. (1996), „’Service Oriented’ Architectures, Part 1“, Gartner Research ID Number SPA-401-068, Gartner, Stamford, USA.

Sommerville, I. (2001), *Software Engineering*, 6. Auflage, Pearson Studium, München.

Szyperski, C. (1998), *Component Software: Beyond Object-Oriented Programming*, Addison Wesley, Harlow, UK.

Teubner, A. (2006), „IT / Business Alignment“, *Wirtschaftsinformatik*, 48 (5): 368-371.

Turowski, K. (2003), *Fachkomponenten: Komponentenbasierte betriebliche Anwendungssysteme*, Shaker, Aachen.

Voas, J. (2001), „Faster, Better, and Cheaper“, *IEEE Software*, 18 (3): 96-97.

Wassermann, A.; Gutz, S. (1982), „The Future of Programming“, *Communications of the ACM*, 25 (3): 196-206.

Wikipedia (2009), Observable universe, Abruf am 17. Juli 2009, <http://en.wikipedia.org/wiki/Observable_universe>.

Wissenschaftliche Kommission Wirtschaftsinformatik (1994), „Profil der Wirtschaftsinformatik“, *Wirtschaftsinformatik*, 36 (1): 80-81.

Wissenschaftliche Kommission Wirtschaftsinformatik (2003), *Rahmenempfehlung für die Universitätsausbildung in Wirtschaftsinformatik*, Gesellschaft für Informatik, Bonn.

II Strategischer Rahmen

Beitrag B1:	**A theory of software reuse strategies in ideal type stable and turbulent market environments**
Autoren:	Oliver Skroch, Klaus Turowski Lehrstuhl für Wirtschaftsinformatik und Systems Engineering Universität Augsburg, Universitätsstr. 16, D-86159 Augsburg [oliver.skroch|klaus.turowski]@wiwi.uni-augsburg.de
Veröffentlichung:	Proceedings of the 15th Americas Conference on Information Systems, Association for Information Systems, 6.-9. August 2009, San Francisco, USA: #272 (2009).

Informationssysteme (IS) müssen zunehmend Ziele unterstützen, die jenseits von Projektgrenzen auf der Ebene langfristiger, globaler geschäftlicher Gesamtstrategien vorgegeben werden. In diesem Zusammenhang wird die Wiederverwendung von Software als eines der am meisten versprechenden Konzepte diskutiert. Wiederverwendung von Software gilt als einer der Schlüsselfaktoren für die erfolgreiche Entwicklung von IS und Anwendungssoftware. Als strategisches Konzept macht die Betrachtung hier nicht an Projektgrenzen halt, sondern ihr Horizont reicht bis hin zu globalen Märkten. Daher sind speziell auch Marktbedingungen für strategische Erwägungen in der Software-Wiederverwendung wichtig.

Neben den traditionellen, stabileren Marktbedingungen der „old economy“ findet man sich immer mehr auch in turbulenten „high tech“ Marktumfeldern wieder. Der folgende Beitrag untersucht in diesen beiden unterschiedlichen, idealtypischen Marktmilieus Geschäftsstrategien ihrer Marktteilnehmer, und die damit zusammenhängenden Möglichkeiten der generativen bzw. kompositorischen Software-Wiederverwendung. Unterstützend wird die Analyse von Erfahrungswerten aus drei großen Praxisprojekten präsentiert. Im Ergebnis wird eine entsprechende Wiederverwendungs-Theorie begründet, die den Beitrag mit der Formulierung zweier Hypothesen zu strategischen Management-Präferenzen bei der Wiederverwendung von Software schließt. Demnach wird die generative Wiederverwendung eher in traditionellen, stabilen Märkten bevorzugt, während die kompositorische Wiederverwendung eher in turbulenten Marktumfeldern vorteilhaft ist.

1 Introduction and objectives

The paramount relevance of IS (information systems) for today's businesses is being studied since many years, and strategies to leverage aligned business value (Henderson & Venkatraman 1993; Luftman, Papp & Brier 1999) from IS assets have become vital in many markets. For software businesses it was even mentioned that "value creation is the final arbiter of success […] In particular, there is a deeper understanding of the role of strategy in creating value" (Boehm & Sullivan 2000).

Software reuse is an important and promising strategic approach pursued for applications and IS to stipulate, contribute and align to business values. Reuse was recognized as a financial investment (Barnes & Bollinger 1991) and the relative costs of building IS from reuse, as opposed to building them for reuse, have been studied (Favaro 1991). As mentioned by Favaro (1996), value based principles for the management of reuse in the enterprise advocate the maximization of economic value as governing objective. The idea that "business decisions drive reuse" (Poulin 1997) was pointed out. Management processes of reuse were investigated, including the idea that reuse concepts evolve with increased investment and experience (Jacobson, Griss & Jonsson 1997). Strategic planning and metrics of reuse in large corporations were discussed in detail (Lim 1998).

While reuse offers various options and advantages today, one of the major remaining challenges is "a deeper understanding of when to use particular methods, based, for example, on […] business context" (Frakes & Kang 2005). This paper proposes a theory on this subject. It investigates strategic reuse options in software businesses and their potential value propositions in the context of two model type market environments with their core strategies.

Strategic management options in software processes can be explained in the multi-path process model in Figure II-B1-1, based on Ortner (1998) and Overhage (2006). The model proposes four strategy levels – individual solution, component solution, off-the-shelf solution and outsourcing. Two levels emphasize overall IS and applications: off-the-shelf solution implies the introduction of COTS (commercial off-the-shelf) applications, and outsourcing aims at service level agreements with 3rd party suppliers. Focus of this theory is on the two deeper multi-path levels which relate to organizations centered on software development aspects: individual solution and component solution. We recognize that these two levels imply different focal points for reuse, and we apply a classic distinction introduced by Biggerstaff and Richter (1987) associating *generative reuse* and *compositional reuse* with the two levels.

With highly specific features of an individual solution, focus is on the design and implementation of the new features *for* reuse, e.g. in other projects or on global markets. With common features of a component solution, focus is on IS design *from* reusable components, e.g. from catalogues or, again, global markets. These two reuse options differ widely in terms of management, applicability, and value contribution.

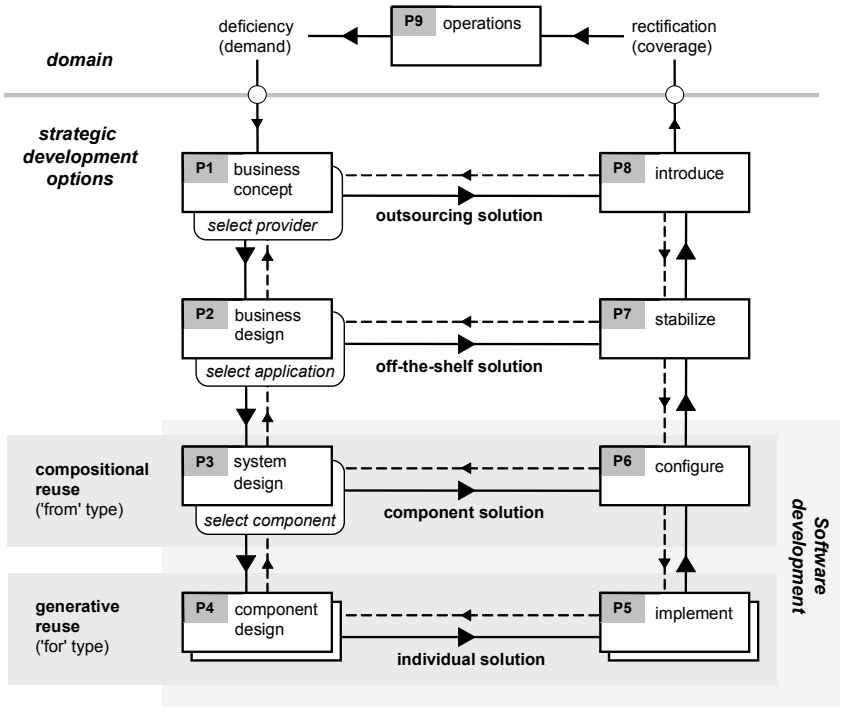

Figure II-B1-1: Strategic options in the multi-path process model.

On the business strategy level, management needs (among other things) strong market orientation for sustaining success. Market conditions purport business objectives; therefore we examine two different, ideal type business conditions and their respective market player strategies: *defenders* in traditional stable markets of diminishing returns and *prospectors* in turbulent "high-tech" markets of increasing returns. We show that underlying competition styles differ, drive distinct business goals and stipulate different entrepreneurial, managerial, engineering and administrative decisions (Miles & Snow 1978; Arthur 1996). Therefore, different value propositions are required, including specifically also software reuse approaches.

Combining these considerations, we derive a theory of reuse options supporting business strategies under the two market conditions. Similar theory building approaches have recently been taken e.g. to align IS architecture to business interaction patterns (Schlueter-Langdon 2003), to manage IT-enabled decision support in turbulent environments (Carlsson & El Sawy 2008), or to examine the contribution of network-

based market environments to the domain of information and communication technology (Rossignoli 2009).

Development of reasonable theory is a central activity in research and is traditionally based on a combination of previous theory and literature, common sense and experience, e.g. (Eisenhardt 1989; Yin 2003). Theory building, as research in its own right, precedes empirical hypothesis testing. Accordingly, this paper takes first steps first and constructs theory from the analysis of previously existing theories and literature, and from well-understood cases from practical experience.

2 Basic software reuse options

Many definitions exist for the concept of software reuse. We give two examples only – "the degree to which a software module or other work product can be used in more than one computer program or software system" (IEEE Standards Board 1990) and "the process of creating systems from existing software rather than building systems from scratch" (Krueger 1992) – and state that most reuse definitions implicitly suggest the intention to capitalize on pre-existing assets and knowledge already acquired in the past.

A widely accepted taxonomy proposed already by Biggerstaff and Richter (1987) distinguishes compositional reuse from generative reuse. This is so elementary that it can repeatedly be found under other names, e.g. "reuse techniques" (Prieto-Díaz 1993), or "software reuse technical tools" (Lim 1998). Table II-B1-1 is based on Biggerstaff and Richter (1987), provides an overview of compositional and generative reuse, and mentions some of their characteristics.

Table II-B1-1: Fundamental reuse strategies.

Reuse Strategy	**Compositional**	**Generative**
Reused Entity	building blocks	solution patterns
Nature of Entity	atomic and immutable, passive	diffuse and malleable, active
Emphasis	repositories (markets), composition principles	generators, processes
Examples	class library, Web service, component	4th generation language, code generator, design pattern

The compositional idea aims at directly reusing binary artifacts from repositories or markets to put together large applications. The generative method "is based on the reuse of a generation process" (Sametinger 1997) which is a higher level of abstraction and works indirectly by generating, partly automated, software from abstract patterns or models.

Business value creation from software reuse depends upon its field of adoption and the higher ranking business objectives derived (among others) from market environments. Reuse can for example reduce the time required to create or modify enterprise applications, providing increased adaptation capabilities and shortened delivery timescales for the enterprise (Lim 1998). Or, combining individually programmed applications with COTS systems can lead to optimized application portfolios, delivering higher quality with reduced lifecycle costs to the enterprise (Orfali, Harkey & Edwards 1996). Therefore strategic management decisions on reuse can make a difference and require consideration.

2.1 Compositional reuse – building blocks

In compositional reuse, prefabricated artifacts are reused to assemble large applications. The vision is, eventually, to establish a software components industry. This concept can be traced back to the 1960s (McIlroy 1969). Compositional reuse can be understood from the idea of *modularity* in systems theory (Simon 1981) and software engineering (Parnas 1972) among others. By assembling modular compounds from smaller sub-compounds that can be designed independently yet function together as a whole, traditional industries (e.g. electronics, automotive) have experienced previously unknown levels of innovation and growth, e.g. Baldwin and Clark (1999). Software businesses set out to follow such success stories through reusable binary software components (Szyperski, Gruntz & Murer 2002). But building IS from compositional reuse remains difficult. While component trading has arrived on (electronic) markets – e.g. for Web services, which can be seen as flavors of compositional reuse (Atkinson et al. 2002) – it has not become mainstream practice yet. Among the reason mentioned is the insufficient maturity of the software engineering discipline with its particular absence of commonly accepted standards (Hahn & Turowski 2005).

An important managerial issue in compositional reuse is the black box type of access the reusing party has to the component. Black box reuse employs existing assets in plug and play style without modification, only on the basis of a specified behavior at the interfaces, e.g. Brown (2000). Black box style reuse inevitably is restrained by the design that was chosen for the implementation of the selected components. This design cannot be changed and if a certain component behaves differently from specific design constraints in the overall IS then its reuse adds no value, because it might be inefficient or even impossible to fit in this particular component.

2.2 Generative reuse – solution patterns

Leading generative reuse approaches include scavenging, generative programming, model-driven architecture and product-line engineering. In scavenging (Krueger 1992), fragments of source code are copied. Generative programming (Czarnecki & Eisenecker 2000) automatically creates software through configuration within a predefined solution space. Model-driven architecture (Soley 2000) captures core software assets as platform-independent models and automatically derives the implementations. Product-line engineering (Weiss & Lai 1999) groups IS development around families of products and manages commonalities and variabilities.

Generative reuse works on a higher abstraction level as compared to compositional reuse, and in particular it is independent from implementation. It can be explained from the fundamental idea of *pattern abstractions*. Alexander et al. (1977) first presented the pattern approach and defined "pattern languages" as sets of abstract, well-proven solutions for reoccurring problems which emerge as the related domain develops. The pattern idea was also embraced in software businesses for reusing suitable solutions and concepts that have been worked out and used successfully before. Patterns became widely accepted with object-oriented design patterns (Gamma et al. 1995) latest. Pattern abstractions have been identified, described and used for many more aspects since then.

Significant managerial issues with generative reuse are its domain specific quality and the operational difficulties with generators that synthesize software for a target IS. Patterns are specific for a business, industry, market or domain. They alone lack the implementation paragraph required for reuse. The generative reuse approach is therefore based on the reuse of both a (formalized) pattern abstraction and a generative process (automatically) creating the reused entity from this abstraction.

3 Two ideal type market environments and their business strategy

Two different model types of market environments can be distinguished as shown in Table II-B1-2: traditional stable markets of diminishing returns and turbulent markets of increasing returns (Arthur 1996). The traditional view on markets as coordination mechanisms describes development on substitutable resources. Players expand in perfect competition until eventually a stable equilibrium is established that generates small predictable margins with prices at the average production cost. But observations in modern "high-tech" businesses reveal a different scenario with markets that develop on knowledge with the first winning mover out of a turbulent uncertainty being able to lock the market into an instable positive feedback loop thus generating large margins.

Following Miles and Snow (1978), typical players in these two environments can be characterized as defenders and prospectors.

Table II-B1-2: Two ideal type market environments.

Market Environment	**Traditional**	**Turbulent**
Dynamics	quite stable	highly dynamic
Returns	diminishing	increasing
Processing	resources	information
Business Models	mature, well established	changing, unprecedented
Competitive Drivers	risk avoidance, cost control, quality assurance	innovation, time to market, flexibility
Typical Player	defender with internal focus	prospector with external focus
Strategy	constant internal improvement at low risk	rapid adaptation to external changes

While real market conditions will rarely reflect one of these two ideal sides in full clarity, we start with a "reductionist" view and acknowledge that both market environments represent two aspects of reality that fundamentally differ in their underlying economics, their character of competition, their entrepreneurial, managerial, engineering and administrative problems, and their related business strategies. They present different challenges for software management, and consequently for the issue of reuse strategies, too.

3.1 Traditional environments – defenders

Traditional markets reflect the 19th century Marshall view of economic machinery that processes substitutable resources. Characteristics of such markets are established and steady market shares in supply, together with noticeable preferences in demand. Most suppliers share a common level of highly developed technologies, products and services. Collaboration is well established, markets "act" as coordination mechanisms and prices reach equilibrium at the average cost of production, which is stable since it generates small predictable margins. Often there are accepted quality standards, sometimes even legally enforced, and de facto pricing categories for products and services exist.

Agents that get ahead eventually face limitations from rising costs (e.g. resource shortage) or falling profits (e.g. increased competition). This can be explained from the high maturity levels that such businesses have passed through. Challenges from unforeseen innovations are unlikely and no strategic management issue, since no player is actually able to corner the market. Stable market environments are associated with the

"old economy" of diminishing returns. A typical player in this environment is the defender organization that devotes primary attention to improving the efficiency of its existing operations (Miles & Snow 1978).

3.1.1 Defensive internal improvement strategies

In businesses characterized by defensive internal improvement strategies, competitors can hardly dislodge established players from their positions, and only major market shifts would create actual opportunities or threats. Management perspectives therefore remain centered on efficient and well-proven technologies. Defenders are managed towards maintaining stability and efficiency, while they are not prepared to face changes. Consequently, larger investments are reasonable only for technological problems that remain common and unsurprising for a longer period (Miles & Snow 1978).

Business strategy is towards avoiding risks and permanently reducing costs at high and stable quality levels. Management steadily improves the repeating processes and sustains slow but continuous long-term improvement in small steps. This can be achieved by constant internal optimization and quality assurance, by planning and hierarchical control (Arthur 1996).

Under stable market conditions, IS advance continuously, too. A process of successive maturation has finally resulted in grown and mature legacy applications and a well practiced business process routine. Both are well aligned and efficiently support a stable business. IS and software applications are regarded as a *commodity*, and the associated IT processes have become routine tasks, too. But there is small and steady market pressure to always slightly improve competitiveness. Further enhancements on top of the already achieved levels are therefore very sophisticated features above the established standards.

Management generally prefers reuse to building software from scratch in such environments. Generative reuse in particular provides more control and promises lower life cycle costs through automation and generators. Patterns for reuse can be discovered (only) in stable and repeating processes. The more a certain domain evolves, the more patterns can be discovered. Documented patterns represent domain specific, highly specialized improvement potential to still deliver lower costs while not decreasing quality. In generative approaches, patterns and models are explicitly documented and therefore can be tailored during the generative process, too. The generative process also implicitly improves measurement and control of the generated software quality.

The time and complexity of realizing generative reuse in particular includes the laborious and difficult formalization of the underlying models and the building of software generators. This can be acceptable in this environment, as long as higher optimization levels are reached while competition remains stable, and a positive long-time return is assured.

3.1.2 Defenders' dilemma

Managing such defensive strategy faces an important dilemma: with increasing sophistication of businesses, IS, and application software, further improvement is attended by higher efforts while at the same time marginal gains decline. Another incremental improvement might always be found, but the increments become smaller, the related efforts grow, and beyond a certain point negative returns might result – even with formal models and automation.

Compositional reuse seems no option here since it would assume, as a prerequisite, the availability of suitable components that provide factual advantages. While it is very likely that high quality prefabricated components exist in mature markets, it is unlikely that these will provide any competitive edge. Their functionalities and qualities will be close to established de facto standards and therefore they will neither threat (respectively help) an established player, nor will they provide true, unique advantages to newcomers.

In brief, the analysis of traditional business environments with defensive market players suggests the theory that management follows low risk optimization strategies and considers especially the generative reuse option.

3.2 Turbulent environments – prospectors

Turbulent markets are described from the outstanding performance of the "high-tech" sector in the late 1990s (Gordon 2000). Such environments are characterized as ICT (information and communication technology) driven (Klodt 2001). These markets are only loosely regulated, highly complex and unstable, and face coordination challenges. New goods based on intangible resources are created rapidly. They alter quickly and unpredictably, and change during IS development. Market entry barriers are high: new technologies require significant up-front engagement with the risk of an uncertain outcome.

These markets always change and players and collaborations rapidly emerge and vanish. But successful players can grow at high rates and realize excessive margins, since the markets show "winner-takes-all" properties: the first successful mover is able to lock a market for the own product or service. Turbulent markets spread, because of the increasing importance of intangible resources (information, knowledge, etc.), which in parallel becomes widely and cheaply available (through software, on the Internet, etc.) (Boehm 2005). Most of their dynamics can be explained in traditional terms and no new strategic textbooks are required (Porter 2001).

We associate turbulent markets with increasing returns environments (Katz & Shapiro 1985; Elsner 2004). A typical player characteristic for this environment is the prospector organization that embraces change and shows a strong concern for product and market innovation (Miles & Snow 1978).

3.2.1 Prospective rapid adaptation strategies

In businesses characterized by prospective external adaptation strategies, players meet changing conditions with own innovations, but run the risk of overextending their resources. Management focus is on technological flexibility to enable rapid responses, while maximum efficiency cannot develop. Prospectors are managed to maintain flexibility but may not optimally utilize their resources (Miles & Snow 1978).

Business strategy is towards rapid external adaptation, as unpredictable situations demand reactivity and quick response from management. Prospectors are managed as mission oriented organizations which compete for the next winning business model or technology, and the winner will take most. Hence it is imperative to enter the market first if possible, with a new business model and IS that work *well enough* to support the new business and become widely accepted (Arthur 1996).

The associated IS are completely new or even not existing yet. Moreover, in turbulent "high-tech" environments the IS are often expected to stipulate new business models or support new business functions for the first time in the market. Such ideas permanently appear and vanish and management has little indication of their longer term significance.

To still support the overall business strategy, the organization needs to be primarily managed towards high flexibility. Flexibility in building IS originates in low development efforts. Compositional reuse reduces efforts and provides flexibility by assembling IS from ready-to-use components that are loosely "plugged" onto frameworks. Management can minimize overall efforts through skillful demarcation of

the domain and through covering demanded features with existing components where possible. Related IS might then start as component tapestry, put together ad hoc to satisfy the current business well enough. In unstable domain parts, the IS adopts by exchanging components. In parts that become stable the IS evolve into persistent domain specific frameworks.

3.2.2 Prospectors' dilemma

Management encounters the main dilemma for prospectors: the IS life cycle is unknown beforehand. Many ideas for new products and services are brought forward but their commercial prospects can hardly be predicted. Organizations need to be prepared to start over from zero again and again, chasing new ideas as they appear. At the same time, if a business, product or service survives, supporting IS that were quickly plugged together might have to be sustained, possibly over a longer period of time, until they are eventually either replaced or become properly institutionalized.

Generative reuse seems no option here since there is little maturity in these continuously changing environments and few if any patterns can be identified. A situation will rarely reappear, and the successful reuse of patterns is unlikely. Also the amount of time and effort required to prepare and maintain formal models and generators opposes the business strategy.

In brief, the analysis of turbulent business environments with prospective market players suggests the theory that management follows fast external adaptation strategies and favors especially the compositional reuse option.

4 Supporting experience: projects from practice

We support our assumptions through three selected projects which we were involved in between 2000 and 2005 (the reports had to be made anonymous, which does not affect their arguments). The experience provides valid substantiation for our suggestions. This is not meant as empirical evidence to test our theory, which is a subsequent step after having derived reasonable hypotheses in the first place. But it is a core element in theory building, as described e.g. by Eisenhardt (1989).

4.1 Stable environment – fraud detection

A multinational corporation was working in a holding-type structure with one head quarter and several operative units on two continents. The head quarter received

management reports from all units in a central reporting database. This procedure was highly standardized, most steps were automated. Certain reappearing irregularities in the figures were found manually and management suspected a new type of fraud. A self-learning fraud detection tool was used as part of the IS since long on all reporting figures as part of the daily processes. This tool was made individually for the firm, but it failed to identify the new fraud type.

No functionality recognizing this specific irregularity was available as prefabricated solution. No market demand for such highly specific feature existed, hence no supply either. The feature was then implemented individually to enhance the existing IS, which could deal with a whole new fraud class afterwards. The implementation also used existing software automation tools for generating code skeletons.

In this stable environment, generative reuse worked well on a bespoke functionality, its pattern abstraction, and the partly automated generation of software from that abstraction. Compositional reuse would have failed because no component existed for the highly specific requirement.

4.2 Turbulent environment – software simulator

One of the leading diversified corporations world-wide acquired a base technology patent and created a business case for it. The new technology had to be simulated by software first, to prove that the technology works in principle and to clear the budget for a physical prototype.

A number of simulation software product suites were available on the market. The actual simulation requirements were not fully understood and it was expected that they would change during development. Coding from scratch was recognized as inevitable for most parts of the simulation core. But for the general parts of the simulator, e.g. user interfaces, random number generation, scenario logging and replay, etc., standard components could be found and put together. Meanwhile, all specific new functionality was developed from scratch.

In this “high-tech” business situation, compositional reuse worked well to quickly deliver unspecific functions, while coding from scratch was minimized to the new features. Generative reuse would have failed because it is impossible to identify patterns and implement a generative process for a solution that is unknown at development time.

4.3 Hybrid environment – portal architecture

A large multinational publisher ran its print products business very successfully since decades. Business was managed decentrally, and each subsidiary had own IS landscapes consisting of COTS and a number of individually created tools. The situation was stable and the IS worked nicely in the absence of larger changes.

Following the shift in publishing markets towards digital content, new IS became necessary. Prefabricated portal components available on the market were planned to encapsulate the back-office legacy. Small individually designed back-office amendments, mainly in the form of adaptors, were to enable inter-operability. The implementation approach was to realize the changes in one reference environment, and to reuse this as blueprint in the other subsidiaries.

Market changes shaped a complicated hybrid situation with the traditional business still running while an uncertain new business had to be realized. The target IS was based on compositional reuse to provide new functionality for the new business lines, and generative reuse to encapsulate legacy applications supporting the traditional businesses.

5 Concluding hypotheses, limitations and further steps

We investigated software reuse strategies and saw that there are two fundamental options for organizations building software applications for large IS: compositional reuse based on assembling prefabricated components, and generative reuse based on models, patterns and generators. We also investigated two ideal type business conditions, stable and turbulent, each with typical players, defenders and prospectors, with their typical business strategies.

Combining the concepts, we argued that generative reuse is more likely to yield value for defenders in traditional stable environments where marginal gains are low and improvements difficult to achieve. In contrast, we argued that compositional reuse is more likely to be useful for prospectors in turbulent businesses because it is faster. We strengthened our argument with experience from three selected projects, not as a test of theory but as one step in building reasonable theory in the first place. Essentially, we believe that successful software reuse management delivers low risk improvements for defensive business strategies rather through generative reuse concepts, and short time-to-market for prospective business strategies rather through compositional reuse approaches.

We can state this as two hypotheses now:

- Generative reuse is an adequate strategic software reuse management option in traditional stable markets characterized by defender organizations.
- Compositional reuse is an adequate strategic software reuse management option in turbulent dynamic markets characterized by prospector organizations.

Table II-B1-3 briefly sums up the synthesis of the hypotheses. Managerial implications include the need to assess the type of market environment for the considered business, product, or service, with their supporting IS. With the type of market environment as one influence factor, management could then derive an unspecific preference for a software reuse strategy option.

Table II-B1-3: Reuse options and market players.

Market Player	**Defender**	**Prospector**
Market environment	traditional	turbulent
Strategic focus	constant internal improvement at low risk	rapid adaptation to external opportunities and threats
IS and software applications	grown legacy systems, highly evolved	ad hoc / none, frameworks
Reuse objectives	well-understood and proven patterns, improvement in small increments	low development efforts, being fast and "good enough"
Dilemma	declining cost-benefit ratio	unknown system life cycles
Preferred reuse strategy	generative	compositional

Our theory is limited by the fact that real situations show highly complex, multifaceted markets, businesses, IS, and software applications, with a growing importance of increasing returns effects (Boehm 2005; Samavi, Yu & Topaloglou 2009). The model type market environments – which we deliberately had to assume to find a "reductionist" starting point for theory development – are only weak approximations of real market conditions. Moreover, there are other important factors influencing strategic software reuse decisions apart from market environments, which is also out of scope of the present theory. Further limitations come from the fact that real life management alternatives are seldom fully confined model type options, and e.g. Llorens et al. (2006) reason about advantages of a holistic "incremental software reuse" theory (without framing it concretely). Furthermore, as we saw in the hybrid environment case, traditional lines of business can (and often do) exist together with turbulent businesses in one company. Management could e.g. separate out the domains, but our present strategic hypotheses do not focus on related operational issues. The hypotheses are no broad software reuse strategy guide, but a step towards recognizing adequate strategic

reuse preferences that suggest themselves in opposing market environments. Finally, our theory is only constructed by now and not empirically confirmed yet.

Main contribution of this work is that we could constitute – by reasonably reducing considerations – two concrete hypotheses of software reuse management strategies in different market environments. This qualitative theory building approach can now be expanded by a quantitative approach to challenge the theory and to establish reconfirmed ex-ante management strategy support as also ex-post assessment frameworks that can help to approximate the diligence of software management strategies.

References (B1)

Alexander, C.; Ishikawa, S.; Silverstein, M.; Jacobson, M.; Fiksdahl-King, I.; Angel, S. (1977), *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, New York, USA.

Arthur, B. (1996), "Increasing Returns and the Two Worlds of Business", *Harvard Business Review*, 74 (4): 100-109.

Atkinson, C.; Bunse, C.; Groß H.; Kühne, T. (2002), "Towards a General Component Model for Web-Based Applications", *Annals of Software Engineering*, 13 (1): 35-69.

Baldwin, C.; Clark, K. (1999), *Design Rules Volume 1: The Power of Modularity*, MIT Press, Cambridge, USA.

Barnes, B.; Bollinger, T. (1991), "Making Reuse Cost-Effective", *IEEE Software*, 8 (1): 13-24.

Biggerstaff, T.; Richter, C. (1987), "Reusability Framework, Assessment, and Directions", *IEEE Software*, 4 (2): 41-49.

Boehm, B. (2005), "The Future of Software Processes", *Unifying the Software Process Spectrum: Proceedings of the International Software Process Workshop: Revised Selected Papers*, Lecture Notes in Computer Science 3840, Springer, 25-27 May 2005, Beijing, China: 10-24.

Boehm, B.; Sullivan, K. (2000), "Software Economics: A Roadmap", *Proceedings of the 22nd International Conference on Software Engineering: Future of Software Engineering Track*, ACM, 4-11 June 2000, Limerick, Ireland: 319-343.

Brown, A. (2000), *Large-Scale, Component-Based Development*, Prentice Hall, Upper Saddle River, USA.

Carlsson, S.; El Sawy, O. (2008), "Managing the five tensions of IT-enabled decision support in turbulent and high-velocity environments", *Information Systems and e-Business Management*, 6 (3): 225-237.

Czarnecki, K.; Eisenecker, U. (2000), *Generative Programming: Methods, Tools, and Applications*, Addison Wesley, Boston, USA.

Eisenhardt, K. (1989), "Building Theories from Case Study Research", *Academy of Management Review*, 14 (4): 532-550.

Elsner, W. (2004), "The 'New' Economy: Complexity, Coordination and a Hybrid Governance Approach", *International Journal of Social Economics*, 31 (11/12): 1029-1049.

Favaro, J. (1991), "What Price Reusability? A Case Study", *ACM SIG Ada – Ada Letters*, 11 (3): 115-124.

Favaro, J. (1996), "Value Based Principles for Management of Reuse in the Enterprise", *Proceedings of the 4th International Conference on Software Reuse*, IEEE Computer Society, 23-26 April 1996, Orlando, USA: 221-222.

Frakes, W.; Kang, K. (2005), "Software Reuse Research: Status and Future", *IEEE Transactions on Software Engineering*, 31 (7): 529-536.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Boston, USA.

Gordon, R. (2000), "Does the 'New Economy' Measure up to the Great Inventions of the Past?", *Journal of Economic Perspectives*, 14 (4): 49-74.

Hahn, H.; Turowski, K. (2005), "Modularity of the Software Industry: A Model for the Use of Standards and Alternative Coordination Mechanisms", *International Journal of IT Standards and Standardization Research*, 3 (2): 68-80.

Henderson, J.; Venkatraman, N. (1993), "Strategic Alignment: Leveraging information technology for transforming organizations", *IBM Systems Journal*, 32 (1): 4-16.

IEEE Standards Board (1990), *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, New York, USA.

Jacobson, I.; Griss, M.; Jonsson, P. (1997), *Software Reuse: Architecture, Process and Organization for Business Success*, ACM Press, New York, USA.

Katz, M.; Shapiro, C. (1985), "Network externalities, competition, and compatibility", *American Economic Review*, 75 (3): 424-440.

Klodt, H. (2001), "The Essence of the New Economy", Kiel Discussion Paper 375, Kiel Institute for World Economics, Kiel.

Krueger, C. (1992), "Software Reuse", *ACM Computing Surveys*, 24 (2): 131-183.

Lim, W. (1998), *Managing Software Reuse*, Prentice Hall, Upper Saddle River, USA.

Luftman, J.; Papp, R.; Brier, T. (1999), "Enablers and Inhibitors of Business-IT Alignment", *Communications of the Association for Information Systems*, 1: 11.

Llorens, J.; Fuentes, J.; Prieto-Díaz, R.; Astudillo, H. (2006), "Incremental Software Reuse", *Reuse of Off-the-Shelf Components: Proceedings of the 9th International Conference on Software Reuse*, Lecture Notes in Computer Science 4039, Springer, 11-14 June 2006, Turin, Italy: 386-389.

McIlroy, M. (1969), "Mass Produced Software Components", *Software Engineering: Report on a conference sponsored by the NATO Science Committee*, NATO Scientific Affairs Division, 7-11 October 1968, Garmisch: 138-155.

Miles, R.; Snow, C. (1978), *Organizational Strategy, Structure, and Process*, McGraw Hill, New York, USA.

Orfali, R.; Harkey, D.; Edwards, J. (1996), *The Essential Distributed Objects Survival Guide*, Wiley, New York, USA.

Ortner, E. (1998), "Ein Multipfad-Vorgehensmodell für die Entwicklung von Informationssystemen – dargestellt am Beispiel von Workflow-Management Anwendungen", *Wirtschaftsinformatik*, 40 (4): 329-337.

Overhage, S. (2006), "Vereinheitlichte Spezifikation von Komponenten: Grundlagen, UNSCOM Spezifikationsrahmen und Anwendung", Dissertation, Universität Augsburg.

Parnas, D. (1972), "On the Criteria To Be Used in Decomposing Systems into Modules", *Communications of the ACM*, 15 (12): 1053-1058.

Porter, M. (2001), "Strategy and The Internet", *Harvard Business Review*, 79 (2): 63-78.

Poulin, J. (1997), *Measuring Software Reuse: Principles, Practices, and Economic Models*, Addison Wesley, Reading, USA.

Prieto-Díaz, R. (1993), "Status Report: Software Reusability", *IEEE Software*, 10 (3): 61-66.

Rossignoli, C. (2009), "The contribution of transaction cost theory and other network-oriented techniques to digital markets", *Information Systems and e-Business Management*, 7 (1): 57-79.

Samavi, R.; Yu, E.; Topaloglou, T. (2009), "Strategic reasoning about business models: a conceptual modeling approach", *Information Systems and e-Business Management*, 7 (2): 171-198.

Sametinger, J. (1997), *Software Engineering with Reusable Components*, Springer, Berlin.

Schlueter-Langdon, C. (2003), "Information systems architecture styles and business interaction patterns: Toward theoretic correspondence", *Information Systems and e-Business Management*, 1 (3): 283-304.

Simon, H. (1981), *The Sciences Of The Artificial*, MIT Press, Cambridge, USA.

Soley, R. (2000), "Model Driven Architecture: Object Management Group White Paper", OMG, Needham, USA.

Szyperski, C.; Gruntz, D.; Murer, S. (2002), *Component Software: Beyond Object-Oriented Programming*, 2nd edition, Addison Wesley, London, UK.

Weiss, D.; Lai, C. (1999), *Software Product-Line Engineering: A Family-Based Software Development Process*, Addison Wesley, Reading, USA.

Yin, R. (2003), *Case Study Research: Design and Methods*, 3rd edition, Sage, Thousand Oaks, USA.

Beitrag B2:	**Integration assessment of an individually developed application vs. software packages from the market – an experience report**
Autor:	Oliver Skroch Lehrstuhl für Wirtschaftsinformatik und Systems Engineering Universität Augsburg, Universitätsstr. 16, D-86159 Augsburg oliver.skroch@wiwi.uni-augsburg.de
Veröffentlichung:	Integration, Informationslogistik und Architektur: Tagungsband DW 2006, Lecture Notes in Informatics P-90, Gesellschaft für Informatik, 21.-22. September 2006, Friedrichshafen: 329-340 (2006).

Der folgende Beitrag präsentiert die wissenschaftliche Analyse einer Auswahl von Ergebnissen aus einem strategischen Beratungsprojekt, das für einen internationalen Konzern aus der Kommunikationsbranche durchgeführt wurde. Die Projektergebnisse werden hinsichtlich der „make-or-buy“ Entwicklungsstrategie bei sehr großen betrieblichen Anwendungssystemen untersucht. Die Untersuchungen beinhalten die Analyse und Beurteilung von Aktivitäten und Ergebnissen des Projekts zur Individualentwicklung, sowie von alternativen Planungsmöglichkeiten, die anstelle der Individualentwicklung auf der Wiederverwendung von am Markt erhältlichen Softwareprodukten und -komponenten basieren. Es erfolgt ein Vergleich zwischen den strategischen Alternativen der Individualentwicklung und der Beschaffung von Softwareprodukten und -paketen am Markt. Im Ergebnis der Analysen zeigen sich aus dem Referenzprojekt zwar Vorteile für die Beschaffung, das Gesamtbild über mehrere Projekte ist unter dem Strich aber nicht mehr eindeutig.

1 Introduction and setting

National monopolist providers still dominate fixed wire telecommunications landscapes in many parts of this world. Nevertheless the overall picture started changing since some years, with regulatory bodies promoting privatization and competition, and further players entering the markets put pressure on the incumbents. This has been examined in a survey on the implications of EU (European Union) legislation on telecom providers in the new EU member states (Ewers et al. 2004). Hence, especially communications suppliers that are still monopolists in their markets today have started preparing for a competitive future, e.g. Pyshkin (2003).

One of the core competitive assets in the supply of communications services are information systems. Here, providers typically make the distinction between OSS (operation support systems) and BSS (business support systems). In most definitions, OSS includes all systems that are directly related to the telecom networks themselves and their technical processes, such as network management or IN (intelligent network) platforms. BSS, on the other side, include the downstream applications less directly related to network technology and mainly driven by business needs. Typical examples for BSS functionalities include billing, CRM (customer relationship management), or order processing. The actual mapping of a system or component can be ambiguous and is also subject to change with the NGN (next generation networks) trend or the spread of voice over IP (internet protocol), e.g. Skroch and Turowski (2006). Further topical insight provide for example the ITU-T (International Telecommunication Union – Telecommunication Standardization Sector) Recommendations M Series.

Compared to OSS, the BSS area is much less standardized – even very few accepted industry standards exist. BSS functions are significantly more complex and intertwined, and are expected to be very well aligned to fully integrate the respective providers' businesses. Furthermore, many parts of the BSS in telecoms need to be high-availability and high-performance systems. Finally, many BSS functions are subject to rapid, unexpected and market driven change, in particular as to the fast implementation of new marketing ideas. BSS suppliers providing respective carrier-grade software systems form a highly fragmented market, e.g. Frost and Sullivan (2003), and many telecom providers, especially the very large ones, still have major parts of their BSS individually developed. Traditional individual software development, however, faces more and more constraints, and related considerations continue in theory as well as in practice (Taubner 2005).

To master this software challenge, next to others the concept of compositional reuse (Biggerstaff & Richter 1987) integrating prefabricated business components traded on markets (Turowski 2003; Szyperski 1998) is pursued in theory and practice since long. It can be seen as complementary alternative to traditional approaches such as individual development today. As early as in 1969 software components were proposed, with catalogues of software parts that can be retrieved and composed to large applications, similar as electronic parts (McIlroy 1969). Later it has even been stated that reuse is the "only realistic approach" to meet the future software needs (Mili, Mili & Mili 1995, p. 528).

Important differences in development and integration approaches, and specifically also peculiarities of procured solutions, component solutions and individual solutions, can be explained in the flexible multi-path process model in Figure II-B2-1. Four development levels are presented in the model. One or more of the levels can be chosen to satisfy an identified requirement through information systems support.

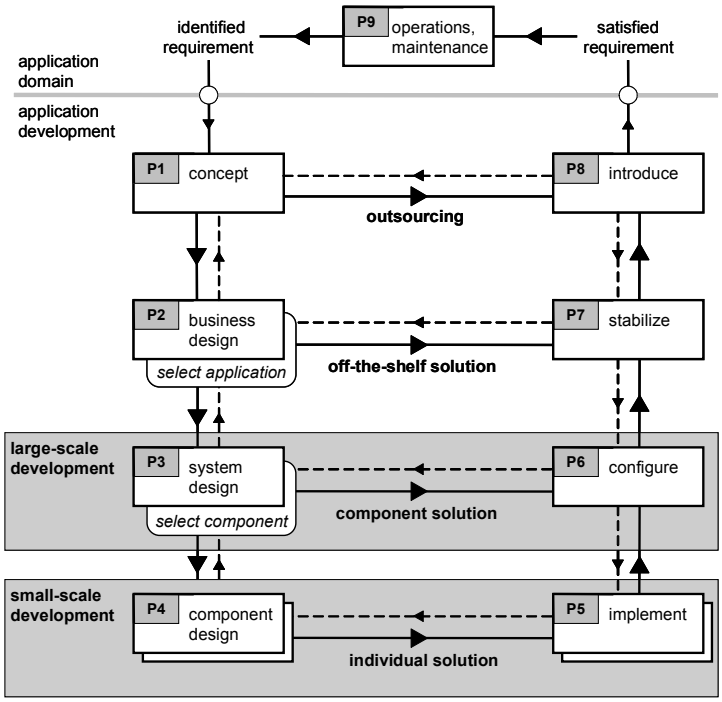

Figure II-B2-1: Multi-path process model.
Becker and Overhage (2003, p. 19), based on Ortner (1998, p. 332).

Individual solution for component design means coding programs to implement the features – the so-called "small-scale development"; component solution for system design means that features are covered by composing existing components into a respective configuration – the so-called "large-scale development". Off-the-shelf solution means migrating to standard applications and stabilizing them; outsourcing means defining service level agreements and contracting 3rd party suppliers.

In the experience report presented here, the client wanted to reconsider a strategic IT decision that can be explained as a choice for a development level in the multi-path process model. The client was a very large national telecommunications monopolist, and one of the leading full-service communications providers in the whole region. The client's firm was a multi-company corporation and offered an extensive product and service portfolio including landline and mobile voice communications, data communications including the Internet, complex and custom made corporate solutions, call centers, software development, clearing house services, sea cables and satellite operations, pay TV, smart card manufacturing, etc.

The mission of the client was to replace most parts of the existing home-grown BSS tapestry with a fully integrated corporate wide (i.e. inter-organizational) solution. To realize this, the client started to individually develop respective software from scratch. This process commenced about two years before the reported IT strategy consulting project. The mission had top level management attention at any time, but it still had repeatedly missed its deadlines and had failed to deliver. The corporation finally engaged the consultant to assess the ongoing development, to create an alternative planning based on the integration of software products procured on the market, and finally to compare the running ”Make” integration project with ”Buy” integration planning expected from the consultant.

2 Project approach and selected results

The client wanted the consultant to support a strategic IT decision regarding the integration of the new BSS solution: continuation of the long running individual development mission that had repeatedly failed to deliver, or switching to the integration of ready-made solutions bought on the market. The client's core drivers for the intended BSS solution were, in order of decreasing relevance: functionality, flexibility, risks, cost, and time to market.

The retrieval and generation of decision relevant information by the consultant was structured in several work areas, among them the following two which can be presented in more detail in this paper:

- Functional comparison of available packages.
- Integration scenario case studies.

The consulting was based on the information provided by the client for the ongoing development and integration project creating a bespoke solution, and on the consultant's expertise in integrating systems made from predefined parts.

2.1 Functional comparison of available packages

The consultant created a comparison between the functionalities provided by five packages available on the market, and the functionalities of the intended individually developed solution. Due to the very broad scope of required functionality, the five market packages chosen by the consultant each consisted of vendor package offers made up from a number of each vendor's products plus further pre-selected components plugged on top.

Basis for the functional evaluation were the client's extensive requirements specifications that had been created to develop the intended bespoke solution from scratch. The classification of all requirements into 20 high level characteristics, or feature sets, were based on these requirements specifications and were created by the consultant together with the client. The final feature sets covered quite the complete range of business support that a large full-service communications provider needs. The consultant's experience and some theoretic suggestions from literature (Tam & Tummala 2001) complemented the requirements where it was necessary. To give an idea, the feature sets were for example payments management, workforce management, product and service management, etc.

The evaluation itself was based on a detailed questionnaire with roughly 1'200 single assessment items which were derived directly from the feature sets. The assessment items measured, within each feature set, if the requirements element in question was fulfilled by the examined solution, or not. The positive items inside a predefined feature set were counted and the percentage against all items in the feature set was calculated. In Figure II-B2-2, this overall percentage of fulfillment shown as distance from the center of the diagram, with the 20 feature sets shown as segments of the circle.

The survey was conducted by filling in the questionnaire for each of the five packages from the market plus the individual solution, and this examination was done for each of the six analyzed solutions and for each feature set. In Figure II-B2-2, the result of this evaluation is shown. The strong dashed line represents the functional scope of the individual development and the five thin lines each represent the functional scope of one package that could be procured on the market.

The largest difference between the best package from the market and the individual solution with a major discrepancy of 36 percentage points difference (feature set no. 11) was in disadvantage of the bespoke solution. The second largest difference (no. 18), with 22 percentage points from the leading market package, was again in disadvantage of the individual development. The third largest difference (no. 12), with 17 percentage points from the leading market package, was again in disadvantage of the individual development. Out of the 20 characteristics assessed, ten favoured one or the other package solution and six favoured the individual solution, with four draws. Depending on the actual metrics used for comparison, the result can look a little different but is always in favour of the packages from the market.

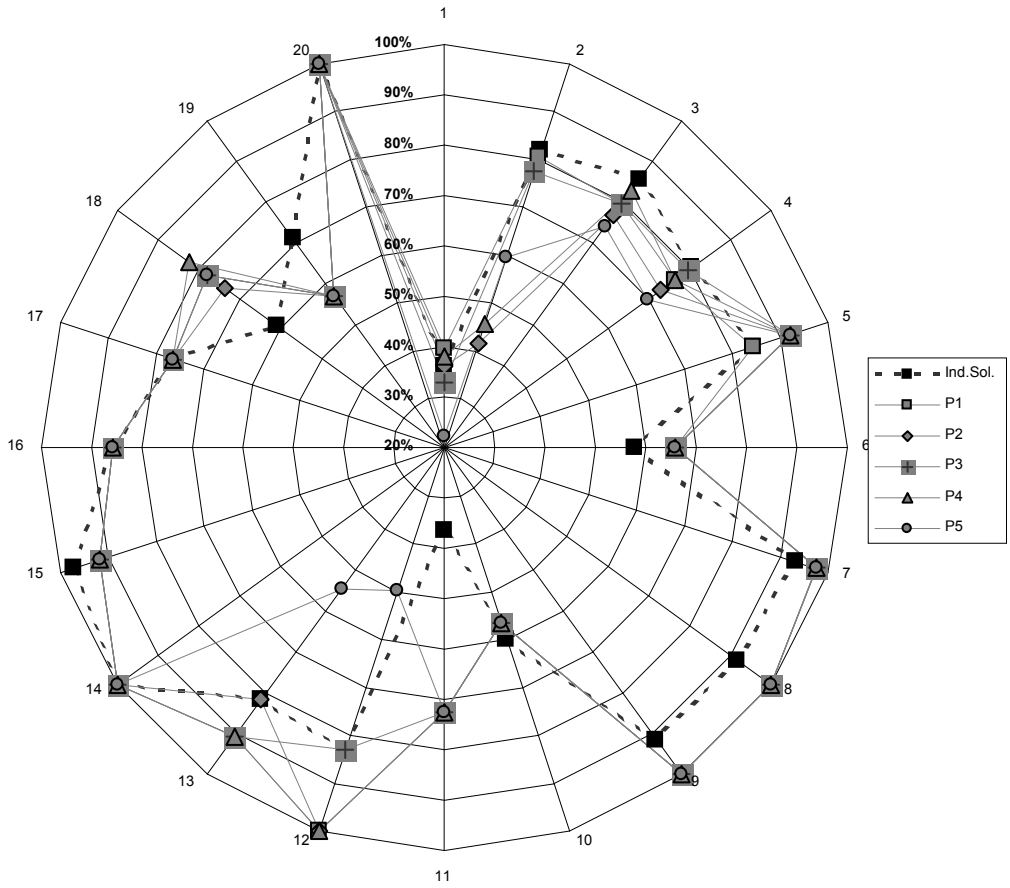

Figure II-B2-2: Functional evaluation, available packages vs. individual solution.

Note that the individual solution initially was expected to cover all feature sets extremely well, since the feature sets were defined from the original requirements that also drove the development of this very solution. Note further that the five package solutions represented functionality that was actually available, while the individual solution was an unfinished work in progress at the time of the analysis, and the recorded functionality was the system's intended functionality once development work was completed. Against initial expectations, this comparison indicated that the functionality

of software packages from products and components available on the market had a better fit with the client's requirements than the bespoke solution developed specifically for these requirements.

Discussing the results of this functional evaluation, the client perceived the following core topics as also influencing the decision:

Complexity. "Components tend to be complex because they implement many features, or because the features are difficult to implement properly, or both. Complexity arises from the fact that a component vendor must convince us that it is better to buy the component than it is to build it."

Dependency. "Producing high-quality implementations is an expensive and in business terms risky undertaking. It is obvious, then, that component vendors strive to make us depend upon their components to protect their revenues as we purchase software support and component upgrades. Thus components tend to be highly product-specific, and also the all-important task of integrating different components becomes more difficult."

Hyper-competition. "In the component industry successful features are quickly copied by competitors. This forces the original vendor to seek new ways to differentiate its component, leading to a new round of innovation, and so forth. The hyper-competitive nature of the component market made commercial software technology reach capabilities today that could only be dreamed of only few years ago. However, such pace of innovation ensures that whatever component competence we obtain is sure to become stale within surprisingly short time. Component competence, then a key organizational asset, wastes rapidly in a hyper-competitive environment."

Double constraints. "A fully integrated system made from available components is constrained twice: first by requirements of our end users and second by capabilities of available components. Today it is almost certainly hopeless to assume that somewhere in the marketplace we find a collection of commercial products that happen to fit perfectly with our needs." This perception is interesting especially vs. the evidence of the functional evaluation.

Pragmatism. "Component evaluation has a new element of pragmatism. We assess requisite functional capabilities that we need, but we also look at what else the component might do. An unexpected and useful feature might lead us to reconsider the overall system design."

2.2 Integration scenario case studies

The consultant provided seven comparable case studies of renowned incumbents describing their choice between "Make" or "Buy" based integration. The consultant selected the different scenarios from references comparable with the client's situation.

Figure II-B2-3 shows the case studies, indicating the classification as development or procurement of software, a classification of the overall project success, and also the project size (symbolic). Three of the described cases were "Make" integration scenarios (A2, A3, A6) and four cases were "Buy" integration scenarios (A1, A4, A5, A7). Five of the projects reached the objectives (A1, A3, A5, A6, A7) and two did not (A2, A4). These case studies were intended to support a concrete client in an actual decision. They do not bear any statistical significance since they were no random sample but deliberately chosen to match with certain aspects of the client's situation. This means also that conclusions such as "bigger projects tend to fail more often" should not be drawn from the case studies.

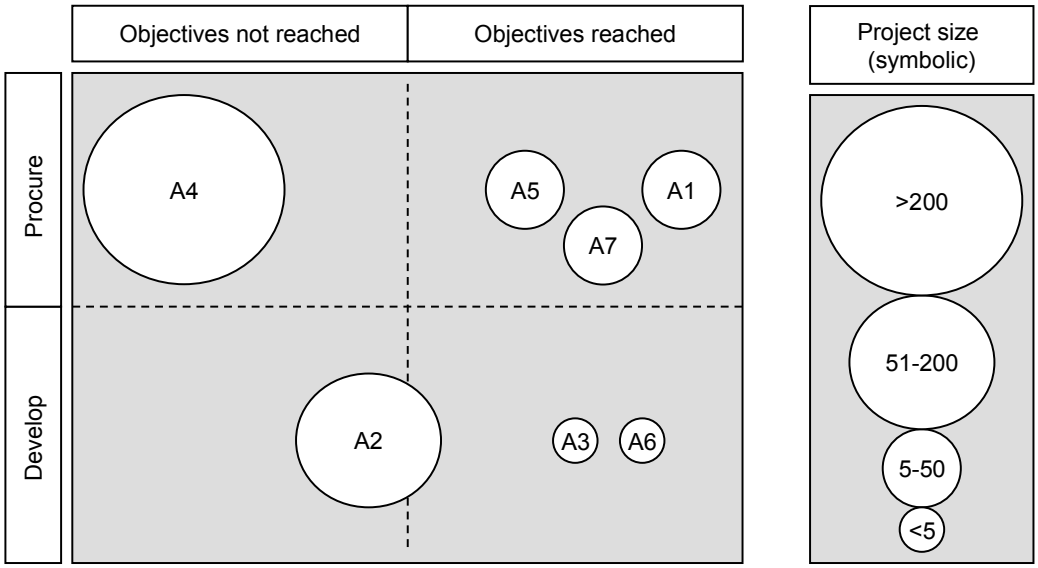

Figure II-B2-3: Seven case studies of software integration scenarios.

A1 had been chosen because the company had a very high number of IT staff in relation to the total number of employees. A1 integrated an externally procured system for business customers, introducing also the new business processes enabled by the new system in parallel. The "Buy" decisions were greatly based on the internal development units not being able to realize the project within the necessary time frame. The project succeeded but the original timing could not be kept, mainly due to migration problems from unexpectedly low source data quality and poor and missing documentation of the source systems.

A2 had been chosen because the company had approximately the size of the client in terms of employees and customers. The intended "Make" solution was based on the re-engineering and functional extension of an existing and inherited proprietary system. In A2, the new system went live at the deadline and worked successfully and error-free due to a sophisticated and elaborate testing from the very beginning. However, the system initially did not deliver its full functionality – in fact, significantly less functionality was available than in the old system before, making users truly unhappy. The intended functionality could only be realized in a number of follow-on releases.

A3 had been chosen because the intended individual solution was very similar to a part of the client's existing IT structure. The solution was built on individually developed software added up with few small externally procured and proprietary products. A3 objectives were mere technical, not functional, including a massive performance improvement of batch throughputs and online response times, and A3 succeeded with a "Make" approach on a very low budget. An extended plan of introducing at the same time functional extensions was blocked by IT management due to limited development resources internally and on supplier side.

A4 had been chosen because the company had a size and structure very similar to the client's. The intended integration in A4 was pure "Buy" with a mixture of products bought from different vendors. A4 tried to restructure the whole enterprise business procedures and integrate the related systems in parallel. The plan failed, even with massive additional external support. Transition and data migration took longer than ever expected. Essential knowledge went to external resources and put A4 in long-term supplier dependencies.

A5 had been chosen because the company had ambitious targets and was located in a cultural environment similar to the client's. Sixteen percent of the personnel were IT staff, and the "Buy" decision replaced an outdated system not any more maintained by the vendor. A5 delivered technically, and in particular the project managed to migrate the legal master data successfully. However, the TCO (total cost of ownership) quickly exceeded the initial software life cycle plan.

A6 had been chosen because it is a successful example for revamping the enterprise business procedures and at the same time integrating a new system that is able to handle the future processes. Ten percent of the company's staff were IT employees, and the "Make" decision included the plan for the software to be maintained and further developed also by external IT staff. However, this case was a comparatively small company, not a full communications portfolio provider as were the other examples.

A7 had been chosen because the company was in a very similar strategic and competitive situation. The "Buy" integration replaced several de-central legacy systems on scattered databases by one central software bought on the market. In parallel the IT organization structure was consolidated, too. The in-house organizations that had developed and maintained the previous systems were set to support the new product. A7 succeeded within the calculated CAPEX but exceeded the time frame by far – while the timing was not the top priority for A7.

For the support of the client's decision, discussions of the presented cases revealed certain of the client's perceptions:

- "Design focus shifts towards fitting pieces together rather than defining internal structures of single functions. No unit tests or inspections of packaged software, because it does not come with source code."
- "Interaction with vendors greatly increases and occurs at different levels throughout a project."
- "Procurement requires more technical knowledge so it is not a pure administrative activity, but technical personnel are often not prepared to deal with procurement issues."
- "Product evaluation becomes a core activity – but developers are not always prepared for it."
- "The amount of bought solutions drives different processes. A project that only uses one large procured package follows completely other processes as compared to an integration of numerous packages that will constitute most of the resulting system."

Discussing the case studies with the client, perceived advantages of "Buy" integration decisions were:

Flexibility: "There is usually some room to adjust requirements to fit the products being used." *Programming*: "Large portions of the system are constituted by a ready-made product and thus do not have to be written and debugged." *Life cycle*: "Possibly because of schedule pressure, 'Buy' integration projects seem to be completed more quickly." *Adherence to schedules*: "There is a perception that schedules are kept better in 'Buy' integration projects, although this cannot be confirmed empirically." *Useful functionality*: "Functionality is discovered in a package that was useful, even though the project had not originally planned to use it."

Discussing the case studies with the client, perceived disadvantages of "Buy" integration decisions were:

Knowledge: "Good or bad surprises having to do with the quality or functionality of a ready-made product." *Communication*: "The vendor constituted one more party with whom communication channels had to be established and maintained." *Dependencies*: "Project personnel had to rely on the vendor for a variety of mainly technical issues." *Negotiations*: "Technical personnel were not always prepared to deal with the business aspects of purchasing and managing a software product."

Further to the described points, practically all projects show that the success of integration depends mostly on the qualification of the involved people, on the actual project set-up and the financial and managerial backing – taking into consideration local culture and principles.

3 Conclusion and remarks

This experience report presented an extract from an IT strategy consulting in a situation that was mission critical for the client. The client requested decision support with a very large individual software development and integration project that had repeatedly failed to deliver. The client requested to propose the alternative of procuring and integrating respective software packages on the market, and to compare this alternative against the finalization of the ongoing individual development and integration. Several work areas were part of the consultation, and two of them were selected for this report and explained in more detail.

A functional comparison was made between the bespoke solution development and five packages assembled from products and components available on the market. Different from expectations, this comparison favoured the integration of package solutions. Further discussions with the client on this topic, specifically on compositional reuse, revealed some reluctance of the client against the component idea for perceived reasons of complexity, dependency, hyper-competition, double constraints and pragmatism.

Case studies that were relevant in certain aspects for the client's decision situation were selected, described and discussed with client management. On the bottom line, the cases studies gave an inconsistent picture for the integration decision and it was concluded that further key success factors, other than the question whether integration is driven by development or by composition, had a strong influence on the cases. The client recognized both pros and cons of the integration approaches, as well as the inconclusiveness of restricting the overall picture to that question.

In the presented report, the consultant's task was restricted to the alternative of full package integration only. Consequently, compositional solutions could only indirectly

be included in the decision support, namely as parts of full packages. Otherwise, as visualized in Figure II-B2-1, a combination of large-scale and small-scale development, also in the sense of a "make and buy" approach (Kurbel et al. 1994), might have been a promising idea to assess.

References (B2)

Becker, S.; Overhage, S. (2003), "Stücklistenbasiertes Komponenten-Konfigurationsmanagement", *Tagungsband 5. Workshop komponentenorientierte betriebliche Anwendungssysteme*, Gesellschaft für Informatik, 25-26 February 2003, Augsburg: 17-32.

Biggerstaff, T.; Richter, C. (1987), "Reusability Framework, Assessment, and Directions", *IEEE Software*, 4 (2): 41-49.

Ewers, J.; Jaekel, T.; Janson, M.; Skroch, O. (2004), "The Impact of EU Liberalization on Telecommunication Service Providers in EU Applicant Countries", Detecon International, Bonn.

Frost and Sullivan (2003), "World Communication Billing Software Market Analysis", Frost and Sullivan, San Jose, USA.

Kurbel, K.; Rautenstrauch, C.; Opitz, B.; Scheuch, R. (1994), "From 'Make or Buy' to 'Make and Buy': Tailoring Information Systems Through Integration Engineering", *Journal of Database Management*, 5 (3): 18-30.

McIlroy, M. (1969), "Mass Produced Software Components", *Software Engineering: Report on a conference sponsored by the NATO Science Committee*, NATO Scientific Affairs Division, 7-11 October 1968, Garmisch: 138-155.

Mili, H.; Mili, F.; Mili, A. (1995), "Reusing Software: Issues and research directions", *IEEE Transactions on Software Engineering*, 21 (6): 528-562.

Ortner, E. (1998), "Ein Multipfad-Vorgehensmodell für die Entwicklung von Informationssystemen – dargestellt am Beispiel von Workflow-Management Anwendungen", *Wirtschaftsinformatik*, 40 (4): 329-337.

Pyshkin, K. (2003), "Operator Strategies and Key Performance Indicator Benchmarks", Analysys Research, Cambridge, UK.

Skroch, O.; Turowski, K. (2006), "Technische Grundlagen von Voice over IP", in Büllesbach, A.; Büchner, W. (eds.), *IT doesn't matter!? – Aktuelle Herausforderungen des Technikrechts*, Schriftenreihe Informationstechnik und Recht der Deutschen Gesellschaft für Recht und Informatik, Volume 15, Otto Schmidt, Cologne: 17-32.

Szyperski, C. (1998), *Component Software: Beyond Object-Oriented Programming*, Addison Wesley, Harlow, UK.

Taubner, D. (2005), "Software-Industrialisierung", *Informatik Spektrum*, 28 (4): 292-296.

Turowski, K. (2003), *Fachkomponenten: Komponentenbasierte betriebliche Anwendungssysteme*, Shaker, Aachen.

Tam, M.; Tummala, R. (2001), "An application of the AHP in vendor selection of a telecommunications system", *Omega – The International Journal of Management Science*, 29 (2): 171-182.

III Spezifikation

Beitrag B3: **Die Bedeutung der Anforderungsspezifikation für erfolgreiche IT-Projekte**

Autoren: Michael Pruß
Dr. Wißner & Pruß EDV-Sachverständige
Im Tal 12, D-86179 Augsburg
pruss@wissner-pruss.de

Oliver Skroch
Lehrstuhl für Wirtschaftsinformatik und Systems Engineering
Universität Augsburg, Universitätsstr. 16, D-86159 Augsburg
oliver.skroch@wiwi.uni-augsburg.de

Veröffentlichung: HMD – Praxis der Wirtschaftsinformatik (zur Veröffentlichung angenommen am 27. Juli 2009).

IT ist heute nicht nur alltäglich und unverzichtbar, sondern auch einer der wichtigsten Wettbewerbsfaktoren im Wirtschaftsgeschehen. Besondere Wettbewerbsvorteile lassen sich für Unternehmen durch individuelle Alleinstellungsmerkmale am Markt erzielen. Informationstechnik muss diese unterstützen bzw. sie überhaupt erst ermöglichen. Für deren Planung, Beschaffung, Erstellung und Betrieb bedarf es im effizienten, arbeitsteiligen Wertschöpfungsprozess möglichst klarer Spezifikationen der genauen Anforderungen. Alle arbeitsteilig beteiligten Parteien – darunter etwa Anwender, Rechenzentren, Systemintegratoren oder Entwickler – profitieren dabei von eindeutigen, belastbaren Vereinbarungen, die die vielfältigen Projektrisiken entlang immer globaler werdenden Wertschöpfungsketten verringern.

Trotzdem sind in der Leistungsbeschreibung, dem inhaltlichen Kern solcher Vereinbarungen, wichtige Gesichtspunkte zwischen den Parteien regelmäßig ungeklärt. Erhebliche taktische Wettbewerbsvorteile bleiben dadurch ungenutzt – oder werden sogar ins Gegenteil verkehrt, wenn Projekte scheitern. Im folgenden Beitrag werden daher die Erfolgsfaktoren identifiziert, die den Wettbewerbsvorteil durch gute Anforderungsspezifikationen und Leistungsbeschreibungen in der Praxis des betrieblichen Alltags ermöglichen. Die Risiken, die mit ihrer Vernachlässigung verbunden sind, werden dann bis hin zum Rechtstreit anschaulich illustriert. Der Beitrag berührt damit interdisziplinäre Fragen an der Schnittstelle zwischen Wirtschaftsinformatik und Recht.

1 Anforderungsspezifikationen im Lösungsprozess

Am Anfang aller systematischen Entwicklungsprozesse stehen die Anforderungen (Lasten) des Auftraggebers als entscheidende Grundlage und Treiber der weiteren Aktivitäten. In arbeitsteiligen Wertschöpfungsketten müssen diese Anforderungen an die Beteiligten kommuniziert werden. Denn nur über klare Vorgaben, *was* erreicht werden soll, findet sich, *wie* es erreicht werden kann. Es ergibt sich also die Notwendigkeit hochwertiger, im Idealfall sogar standardisierter Anforderungsspezifikationen. In der arbeitsteiligen Softwareentwicklung sind solche klaren Vorgaben eine Voraussetzung für die erfolgreiche Komposition von Gesamtlösungen aus Teilen unterschiedlicher Herkunft (Grollius, Lonthoff & Ortner 2007, S. 40-42). Klare Anforderungsspezifikationen, mit denen die zugesicherten Eigenschaften einer gewünschten IT-Lösung aus der reinen Außensicht an den Schnittstellen der Systemteile spezifiziert werden, können zwischen den Beteiligten vertraglich als Leistungsbeschreibungen vereinbart werden. Sie sind damit der vielleicht wichtigste einzelne Schlüsselfaktor in modernen IT-Prozessen (Gsell, Overhage & Turowski 2008, S. 47f).

In der IT-Praxis stellt man aber fest, dass ausgerechnet die Bedeutung der Anforderungsspezifikation von Entscheidern in Unternehmen oder Institutionen vielfach noch nicht voll erkannt wird. Bei Entwicklung, Auswahl und Einführung von IT-Systemen spielt die Qualität der Anforderungsspezifikation deshalb häufig nur eine untergeordnete Rolle. Beispielsweise wurden bei einer Untersuchung von IT-Großprojekten der öffentlichen Verwaltung Schwierigkeiten mit den Spezifikationen als ein Kernproblem identifiziert; allein beim größten der untersuchten Projekte wurde ein entstandener Schaden von ca. fünf Milliarden Euro (einschließlich Opportunitätskosten) geschätzt (Mertens 2009, S. 44, S. 46). Ähnlich gelagerte Probleme aufgrund von Spezifikationen lassen sich regelmäßig in Unternehmen jeder Größe ausmachen. So hat kürzlich ein mittelständischer Hersteller von individuell gefertigten Glasbauteilen ohne detaillierte Anforderungsspezifikation ein kommerzielles Softwareprodukt für die Glasbaubranche erworben. Es stellte sich dann heraus, dass die Software ausgerechnet die spezifischen Aufgaben der Individualfertigung komplexer Glasteile gerade nicht berücksichtigt. Das Projekt führte zu einem juristischen Schlagabtausch.

2 Erfolgsfaktoren

Für die Darstellung der Erfolgsfaktoren zur Erstellung klarer Anforderungsspezifikation unterscheiden wir zunächst die beteiligten Parteien. Der Auftraggeber hat in einem typischen IT-Projekt mit verschiedenen Beteiligten zu tun, z.B. Hersteller, Berater, Systemintegratoren und Qualitätssicherer. Mitentscheidend für die erfolgreiche Zusammenar-

beit aller Beteiligten ist die zu vereinbarende Leistungsbeschreibung mit den zugesicherten Systemeigenschaften, die direkt aus den Anforderungen des Auftraggebers hervor geht. Im Folgenden werden die praktischen Herausforderungen der Anforderungsspezifikation auf Auftraggeberseite betrachtet (Abbildung III-B3-1).

2.1 Systematische Vorgehensweise

Bei der Bereitstellung individueller IT-Lösungen fallen Entwicklungsaufgaben an, die nur systematisch zu bewältigen sind. Systematische technische Entwicklung vollzieht sich von Lasten (Anforderungen) über Pflichten (Design) zu Realisierung und (Abnahme-)Test, wobei die Arbeitsschritte in verschiedener Weise iterativ, verteilt und von qualitätssichernden Maßnahmen begleitet sein können. In der Praxis angewandte Vorgehensweisen weichen beliebig von den theoretischen Vorgehensmodellen ab, sie sind aber dann erfolgreicher, wenn sie sich an einem systematischen Prozess orientieren. Allen Beteiligten sollten deshalb die systematischen Vorgehensweisen bekannt sein.

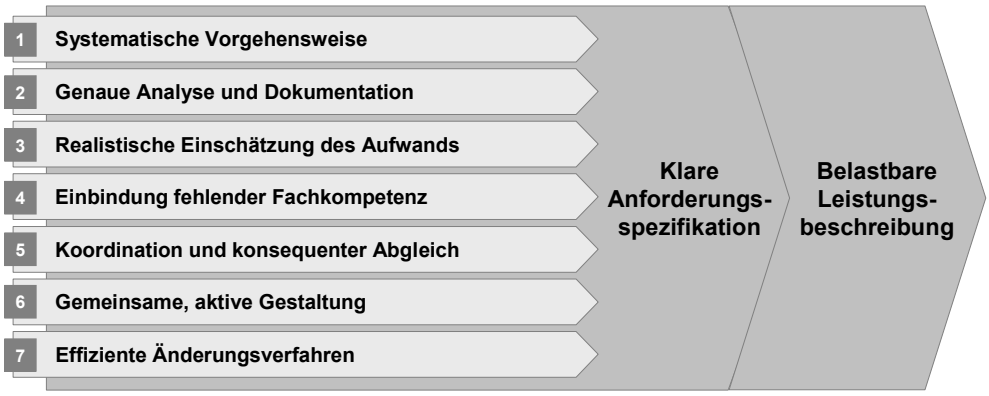

Abbildung III-B3-1: Erfolgsfaktoren für klare Anforderungsspezifikationen.

Anwender sollten in diesem Zusammenhang auch mit den semiformalen Methoden zur Modellierung von Anforderungen vertraut sein, die im Umfeld der Softwareentwicklung in der Wirtschaftsinformatik verwendet werden. Beispiele sind die ereignisgesteuerte Prozesskette (EPK) oder die Unified Modeling Language (UML).

In der Praxis werden hierfür oft IT-Mitarbeiter eingesetzt, ohne deren unterschiedliche Tätigkeitsprofile zu berücksichtigen. Softwareentwicklung unterscheidet sich aber stark z.B. vom Rechenzentrumsbetrieb oder dem IT-Management. In den IT-Abteilungen eines großen deutschen Fachverlags waren beispielsweise eigene Fachleute für Softwareprojekte (Berufsbild z.B. Systemanalytiker, Projektleiter) eingespart worden. Es gab daher kaum interne Kenntnisse und Erfahrungen in Entwicklungsaufgaben und im Erstellen von individuellen Anforderungen. Dennoch wurden den IT-Mitarbeitern (erfolglos) entsprechende Ergebnisse abverlangt.

Konfigurations- und Parametrisierungstätigkeiten bei Softwareprodukten von der Stange sind zunächst von der Individualentwicklung abzugrenzen. Bei komplexen IT-Produkten kann aber bereits die Parametrisierung so schwierig sein, dass sie ebenfalls nur mit systematischem, entwicklungsgleichen Vorgehen beherrschbar ist. Bei kommerziellen Produkten ist weiterhin ein Auswahlprozess aus einer Menge von Angeboten erforderlich. Ad hoc Ansätze führen auch hier allenfalls zu seltenen Glücksgriffen, können aber nicht systematisch eine für die eigenen Anforderungen geeignete Auswahlentscheidung sicherstellen.

2.2 Genaue Analyse und Dokumentation

Anwender, die die zentrale Bedeutung der eigenen Anforderungen erkannt haben, sollten dadurch motiviert sein, sich ausführlich mit ihren Anforderungen zu befassen und sie inhaltlich detailliert zu dokumentieren. Die Anforderungen sind dabei auf einem einheitlichen Detaillierungsgrad zu sammeln, der vorzugeben ist. Es kann sonst passieren – wie es z.B. in einem großen deutschen Krankenhaus der Fall war – dass engagierte und entsprechend ausgebildete Mitarbeiter detailreich spezifizieren, andere dagegen nur oberflächlich arbeiten.

Die Anforderungsspezifikation sollte so einfach wie möglich gehalten werden – aber nicht einfacher. Anwender halten erfahrungsgemäß die ihnen besonders vertrauten Anforderungen (Begrifflichkeiten, Abläufe, usw.) für selbstverständlich und spezifizieren diese nicht. Besonders die Eigenschaften einer schon vorhandenen Software werden implizit erwartet, ohne ausdrücklich zu formulieren, welche dies sind. Es sind viele Fälle bekannt, in denen Anwender sogar nur die einzige Anforderung stellen, dass die gesetzlichen Regelungen eingehalten werden müssen. Sie erliegen damit dem Trugschluss, dass die Eigenschaften aus einer ihnen vertrauten, z.B. aktuell abzulösenden Software ebenfalls zu liefern wären. Es hat aber die Nachfrage beispielsweise nach Finanzbuchhaltungssoftware, die den gesetzlichen Anforderungen entspricht, am Markt zu ganz unterschiedlichen, gesetzeskonformen Produktlösungen geführt. Gerade die individuellen Unterschiede machen bestimmte Lösungen mehr oder weniger vorteilhaft.

Aus der Anforderungsspezifikation geht letztlich die Leistungsbeschreibung als Vertragsbestandteil hervor. Diese wird z.B. im Werkvertrag dann auch zur inhaltlichen Grundlage der juristisch verbindlichen Abnahme. Eine erste Näherung für die geeignete Genauigkeit der Leistungsbeschreibung ist also die Möglichkeit, konkrete und individuelle Testfälle für eine Abnahme herleiten zu können. Idealerweise eignet sich die Leistungsbeschreibung sogar zur selbständigen Anfertigung des entsprechenden Systemdesigns (Pflichtenhefts), mit möglichst geringem Klärungsaufwand und minimalem Inter-

pretationsspielraum durch die Lieferanten / Hersteller in nachgelagerten Entwicklungsschritten.

2.3 Realistische Einschätzung des Aufwands

Selbst eine Spezifikation, die nur die wesentlichsten Anforderungen enthält, kann schnell aus mehreren Tausend Einzelanforderungen bestehen. Der Aufwand für die Erstellung eines so umfangreichen, individuellen Dokuments wird in der Praxis gerne unterschätzt. Als extremer Fall kann eine Investorengruppe erwähnt werden, die der Ansicht war, es sei innerhalb von drei Arbeitstagen möglich, alle Anforderungen an die IT einer landesweit operierenden Firma mit mehreren Tausend Mitarbeitern rechtssicher zu dokumentieren.

In diesem Zusammenhang sollten zunächst vor allem ungeeignete Ausweichstrategien vermieden werden. Eine der Ausweichstrategien ist erfahrungsgemäß das unkritische Vertrauen, das Auftraggeber in die Kompetenz der Lieferanten oder Berater setzen. Suggerieren diese aus vertrieblichem Interesse, dass alle Anwenderprobleme bekannt wären und passende individuelle Lösungen bereit lägen, wird dadurch die Bereitschaft der Auftraggeber zur ausreichenden Beschäftigung mit ihren tatsächlichen Anforderungen weiter gesenkt. Die Praxis zeigt jedoch, dass gerade die Individualität eines Unternehmens stets neue und ungeahnte Herausforderungen stellt.

Eine andere typische Ausweichstrategie liegt vor, wenn Mitarbeiter der Fachbereiche damit betraut werden, Anforderungsspezifikationen nebenbei zu erstellen, während sie gleichzeitig mit ihrem üblichen Tagesgeschäft schon voll ausgelastet sind. Die Analyse und Dokumentation von individuellen Anforderungen ist aber eine anspruchsvolle und mühsame Arbeit, die hohe Konzentration und intensive Kommunikation erfordert, die daher nebenbei kaum ordentlich zu erbringen ist.

Der hohe Aufwand für die Analyse und Dokumentation der Anforderungen wirkt oft abschreckend, aber die Beurteilung einer Spezifikation darf sich nicht nur an Aufwands- und Kostenaspekten orientieren. Um wirtschaftlich sinnvolle Entscheidungen zu ermöglichen, müssen wie bei jeder anderen Investition auch hier der Wertbeitrag bzw. Nutzen sowie die Risiken im Rahmen der wirtschaftlichen Gesamtbedeutung der beabsichtigten Lösung mit berücksichtigt werden (DeMarco 1997, S. 30-33).

2.4 Einbindung fehlender Fachkompetenz

In praktisch allen Technikbereichen ist es üblich, die Anforderungen an ein komplexes zu entwickelndes Produkt oder an eine umfangreiche zu erbringende Leistung durch qualifizierte Fachleute spezifizieren zu lassen, und bei größeren Vorhaben sogar einen ganzen Planungsstab einzusetzen. Wie die Erfahrung zeigt, müssen die planenden Fachleute erhebliche Qualifikationen mitbringen. Neben allgemeinen methodischen Fähigkeiten und Know-how der eigenen technischen Fachdisziplin (z.B. Maschinenbau) sind zusätzlich auch ihre spezifischen Kenntnisse der jeweiligen Zielbranche (z.B. Automotive) erforderlich.

Im erwiesenermaßen besonders schwierigen Bereich der Softwareentwicklung ist fundiertes Fachwissen und Erfahrung, und damit auch die Einbindung von professionellen Experten, umso mehr erforderlich. Nur wird dies paradoxerweise für entbehrlicher gehalten als in vergleichbaren, etablierten technischen Disziplinen. Mehr Verständnis und Professionalität wird daher bereits von vielen Seiten gefordert.

2.5 Koordination und konsequenter Abgleich

Sind vom Anwender detaillierte Anforderungen spezifiziert, so ist darüber hinaus eine geeignete Abstimmung der Anforderungen zwischen den beteiligten Stakeholdern in ein schlüssiges Gesamtpaket notwendig. Widersprüche und Lücken sind aufzudecken. Es sind auch wenig sinnvolle, äußerst aufwändig umzusetzende Anforderungen ohne besonderen Nutzen zu identifizieren. So wurde in einem Bildungsunternehmen der bayerischen Verwaltung von einer Fachabteilung gefordert, Zeugnisse durch eine Mikroschrift mit Spezialdrucker, -toner und -papier fälschungssicher zu machen. Bei einem Abstimmungs-Review wurde klar, dass der Nutzen in keinem Verhältnis zu den enormen Kosten stand. Solche übertriebenen Forderungen einzelner Fachabteilungen müssen identifiziert und entfernt werden, damit sie nicht unreflektiert in einer Leistungsbeschreibung landen.

Erfahrungsgemäß ergibt sich in dieser Phase auch die Gelegenheit, anhand der Anforderungsanalysen falsche oder fehlerträchtige Betriebsabläufe zu identifizieren und entsprechende Konsequenzen zu ziehen. Geschäftsprozesse können dann explizit mit den Anforderungen abgestimmt werden. Dieser Aspekt wird speziell im Rahmen des Aufbaus serviceorientierter Architekturen betont.

Besonders bei umfangreichen Anforderungen und entsprechend großen Lösungen kann auch der ggf. später stattfindende Ausschreibungsprozess besser auf der Grundlage von

gut abgestimmten und sinnvoll gegliederten Anforderungskatalogen durchgeführt werden.

2.6 Gemeinsame, aktive Gestaltung

Holt sich ein Auftraggeber externen Sachverstand zur Anforderungsspezifikation ins Haus, ist er nicht von notwendiger Mitwirkung befreit. Es ist erforderlich, dass Mitarbeiter aus den Fachbereichen und externe Berater die Anforderungsspezifikation in enger und offener Abstimmung gemeinsam im Team erstellen, da die Externen die individuellen Besonderheiten des Auftraggebers nicht kennen und den Internen das IT- und Methoden-Know-how fehlt. Diese gemeinsamen Aktivitäten sind zu kontrollieren und aktiv zu steuern, andernfalls können am Ende alle möglichen Überraschungen auftauchen. Es ist aber das primäre Interesse aller Beteiligten, dass ein echtes gemeinsames Verständnis über die individuell erwartete Leistung erreicht und inhaltlich nachvollziehbar spezifiziert wird.

Als Negativbeispiel kann ein großer Infrastrukturanbieter außerhalb Deutschlands dienen, bei dem ein geschäftskritisches IT-Entwicklungsprojekt nach mehreren Jahren Projektdauer immer noch keine sichtbaren Ergebnisse lieferte. Es gab dort u.a. keine dokumentierten Anforderungen und keine aktuellen Projektpläne, dafür aber eine Projektkultur, in der das offene Ansprechen von Problemen und das Einfordern von Entscheidungen kaum möglich waren.

2.7 Effiziente Änderungsverfahren

Änderungsverfahren sind meist vertraglich für das spätere Projekt vereinbart. Sie sind daher nicht unbedingt ein Erfolgsfaktor für Anforderungsspezifikationen im engeren Sinn. Jedoch kann es selbst bei vergleichsweise hochwertigen Spezifikationen die Notwendigkeit geben, ursprünglich vereinbarte Leistungen im späteren Projektverlauf anzupassen. Je unklarer oder instabiler die Leistungsbeschreibung aber ist, desto mehr tauchen umstrittene Punkte auf, die durch das Änderungsverfahren müssen. Die ursprünglich nur als Ausnahmeregelung gestaltete Änderungsklausel kann dabei schnell zum dauerhaften Vorgehen werden. Das stellt eine erhebliche Zusatzbelastung dar, und es kann sogar soweit gehen, dass laufende Änderungsverfahren den weiteren Fortgang eines Projekts insgesamt blockieren. In der Praxis ist die Situation dann meist nicht mehr handhabbar.

Erfahrungsgemäß empfiehlt sich für Änderungsverfahren bei schlechten Anforderungsspezifikationen die Einbindung eines neutralen, von allen Beteiligten akzeptierten Ex-

perten. Dieser sollte nach Möglichkeit von Anfang an dabei sein, um sich nicht erst in Krisensituationen langwierig einarbeiten zu müssen. Der Experte kann dabei als Schiedsrichter mit Entscheidungsbefugnis oder als Mediator ohne Entscheidungsmacht auftreten.

3 Risiken

Um die Bedeutung von hochwertigen Anforderungsspezifikationen und klaren Leistungsbeschreibungen in arbeitsteiligen Prozessen zu verdeutlichen, ist es besonders hilfreich, die Risiken zu betrachten, die von unklaren, fehlenden oder nicht mehr nachvollziehbaren Vereinbarungen ausgehen (Abbildung III-B3-2).

Abbildung III-B3-2: Risiken unklarer Leistungsbeschreibungen.

Diese Projektrisiken bestehen zwar nicht nur in der modernen IT. Mangelhafte Leistungsvereinbarungen verschärfen die Risiken aber aufgrund ihrer zentralen Bedeutung in komplexen, arbeitsteiligen Entwicklungsprozessen im IT-Bereich erheblich. Speziell die Erfahrung mit gerichtlichen IT-Gutachten zeigt zudem, dass operative Konsequenzen zumeist selbst dann unterbleiben, wenn die Bedeutung einer klaren Leistungsbeschreibung prinzipiell erkannt wurde.

3.1 Strittiger und unvollständiger Leistungsumfang

Hat man sich nur wenig mit den eigenen Anforderungen befasst, so kann sich im späteren Projektverlauf zeigen, je nach Vorgehensmodell in unterschiedlichen Phasen, dass es noch kein echtes gemeinsames Verständnis über den Leistungsumfangs gibt. Bei früh erkannten Unstimmigkeiten lassen sich noch Handlungsalternativen finden. Je später im Projekt die Differenzen erkannt werden, desto problematischer wird aber die Situation. In der Praxis erfüllt sich dann die Hoffnung erfahrungsgemäß nicht mehr, dass Projektziele trotzdem noch eingehalten werden, strittige Punkte sich klären und fehlende Entscheidungen noch getroffen werden.

Die Erfahrung zeigt vor allem auch, dass Anwender die große Bedeutung aussagefähiger, eigener Abnahmetests zur Überprüfung des gelieferten Leistungsumfangs verken-

nen. Es ist zwar in der Regel nicht wirtschaftlich, für alle Anforderungen detaillierte Abnahmetestfälle zu definieren, zumindest die geschäftskritischen Anforderungen sind aber durch systematisch vorbereitete Tests abzusichern. Außerdem sind, für die Nachvollziehbarkeit in einer möglichen Auseinandersetzung, reproduzierbare Testergebnisse erforderlich, mit denen ein zu geringer Leistungsumfang oder aufgetretene Fehler zweifelsfrei nachweisbar sind. In der Praxis vertrauen aber Auftraggeber selbst noch in den juristisch relevanten Abnahmetests immer wieder darauf, dass der Lieferant vollständige und stabile Lösungen geliefert hat. Welche Anforderungen tatsächlich erfüllt sind und welche Defizite die Lösung hat, offenbart sich dann erst im Wirkbetrieb mit den entsprechenden Folgen.

Ein positives Beispiel ist hier ein bis heute erfolgreicher privater Kommunikationsanbieter, der rechtzeitig für die Vorbereitung der Abnahmetests und den Aufbau eines Testlabors sorgte. Damit war dieser Anbieter unter den Konkurrenten der einzige, dem die Einführung eines firmenweit integrierten IT-Systems gelang. Die meisten ehemaligen Wettbewerber sind heute vom Markt verschwunden, wobei einer durch die direkten Konsequenzen einer erfolglosen Systementwicklung insgesamt ruiniert wurde.

3.2 Zeitverlust

Mit dem strittigen Leistungsumfang geht auch ein durch die Auseinandersetzungen und klärenden Abstimmungen verursachter Zeitverlust einher. Je weniger eindeutig und belastbar die Vereinbarungen tatsächlich sind, desto mehr Zeit verbrauchen nachfolgende Diskussionen.

Jede Seite hat die von der jeweils anderen Seite unterschiedlich interpretierten Anforderungen inhaltlich neu zu prüfen. Man versucht, sich zu einigen und muss entscheiden, ob vorgeschlagene Lösungen akzeptabel sind, ob Alternativen erforderlich sind, und welche Konsequenzen sich jeweils ergeben. Zeitpuffer sind für solche Klärungsprozesse erfahrungsgemäß vorher kaum eingeplant. Bestimmte Entscheidungen hängen in der Praxis auch an einzelnen Personen oder Gremien, die nicht immer verfügbar sind. Im Ergebnis kann ein Zeitverlust in unvorhersehbarem Ausmaß eintreten.

Viele Projekte sind in externe, nicht beeinflussbare Zeitvorgaben und Fristen eingebunden. Beispielsweise müssen veränderte gesetzliche Vorgaben zu bestimmten Fristen umgesetzt sein oder Termine sind aus Marketingaspekten fest vorgegeben, wie etwa das Weihnachtsgeschäft. In solchen Situationen kann allein der Zeitverlust schon zum Scheitern des Gesamtprojekts führen.

3.3 Mehrkosten

Hat der Anwender bestimmte Leistungskomplexe nicht bzw. unklar spezifiziert, müssen die für ihn dennoch erforderlichen Eigenschaften mit zusätzlichem, bis dahin ungeplantem Aufwand umgesetzt werden. Für diese Aufwände ergeben sich in der Praxis ungünstigere Konditionen als für die im Rahmen des Gesamtprojekts vorab vereinbarten Leistungen. Ein vermeintlich günstigeres Angebot kann sich so im Nachhinein doch noch als vergleichsweise ungünstiger erweisen.

Am deutlichsten sichtbar werden solche Kostensteigerungen im Bereich der sehr großen Anwendungen bei entsprechend großen Auftraggebern (öffentliche Hand, Großkonzerne, usw.), wo in grundlegende IT-Systeme oftmals aus strategischen, politischen oder sonstigen Überlegungen langfristig vorinvestiert wird. Wie die Praxis zeigt, wird dort nicht selten erheblich nachbudgetiert, da es aus den unterschiedlichsten Gründen als besser angesehen wird, mit dem Projekt irgendwie fortzufahren, als es scheitern zu lassen. Gerade hier merkt man oft viel zu spät, dass auch langfristig geplante Ziele nicht erreicht werden können, wenn die Vorhaben von Anfang an unterschätzt und unprofessionell geplant und durchgeführt werden.

3.4 Rechtstreit

Lässt sich eine Einigung im Rahmen der Regelungen des Vertrags oder durch Verständigung zwischen den Beteiligten nicht erzielen, bleibt als letzte Eskalationsstufe die juristische Auseinandersetzung. Dadurch entstehen Kosten, die niemand einkalkuliert hat. Zudem sind Dauer und Ausgang eines Rechtstreits, selbst bei zunächst vermeintlich klaren Anspruchspositionen, kaum vorhersehbar.

Bei unklaren Anforderungen wird vor Gericht regelmäßig als letzter Ausweg der „Stand der Technik bei einem mittleren Ausführungsstandard“ (BGH 2003, S. 1) oder die „anerkannten Regeln der Technik“ (Bayerlein 2008, §16 RdNr. 21) bemüht. Dieser Ansatz mag in anderen Technikbereichen hinreichend sein, in denen es allgemein anerkannte Normen, Vorschriften und Standards gibt. Die IT hingegen leidet unter einer Vielzahl konkurrierender Standards und Technologien, die nebeneinander existieren, jeweils aber weder allgemein anerkannt sind noch durchgängig angewandt werden. So existiert im Software Engineering heute noch nicht einmal eine allgemein akzeptierte Konstruktionslehre. Was in dieser Situation der Stand der Technik oder ein mittlerer Ausführungsstandard bedeutet, bleibt deshalb nicht objektiv fassbar. Für die typischen Anwendungsbereiche und Zieldomänen der Wirtschaftsinformatik ist der Stand der Technik ebenfalls kaum bestimmt. Selbst die (niedrigeren) anerkannten Regeln der Technik fin-

det man nur in wenigen, klar eingegrenzten Anwendungsbereichen wie z.B. der Finanzbuchhaltung in Form der GoBS (Grundsätze ordnungsgemäßer DV-gestützter Buchführungssysteme).

Vor Gericht haben daher regelmäßig Sachverständige aufgrund ihrer persönlichen Erfahrungen und Marktkenntnisse zu bewerten, was für den jeweiligen Fall zutrifft (Jäger et al. 2003, S. 140-142). Die Entscheidungen der Gerichte hängen dabei stark von der Bewertung des Sachverständigen ab. Da der Sachverständige neutral ist, stimmt er nur selten mit den Ansichten der Prozessparteien überein. Spätestens jetzt wird die Bedeutung detaillierter und gemeinsam einheitlich verstandener Anforderungsdokumente allen Beteiligten deutlich.

4 Zukünftige Anforderungen

Wichtige Erfolgsfaktoren begünstigen die Erstellung klarer Anforderungsspezifikationen. Mit ihrer Vernachlässigung sind erhebliche Risiken verbunden. Die Erfahrung zeigt, dass man notorisch daran scheitert, Anforderungen klar zu spezifizieren und damit Leistungsbeschreibungen zu vereinbaren, die in arbeitsteiligen Entwicklungsprozessen eindeutig interpretierbar und vertraglich belastbar sind.

Die Bedeutung rechtssicherer Leistungsbeschreibungen nimmt in den zwei aktuellen IT-Trends serviceorientierte Architektur / Komponenten- und Serviceentwicklung sowie Outsourcing / Offshoring weiter zu. In beiden Ansätzen werden an den Schnittstellen zwischen Systemen, Anwendungen und Organisationen zunehmend nur noch explizite Abhängigkeiten zugelassen. Die Schnittstellen sind dann im zwischenbetrieblichen und interkulturellen Zusammenhang sehr formal zu betrachten und immer weniger Annahmen können noch implizit getroffen werden. Hochwertige Anforderungsspezifikationen und eindeutige Leistungsvereinbarungen rücken damit noch stärker als bisher in den Mittelpunkt.

Sind die erforderlichen Fähigkeiten zur klaren Anforderungsspezifikation nicht vorhanden, so sind sie intern langfristig aufzubauen oder extern einzubinden. Durch hochwertige Anforderungsspezifikationen und die darauf basierenden, vertraglich klar vereinbarten Leistungsbeschreibungen könnten dann im IT-Bereich deutliche Wettbewerbsvorteile erzielt werden, wie es in vergleichbaren, etablierten Ingenieursdisziplinen schon seit langer Zeit selbstverständlich ist.

Literatur (B3)

Bayerlein, W. (2008), *Praxishandbuch Sachverständigenrecht*, 4. Auflage, Beck, München.

BGH (2003), Bundesgerichtshof, Urteil vom 16. Dezember 2003, X ZR 129/01, Abruf am 26. Januar 2009, <http://www.bundesgerichtshof.de>.

DeMarco, T. (1997), *Warum ist Software so teuer? Und andere Rätsel des Informationszeitalters*, Hanser, München.

Grollius, T.; Lonthoff, J.; Ortner, E. (2007), „Softwareindustrialisierung durch Komponentenorientierung und Arbeitsteilung“, *HMD – Praxis der Wirtschaftsinformatik*, 44 (256): 37-45.

Gsell, B.; Overhage, S.; Turowski, K. (2008), „Unzureichende Leistungsbeschreibung bei der Softwareentwicklung und die Rolle von Standardverträgen“, in Möllers, T. (Hrsg.), *Standardisierung durch Markt und Recht*, Nomos, Baden-Baden: 23-48.

Jäger, K.; Lenzer, J.; Scheider, J.; Wißner, B. (2003), *Begutachtung und rechtliche Bewertung von EDV-Mängeln*. Wißner-Verlag, Augsburg.

Mertens, P. (2009), „Schwierigkeiten mit IT-Projekten der öffentlichen Verwaltung“, *Informatik-Spektrum*, 32 (1): 42-49.

Beitrag B4: **A case study on requirements specifications and critical compensation factors in offshore application development**

Autoren: Oliver Skroch, Sven Overhage, Klaus Turowski
Lehrstuhl für Wirtschaftsinformatik und Systems Engineering
Universität Augsburg, Universitätsstr. 16, D-86159 Augsburg
[oliver.skroch|sven.overhage|klaus.turowski]@wiwi.uni-augsburg.de

Veröffentlichung: (Zur Begutachtung eingereicht am 14. September 2009)

Entwicklungsaufgaben für Informationssysteme werden zunehmend in Offshore-Ansätzen über betriebliche, nationale und kulturelle Grenzen hinweg global verteilt. Dadurch wird die explizite Kommunikation zwischen allen Beteiligten immer wichtiger und es erhöht sich die zentrale Bedeutung der Anforderungsspezifikation als geeignetes Kommunikationsmittel. Der Erfolg von Offshore-Schritten in der Entwicklung betrieblicher Anwendungssysteme hängt somit verstärkt von der Qualität der entsprechenden Anforderungsspezifikationen ab. Oft sind Spezifikationen aber für die Verwendung im Offshore-Kontext nicht ausreichend geeignet. Entscheidend ist dann, wie man mit Spezifikationsdefiziten im weiteren Entwicklungsverlauf umgeht. Der Beitrag präsentiert eine Fallstudie aus einem großen und realen Industrieprojekt zur Planungs- und Entscheidungsunterstützung hinsichtlich der Offshore-Entwicklung einer komplexen, individuellen betrieblichen Anwendungssoftware.

In der Fallstudie wird – zur Unterstützung des Topmanagements eines der weltweit führenden Industriekonzerne im Automotive-Sektor – eine theoretisch fundierte Methode angewandt, mittels der zunächst die Eignung für Offshore-Entwicklungsansätze einer umfangreichen Spezifikation ausführlich beurteilt wird. Danach werden kritische Kompensationsfaktoren identifiziert, die im weiteren Verlauf der Offshore-Entwicklung ein ausgleichendes Potenzial gegen unzureichende Spezifikationsqualität darstellen können. Im Rahmen der Studie werden zudem die Anwendung der Methode und die mit ihr erzielten Bewertungsergebnisse validiert, indem der Methodeneinsatz selbst analysiert wird und die erarbeiteten Ergebnisse mit dem tatsächlichen weiteren Projektverlauf verglichen werden.

1 Introduction

Assigning IS functions to external partners became a sustainable business model already since 1963 when Electronic Data Systems agreed with the Blue Cross health insurance to completely take over their application systems (Dibbern et al. 2004). Today, the outsourcing of IS development and operation tasks to external contractors has become an established option, and clients increasingly focus on offshoring entire steps of their business application development. Such projects aim at transferring certain development steps to "off shore" regions that are envisaged as being far away, barely regulated, and providing low costs (Aspray, Mayadas & Vardi 2006; Kobitzsch, Rombach & Feldmann 2001; Pryor & Keane 2004). Aspray, Mayadas and Vardi (2006) discuss how economic theories and case studies advocate business advantages for both the clients' and the contractors' regions. On the other hand, additional costs that arise from the offshore development are illustrated (Dibbern, Winkler & Heinzl 2008), and e.g. Overby (2003) estimates up to 10 percent premium only for adopting the development processes to a globally distributed setting.

Distributed development with partners around the globe is embedded in an intercultural context (Winkler, Dibbern & Heinzl 2008). With the typical absence of implicit understandings in such a context, unequivocal requirements specifications gain their particular significance, since they become the means to explicitly communicate required features. Today, global offshoring development approaches are pursued even with highly individual and complex, bespoke systems for which clearly specified requirements are critical anyway. But although the quality of these specifications has a decisive impact on the results of further development steps, requirements specifications often remain unclear and cause serious frictions as to the agreed scope of work especially in an offshore context then (Heindl & Biffl 2006; Vlaar, van Fenema & Tiwari 2008).

Before deciding whether to offshore further development steps it is therefore valuable to assess the suitability of corresponding requirements specifications in advance. Despite the relevance of such an initial requirements evaluation for offshore development planning, the topic has rarely been discussed by now. Therefore, we present an exploratory case study in which we address two research questions:

How can requirements specifications systematically be evaluated for their suitability to offshore later development steps? Based on the evaluation results, which recommendations for the offshoring approach can be lined out?

The presented case study reports on a large-scale industry project and has several distinctive features which provide novel insights. First, we applied a rational, theory-based method to evaluate offshoring suitability in a comprehensible way. The method takes into account multiple quality dimensions for rating specifications. Compensation factors are provided by the method to suggest options mitigating negative consequences of specification deficiencies in later offshore development steps. These compensation factors can be seen as critical success factors when offshoring development projects with suboptimal requirements specifications. Second, we analyzed the suitability of requirements specifications without any further stakeholder elicitation. In many large projects, shortcomings in finalized requirements specifications cannot considerably be corrected anymore, since this would bear intolerable repercussions. In our case, the availability of stakeholders for specification tasks was restricted from management, and their willingness was limited to re-iterate requirements documents that had already been discussed. Third, the results described in this paper directly supported management decisions on the project's offshoring approach, and we were able to validate our evaluation results and recommendations by monitoring and reviewing the actual further course and outcome of the project.

The rest of our paper is organized as follows. Chapter 2 describes the background and related work to further motivate the research gap and relate our approach to others. Chapter 3 presents the theoretical framework on which our study was built. Chapter 4 describes the case study in detail, and chapter 5 discusses key results. Chapter 6 concludes the paper.

2 Background and related work

Distributed processes with external development steps are common in the development of IS since long. With the growing global availability of highly qualified IS resources (Aspray, Mayadas & Vardi 2006), the offshoring of application development steps has become a widely considered option in sourcing strategies (Kehal & Singh 2006). Development processes typically distinguish conception, analysis, design, implementation, and acceptance as major development phases. They can (and usually do) include iterative elements and various quality assurance measures (Sommerville 2007). Table III-B4-1 summarizes major subsequent phases in distributed application development models, together with starting points, core activities and deliverables. Similar process patterns are used e.g. by Cusick and Prasad (2006) or King and Torkzadeh (2008) to discuss offshoring approaches. Awarding options are included in the last line of Table III-B4-1. In our case, we studied the offshore awarding for the design and the implementation phases.

But misconceptions are common between onshore teams that are in charge of requirements analysis and offshore teams that are in charge of later development steps. Vlaar, van Fenema and Tiwari (2008) state that "offshore team members could only develop literal understanding of the requirements", and explain how especially "knowledge and experience asymmetries" as well as "complex, novel and instable tasks and requirements" provide difficulties in offshoring. They underline that an offshoring context induces additional challenges to an unambiguous understanding during the already complex requirements analysis and specification. Consequently, when analyzing offshoring costs, Overby (2003) identifies "the ability to write clear specifications" as a key issue in the external assignment of application development functions. A lack of quality in requirements specifications (inconsistent, missing, incorrect) is recognized as one of the most substantial risk factors in global development (Sakthivel 2007). Hofmann and Lehner (2001) present a case study with 76 stakeholders from 15 requirements engineering teams and show that deficient requirements specifications are the single most important reason for the failure of development projects in their study.

Table III-B4-1: An overview of major process phases in the distributed development of individually specified application systems. Offshore awarding of design and implementation are a subject of this case study.

Phase	**Conception**	**Analysis**	**Design**	**Implementation**	**Acceptance**
Starting Point	business goals	scope, feasible targets	requirements	design	system ready for acceptance
Core Activity	high level requirements analysis	detailed requirements analysis	design (and architecture)	programming (and integration)	testing
Core Deliverable	requirements framework, feasibility study	detailed requirements specifications	design specifications	final application system	defects
Awarding	onshore	onshore	on-/offshore	on-/offshore	onshore

Many case studies investigate further into the role of requirements in distributed application development in general, often in situations of noticeable complexity, and some also address options to compensate deficient requirements specifications. Leonardi and Bailey (2008) study an offshoring case and find that engineers have great difficulties in interpreting the implicit knowledge embodied in development artifacts that were created in a different cultural context. They identify five compensating work practices (defining requirements, monitoring progress, fixing returns, strategically routing tasks, and filtering quality) that help to manage offshoring arrangements. Chatzoglou (1997), who investigates 107 different project cases from 74 different organizations, demonstrates how additional factors (such as relationships between requirements stakeholders and developers) and further project characteristics at the requirements capture stage (internal or external client; well defined or poorly defined

problem domain; bespoke or generic IS scope) are given insufficient attention although they directly influence the project. Heeks et al. (2001) further underline the importance of compensating factors. They mention a congruent client-contractor relationship as well as the influence of tacit knowledge, informal information, and culture for successful offshoring. They state: "Although necessary, these [clear, formalized requirements specifications] were not sufficient, because they incorporated a whole set of tacit assumptions and understandings that were not transferred about the nature of the customer, design and programming choices, and working practices."

Nevertheless, we could not find much related work which discusses methodical approaches to evaluate requirements specifications with respect to their offshore development suitability. Among the few methods, which were proposed to support the evaluation of specification documents in a wider sense, are the UML (Unified Modeling Language) quality metrics from Berenbach and Borotto (2006), but they only concern the formal correctness of requirements specifications and neglect the (in-)adequacy of the specified contents. These metrics hence fail short of providing broad statements on quality of specifications and their suitability in an offshoring context. Krogstie (1998) suggests an integrated framework for quality in conceptual models, and addresses many quality dimensions that provide sophisticated insights. But this high level framework remains too abstract for the actual application. An offshoring-specific approach is the transaction-cost-based framework proposed by Dibbern, Winkler and Heinzl (2008) which suggests an explanation for extra costs in offshoring and mentions requirements specifications and design as two of only five explicative factors. But this explanatory model does not support the actual evaluation of requirements specifications or design documents either.

Against this background, we seek to provide a more structured evaluation of the contextual offshore suitability of requirements specifications. While our conception of the related method was based on theoretical work so far, we analyzed its practical applicability as part of the presented case study in this paper.

3 Theoretical framework

3.1 Research method

To investigate into our research questions, we used an exploratory case study (Benbasat, Goldstein & Mead 1987; Yin 2003). Traditionally, case study research aims at the acquisition of supporting experience that influences theoretical constructs derived from previous theory, existing literature and common sense – not as an empirical test but as

an earlier, important step on the research path (Eisenhardt 1989). Building further on our theoretical research, we explored a practical case aiming at (i) a richer understanding of the requirements evaluation process in projects with later offshoring development steps, (ii) the application of our theory-based method in order to evaluate requirements specifications and to tailor its application to practical needs, and (iii) obtaining qualitative and quantitative research data during the application of that method that allows us to assess and to further refine its design on the basis of gathered experiences.

A complex single case design (Eisenhardt 1989) was chosen because we could deeply immerse into the large, real life setting on site and acquire a rich understanding of the client perspectives that drove the evaluation of requirements specifications during the advanced planning of offshore development steps. We could strengthen our theoretical and methodical insights with quantitative data from the application of the method. We obtained qualitative data from regular client interviews about relevant quality dimensions and compensation factors, adjustments of our method to the project context, evaluation outcomes, and our recommendations for the offshoring arrangements. To assess the validity of the data during our study, we observed the further course of the project and analyzed the actual outcomes against our results and predictions. At the end of the project we also interviewed the client and investigated the level of agreement with the initial evaluation results. As we studied a single case only, we will account for the external validity of our results at the end of the paper, where we discuss how far the setting variables and conclusions can be repeated and generalized (Lee 1989).

3.2 Evaluation method

For a rational and comprehensible operational assessment, our evaluation method builds upon the *cost-utility analysis* (CUA) technique. CUA was proposed by Zangemeister (1976) for the multi-dimensional review and selection of project alternatives, and is an established decision-making approach in multi-dimensional settings today (Keeney, Raiffa & Meyer 2003).

Our evaluation method utilizes CUA with eight fundamental *quality dimensions* to analyze if a requirements specification is suitable to support the offshoring of later development steps. The dimensions were determined from literature, e.g. Brown (2000), Davis (1993), Hall (1990), Liskov and Berzins (1986), or Schütte and Rotthowe (1998):

- *Adequacy*. Requirements shall be specified with adequate efforts and shall be documented at least to a precision that further development is possible without additional elicitation or interpretation.
- *Completeness*. All requirements shall be completely described (completeness is understood in relation to actually required features).
- *Comprehensibility*. Requirements shall be formally defined and at the same time easily readable for humans (e.g. through the use of a formal graphical notation and additional prosaic explanations).
- *Consistency*. All relations and dependencies between the requirements shall be explicitly defined.
- *Feasibility*. Notations and methods shall be used which are well-known and established in practice.
- *Flexibility*. Requirements shall be documented in a uniform and modular structure.
- *Neutrality*. Requirements shall be described independently from methods and technologies of further development (such as design paradigms or programming languages).
- *Standardization*. Requirements shall be specified according to predefined specification standards.

To generate an overall quality score, a requirements specification is rated according to each of the dimensions. With a uniform rating scale and predefined weights for each of the dimensions, applying CUA then determines the overall score through aggregation.

Complementing the offshoring suitability determination, the evaluation method supports the analysis of *compensation factors* to possibly mitigate deficiencies detected with one or more of the quality dimensions. The general effectiveness of the five compensation factors which are supported for the assessment was indicated in literature, e.g. by Leonardi and Bailey (2008), Remus and Wiener (2009), or Wada, Nakahigashi and Tsuji (2007):

- *Communication, language, and culture*. How easy or difficult is the daily communication between offshore partners?
- *Contracting*. How do the legal agreements between the offshore partners deal with inexistent, irreproducible, unclear or otherwise deficient requirements in the course of the development?

- *Domain knowledge*. How much tacit general knowledge and previous experience with the IS domain under development does the offshore partner contribute?
- *Learning relationship*. How mutually familiar are the offshore partners with the company, the processes, the people and the peculiarities on the respective other side?
- *Reliability*. What business model and strategic interest is pursued by each offshore partner, and how qualified and motivated are the operational teams?

These compensation factors are analyzed for their potential to provide counterbalancing means that can be managed against the quality dimensions in the actual context. From the general experience that compensation shows stronger effects against lower quality, compensation focus is on the quality dimensions rated lowest.

4 Case study

4.1 Basic assumptions

We found our two basic assumptions for an offshore project context shared by the client for the project. First, requirements are starting point and fundamental driver for subsequent steps in the IS development process. They describe the functionality that an application has to provide in its context. Requirements specifications document this by precisely defining the observable application behavior "from the outside" in black box style. They declare *what* an IS does, without any further suggestion about *how* it is achieved (Liskov & Berzins 1986).

Second, a decreasing quality of requirements specifications results in a wider interpretation range as to the actual meaning of the requested properties. This interpretation range gives room to misconceptions, in particular in offshoring situations with their need to be explicit. Thus it directly generates design and implementation risks for further development steps. To still deliver the expected IS solution in the required scope, timeframe, and quality, such misconceptions have to be rectified in the further course of a development project (Vlaar, van Fenema & Tiwari 2008). The more the quality of requirements specifications declines, the more it is difficult and unlikely that an appropriate compensation can still be achieved. Finally, scope and quality of the implemented IS solution suffer (Sommerville 2007). Hence it becomes vital to identify managerial countermeasures from a range of critical compensation factors that can (partly) balance out specification deficits during further offshore development.

4.2 Project outline

Our case study was hosted within a large, complex IS development project. The specific project provided to us a complex global scenario with extensive requirements specifications as evaluation basis. We were able to accompany the entire project in order to review our evaluation results and predictions later. The client, a global industry leader in the automotive business, developed a complex, highly individual and mission critical application system. This system was the corporation's designated global core application for sales support and sales automation in one of the business lines. The development project had top-level management attention and included the option to offshore major parts of the design and implementation work.

At the time of the case study, requirements specifications were already being developed in a joint effort by internal client teams and an external onshore contractor. In the project with its globally distributed work setting, requirements were to be communicated across organizational, national and cultural borders. The relevant requirements were extensive and comprised, among other artifacts, nearly 700 use cases only. Several thousand full time equivalent head count days were allocated merely for the requirements specification work within the client's internal teams. The case study commenced on requirements documents that were considered to be stable, final versions. It was not intended to change or clarify them much further. Within the case study, we interacted with the client management level only and we were asked not to involve stakeholders from the client departments who represented the requirements content-wise. In this sense, our study was conducted independently.

4.3 Realization of the case study

Sharing the basic assumptions, we discussed the assessment procedure with the client management in detail at the initial stage of the study. The proposed quality dimensions and the compensation factors were reviewed and agreed upon by the client. The CUA technique was lined out to the client and approved. Further operational details such as non-disclosure, communication channels, schedules and venues of regular meetings, etc. were settled in the initial phase, too.

The requirements specification then was assessed against each single quality dimension individually, through qualitative judgments from an evaluation group made up of the authors and external consultants. The group was supported by a number of assistants who took care of organizational tasks (mainly document management). The in-depth analysis of the documents took slightly more than two months, and the predefined set of

quality dimensions ensured the broadness of the review. The evaluation procedure itself was three-tiered and followed the CUA approach.

In stage one, an investigation documented, examined and evaluated the cross references between all specification parts for inconsistencies, contradictions, gaps, redundancies, missing parts, and missing or wrong identifiers.

To keep the inspection efforts manageable in the following two stages, we had to reduce the full specification by Pareto analysis ("ABC-analysis") to 20 percent "A" items that account for 80 percent of total effects (Juran & Gryna 1970; Reed 2001). The cross references from stage one were reused for this purpose. The "A" items identified on that basis included, next to other things, 22 percent core use cases.

In stage two, these "A" items were verified against the client's internal formal directives and, as far as applicable, against external formal directives such as those of the UML. In stage three, the "A" specification parts were validated with regard to their implementation by determining the missing details that prevent an interpretation-free system design. In this last step, the assessment also included the granularity of the requirements. It was supported next to other things by a design evaluation template created from internal and external design rules.

Table III-B4-2: Rating scale to evaluate quality dimensions of requirements specifications. A higher value indicates a higher quality of the examined requirements specification documents.

Rating	**1**	**2**	**3**	**4**
Label	deficient	below avg.	above avg.	good
Range (aggregated)	[1,00…1,75 [	[1,75…2,50 [	[2,50…3,25 [	[3,25…4,00]

The group discussed all findings in several team sessions with each other and came to a joint opinion on each of the eight quality dimensions. The rankings for each dimension were done according to the agreed scale (Table III-B4-2). The eight rated dimensions were finally aggregated as an equally weighed sum which denoted the overall requirements specification quality. For the quality dimensions that were evaluated as deficient, compensation options in the given project situation were assessed by the group and the client management in four subsequent sessions. For each deficient quality dimension, the five compensation factors were extensively discussed if they can accomplish any balancing contribution against the deficit in the contextual setting.

5 Key findings and discussion

5.1 Suitability of the requirements specification

The assessment of the eight quality dimensions against the requirements specification documents produced the following evaluation results (in ascending order of assessed quality):

- Adequacy: 1. The majority of the requirements were specified imprecisely, so it was not possible to generate an application design without further elicitation or interpretation.
- Consistency: 1. No higher-ranking structure of the requirements documents existed to explain the relations of the various parts, so the overall requirements structure remained unclear.
- Flexibility: 1. Several requirements had strong mutual dependencies, but the dependencies were not sufficiently specified in an explicit manner.
- Completeness: 2. Some requirements were incompletely specified or even placeholders only.
- Comprehensibility: 2. The specifications did not convey a satisfactory understanding of the required business functionality without further interpretation or background knowledge.
- Feasibility: 3. Notations used in the specification documents were established in practice and feasible. Some of them were used exclusively at the client though.
- Standardization: 3. There were only few breaches of specification standards and guidelines.
- Neutrality: 4. The requirements were specified independently of technologies and methods of further development.

The assessment of the requirements specification documents revealed serious deficiencies. The aggregated quality value was 2.125, in the “below average” range of the evaluation scale (Table III-B4-2). The specification was classified as “deficient” in three out of eight quality dimensions. Clearly, it was not possible to create a full systems design from the requirements documents without either broad interpretation from the designers, or considerable further elicitation between the designers and the requirements stakeholders. These findings produced approval and indicated caution on the client side, but also some neglecting from the external onshore contractor that supported the requirements analysis.

5.2 Scope of options from critical compensation factors

Facing a suboptimal requirements specification, the three quality dimensions with the lowest evaluation results were discussed between the evaluation group and the client management against the critical compensation factors. Figure III-B4-1 illustrates the results, i.e. the contextual scope of options that could be derived from the critical compensation factors as concrete countermeasures that can be operated against the negative specification quality observations and findings in the project:

- Communication, language, and culture. The clarification of dependencies between different specification parts requires good communication. A common language (such as English) and frictionless communication are required to deal with deficient requirements descriptions in the first place. Compatible conflict cultures are required to advance problem-solving in controversies arising from requirements issues.
- Contracting. Appropriate contractual offshoring arrangements are a basic precondition to be prepared for dealing with imprecise or instable requirements in the later course of the project.
- Domain knowledge. Offshoring contractors with a generally high knowledge level of the domain under development are needed to render unclear specification parts more precisely with less risk and to better and faster narrow the interpretation space.
- Learning relationship. Earlier experience in the collaboration between the offshoring partners helps overcoming problems with poorly aligned requirements specifications parts. The proper interpretation of requirements becomes simpler along a joint learning curve from the past.
- Reliability. High mutual confidence and strategic trust between offshoring partners is essential to ease up modifications to deficient parts of requirements specifications even if they have far reaching consequences.

From the joint analysis of compensation factors against identified requirements specification weaknesses in the actual situation, the offshore development suitability was seen as critical altogether.

For the selection of and the arrangements with offshore partners, the study results recommended to account for the identified compensation options, i.e. to manage towards a high degree of compensation when proceeding any further into offshoring development. The client management agreed to the options and their counterbalancing potential to mitigate offshoring complications that could be expected from the low specification quality.

		compensation factors				
		Communication, language, and culture	Contracting	Domain knowledge	Learning relationship	Reliability
low quality criteria	Adequate		↵	↵		
	Consistent	↵	↵	↵	↵	↵
	Flexible		↵	↵	↵	↵

Figure III-B4-1: Illustration of the possible scope of contextual counteractions from critical compensation factors. The ↵ symbol points out the direction of the compensation effect.

It was also explained that this potential can be realized only with offshoring partners that comply with the majority of the compensation factors. This was considered to be difficult in the project, in particular with respect to the implicit domain knowledge about a highly specific IS, to cultural aspects and to the learning relationship.

5.3 Reception and further course of the development project

The evaluation results were used by the client management as decision support for planning the further offshoring approach. Since the client considered it difficult to realize the compensation options, the most visible client reaction was to limit the offshore portion in the project to a maximum of 40 percent in design and implementation, from the initial idea of a complete offshoring.

Towards the end of the project, we could revisit and review the actual offshore development progress against our evaluation results. The client characterized the offshore development portion as problematic then. Around 25 percent of the features developed offshore had to be re-developed completely. Further 25-50 percent had to be re-developed in parts. Overall, less than half of the offshore development remained without rectification. Main reason mentioned was the offshore contractor's missing familiarity with the individual peculiarities on the client side. Compensation options could not fully be realized in the actual context, so it was hard for the client and the contractor to balance out specification deficiencies. Furthermore, contractual arrangements caused debates about the assignment of responsibilities for identified issues.

Consequently, the offshore quota in the correction of defects and in change requests was reduced to zero in the later project. The entire offshore portion for the whole project finally was below 10 percent in design and implementation. The client observed that offshoring on the basis of suboptimal requirements specifications worked better with largely standardized and generally known features and processes, but worse for individual and highly specific features which constituted the majority of requirements in

the project. The client again and assertively appraised the evaluation results – low specification quality and necessary high level of compensation – as being "altogether correct" in a retrospective review from the client perspective.

Finally it is striking that, despite of these seemingly dire figures, the client still reported an unspecific offshoring advantage of 10-15 percent at the bottom line. Does this indicate the benefits of offshoring if suboptimal specifications and the related countermeasures are specifically known and receive management attention early in the project? Does it hint at offshoring advantages from a rational and well-informed planning as supported by our proposed method? These could be interesting future research questions.

5.4 Methodological implications

The case study has two positive methodological research implications. First, the course of the case study demonstrated the smooth applicability of the proposed method in a large industrial setting. This includes the confirmation that the method does not depend on unrealistic assumptions, such as the existence of unavailable inputs or other practical impossibilities. Using the method, quality properties and compensation options could be determined without significant problems. The quality assessment could be performed independently from stakeholders on the basis of the requirements documents only. Managerial compensation options could be derived with the client management from the quality rating and the in-depth analysis of the actual project situation and its context.

Second, the results that were produced by the method were confirmed in three ways. The recommendations were included straight away into the decision making of the client management. For the assessment results, we received an explicit confirmation through subjective judgments from the client management in a retrospective review interview. The actual course of the development project, in particular the offshoring issues encountered, implicitly confirmed the initial assessment and recommendations.

6 Conclusions

Given the initial research questions, we could show that the applied method is capable to systematically evaluate requirements specifications for their offshoring suitability, and that it can also line out context-specific recommendations. The presented study was hosted in a single, but prolific industry case. The smooth applicability of the proposed suitability assessment method was confirmed in this context, as well as the validity of the findings. The applied method therefore appears to be adequate for practice. From an

academic perspective, the case study applied a theoretically derived method and thus contributed an inevitable step towards closing a research gap in systematic decision support approaches for the planning of offshore application development based on requirements specifications.

To what extent can the results from our single case study be repeated and generalized (Lee 1989)? Repeating the study in other projects seems trivial since the method itself works straightforward and has been documented in this paper. Its application is meaningful as long as the basic assumptions (described in section 4.1) are met. Generalizing the study results is difficult since a single case cannot constitute enough empirical evidence. But we may consider the facts that the study was performed in a large and relevant project, that all involved parties were experienced players in globally distributed application development, that the examined requirements had an ample scope, and that there was a globally distributed setting from the beginning. Therefore, we deem the issues we dealt with and many of the results we achieved as being typical. While not yet proven to be generally valid, the method that we applied was clearly validated in a large industry setting. It hence seems to be relevant at the least, and its application is repeatable.

The presented case study is one step towards constructing a confirmed, systematic decision support method for offshoring approaches based on the suitability of underlying requirements specifications in combination with critical compensation factors. Further research is already on its way and aims at formalizing the evaluation method and applying and refining the approach in other cases.

References (B4)

Aspray, W.; Mayadas, F.; Vardi, M. (eds.) (2006), "Globalization and Offshoring of Software: A Report of the ACM Job Migration Task Force: The Executive Summary, Findings, and Overview of a Comprehensive ACM Report on the Offshoring of Software Worldwide", ACM, New York, USA.

Benbasat, I.; Goldstein, D.; Mead, M. (1987), "The Case Research Strategy in Studies of Information Systems", *MIS Quarterly*, 11 (3): 369-386.

Berenbach, B.; Borotto, G. (2006), "Metrics for Model Driven Requirements Development", *Proceedings of the 28th International Conference on Software Engineering*, ACM, 20-28 May 2006, Shanghai, China: 445-451.

Brown, A. (2000), *Large-Scale, Component-Based Development*, Prentice Hall, Upper Saddle River, USA.

Chatzoglou, P. (1997), "Factors Affecting Completion of the Requirements Capture Stage of Projects with Different Characteristics", *Information and Software Technology*, 39 (9): 627-640.

Corriveau, J. (2007), "Testable Requirements for Offshore Outsourcing", *Proceedings of the First International Conference on Software Engineering Approaches for Offshore and Outsourced Development*, Lecture Notes in Computer Science 4716, Springer, 5-7 February 2007, Zurich, Switzerland: 27-43.

Cusick, J.; Prasad, A. (2006), "A Practical Management and Engineering Approach to Offshore Collaboration", *IEEE Software*, 23 (5): 20-29.

Davis, A. (1993), *Software Requirements: Objects, Functions, and States*, Prentice Hall, Englewood Cliffs, USA.

Dibbern, J.; Goles, T.; Hirschheim, R.; Jayatilaka, B. (2004), "Information Systems Outsourcing: A Survey and Analysis of the Literature", *The DATA BASE for Advances in Infomation Systems*, 35 (4): 6-102.

Dibbern, J.; Winkler, J.; Heinzl, A. (2008), "Explaining Variations in Client Extra Costs Between Software Projects Offshored to India", *MIS Quarterly*, 32 (2): 333-366.

Eisenhardt, K. (1989), "Building Theories from Case Study Research", *Academy of Management Review*, 14 (4): 532-550.

Gefen, D.; Wyss, S.; Lichtenstein, Y. (2008), "Business Familiarity as Risk Mitigation in Software Development Outsourcing Contracts", *MIS Quarterly*, 32 (3): 531-551.

Hall, A. (1990), "Seven Myths of Formal Methods", *IEEE Software*, 7 (5): 11-19.

Heeks, R.; Krishna, S.; Nicholson, B.; Sahay, S. (2001), "Synching or Sinking: Global Software Outsourcing Relationships", *IEEE Software*, 18 (2): 54-60.

Heindl, M.; Biffl, S. (2006), "Risk Management with Enhanced Tracing of Requirements Rationale in Highly Distributed Projects", *Proceedings of the 2006 International Workshop on Global Software Development for the Practitioner*, ACM, 20-28 May 2006, Shanghai, China: 20-26.

Hofmann, H.; Lehner, F. (2001), "Requirements Engineering as a Success Factor in Software Projects", *IEEE Software*, 18 (4): 58-66.

Hofstede, G. (2002), *Culture's Consequences: Comparing Values, Behaviors, Institutions, and Organizations Across Nations*, 2nd edition, Sage, Thousand Oaks, USA.

Juran, J.; Gryna, F. (1970), *Quality Planning and Analysis: From Product Development Through Usage*, McGraw Hill, New York, USA.

Keeney, R.; Raiffa, H.; Meyer, R. (2003), *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*, Cambridge University Press, Cambridge, UK.

Kehal, H.; Singh, V. (eds.) (2006), *Outsourcing and Offshoring in the 21st Century: A Socio-Economic Perspective*, Idea Group, Hershey, USA.

King, W.; Torkzadeh, G. (2008), "Information Systems Offshoring: Research Status and Issues", *MIS Quarterly*, 32 (2): 205-225.

Kobitzsch, W.; Rombach, D.; Feldmann, R. (2001), "Outsourcing in India", *IEEE Software*, 18 (2): 78-86.

Krogstie, J. (1998), "Integrating the Understanding of Quality in Requirements Specifications and Conceptual Modeling", *ACM SIGSOFT Software Engineering Notes*, 23 (1): 86-91.

Lacity, M.; Willcocks, L. (2003), "IT Sourcing Reflections: Lessons for Customers and Suppliers", *Wirtschaftsinformatik*, 45 (2): 115-125.

Lee, A. (1989), "A Scientific Methodology for MIS Case Studies", *MIS Quarterly*, 13 (1): 33-52.

Leonardi, P.; Bailey, D. (2008), "Transformational Technologies and the Creation of New Work Practices: Making Implicit Knowledge Explicit in Task-Based Offshoring", *MIS Quarterly*, 32 (2): 411-436.

Liskov, B.; Berzins, V. (1986), "An Appraisal of Program Specifications", in Gehani, N.; McGettrick, A. (eds.), *Software Specification Techniques*, Addison Wesley, Wokingham, UK: 3-24.

Overby, S. (2003), "Offshore Outsourcing The Money: Moving Jobs Overseas Can Be a Much More Expensive Proposition Than You May Think", *CIO*, 16 (22): 60-66.

Pryor, B.; Keane, B. (2004), "Critical Success Factors in Outsourcing", *A CFO Magazine Symposium: Offshore Outsourcing – Risks and Rewards*, CFO Publishing, 17 June 2004, New York, USA: 11-13.

Reed, W. (2001), "The Pareto, Zipf and Other Power Laws", *Economic Letters*, 74 (1): 15-19.

Remus, U.; Wiener, M. (2009), "Critical Success Factors for Managing Offshore Software Development Projects", *Journal of Global Information Technology Management*, 12 (1): 6-29.

Sakthivel, S. (2007), "Managing Risk in Offshore Systems Development", *Communications of the ACM*, 50 (4): 69-75.

Schütte, R.; Rotthowe, T. (1998), "The Guidelines of Modeling – An Approach to Enhance the Quality in Information Models", *Proceedings of the 17th International Conference on Conceptual Modeling*, Lecture Notes in Computer Science 1507, Springer, 16-19 November 1998, Singapore, Singapore: 240-254.

Sommerville, I. (2007), *Software Engineering*, 8th edition, Pearson Education, Harlow, UK.

Vlaar, P.; van Fenema, P.; Tiwari, V. (2008), "Cocreating Understanding and Value in Distributed Work: How Members of Onsite and Offshore Vendor Teams Give, Make, Demand and Break Sense", *MIS Quarterly*, 32 (2): 227-255.

Wada, Y.; Nakahigashi, D.; Tsuji, H. (2007), "An Evaluation Method for Offshore Software Development by Structural Equation Modeling", *Proceedings of the First International Conference on Software Engineering Approaches for Offshore and Outsourced Development*, Lecture Notes in Computer Science 4716, Springer, 5-7 February 2007, Zurich, Switzerland: 114-127.

Winkler, J.; Dibbern, J.; Heinzl, A. (2008), "The Impact of Cultural Differences in Offshore Outsourcing – Case Study Results from German-Indian Application Development Projects", *Information Systems Frontiers*, 10 (2): 243-258.

Yin, R. (2003), *Case Study Research: Design and Methods*, 3rd edition, Sage, Thousand Oaks, USA.

Zangemeister, C. (1976), *Nutzwertanalyse in der Systemtechnik – Eine Methodik zur multidimensionalen Bewertung und Auswahl von Projektalternativen*, 4th edition, Wittemann, Munich.

IV Selektion

Beitrag B5:	**Optimal stopping for the run-time self-adaptation of software systems**
Autoren:	Oliver Skroch, Klaus Turowski Lehrstuhl für Wirtschaftsinformatik und Systems Engineering Universität Augsburg, Universitätsstr. 16, D-86159 Augsburg [oliver.skroch|klaus.turowski]@wiwi.uni-augsburg.de
Veröffentlichung:	Journal of Information & Optimization Sciences (zur Veröffentlichung angenommen am 30. September 2009).

Hochentwickelte Softwaresysteme besitzen die Fähigkeit, sich selbst zur Laufzeit neu zu konfigurieren, wobei sie für die Ausführung bestimmter Funktionen zwischen alternativ möglichen Diensten wählen können. Diese möglichen Alternativen können bereits bei der Entwicklung in die Systeme eingebaut werden. Sie können aber auch erst später zur Laufzeit in offenen und unkontrollierbaren Netzen extern verfügbar werden. Ein aktuelles Anwendungsbeispiel sind Mashups und Web-Dienste im Internet. Der folgende Beitrag führt die Forschung von Skroch und Turowski (2007) fort und zeigt, wie die opportunistische Suche nach Möglichkeiten zur Selbst-Rekonfiguration von Softwaresystemen unter Einbeziehung von unkontrollierten, externen Optionen zur Laufzeit optimiert werden kann. Hierzu wird auf Ergebnissen aus der mathematischen Stopptheorie aufgebaut, die die bestmögliche, untere Wahrscheinlichkeitsgrenze für die Auswahl der optimalen Option garantieren. Der Beitrag präsentiert zwei Anwendungsszenarien und leitet jeweils effiziente Optimierungsalgorithmen ab. Die Theorie wird durch Simulation für beide Szenarien bestätigt, in der die Vorteile im Vergleich zu einem geschlossenen Softwaresystem gemessen werden.

1 Introduction

Flexible software systems are based on the concept of modularity. They can be constructed through component-based and service-oriented software engineering approaches. These approaches promote the reuse of software that has already been made available before. Ideally, a larger application can be build by identifying already existing, suitable components or services first, and then composing the parts into a loosely coupled, larger system. The resulting larger software system jointly performs all operations required, while mutual dependencies between its parts are fully explicit (Achermann & Nierstrasz 2005; Parnas 1972; Szyperski, Grunz & Murer 2002).

Selecting suitable components and services is one decisive step in composing such software systems. In component-based and service-oriented approaches, suitable components and services can be identified by searching through software repositories or electronic markets. It can be done at build-time when designing and implementing the application. This leads to a closed software architecture where all components and services are internal parts of the larger system. Closed systems can already provide internal run-time self-adaptation capabilities. One example are VoIP (Voice over Internet Protocol) clients which can select, from a number of codec options integrated at build-time, one suitable codec according to actual data rates measured at run-time.

Advanced software architectures can perform parts of their functionality also through external components or services that were not integrated into the software. Such external components or services could be unknown at build-time, but on open and uncontrolled software platforms they may become available in large numbers later at run-time. Prominent examples on the Internet are Web services in general (Atkinson et al. 2002) and service mashups in particular (Bernstein & Haas 2008; Gamble & Gamble 2008).

Exhaustive run-time search for better service options is impossible on huge open platforms such as the Internet. Anonymous, independent services from distributed open platforms are unknown and can not be controlled either. At first glance, searching under these conditions seemingly can be improved with heuristics only. Still we propose an exact algorithmic optimization for self-adaptation processes under these conditions: to determine the best moment when to stop a search for further service options. From stopping theory, we derive and simulate efficient algorithms that implement a search strategy with the best possible lower probability bound for choosing an optimal self-adaptation option.

The rest of the paper is structured as follows. Chapter 2 explains the required background and states assumptions made. Chapter 3 lines out two different run-time self-adaptation scenarios and presents the applicable stopping theory for both scenarios. Chapter 4 applies the theory to both scenarios, derives actual algorithms, and presents results from two extended simulation examples, where advantages over a corresponding static software system are measured. Chapter 5 summarizes and concludes the paper.

2 Flexible software architectures and matching schemes for self-adaptation

Figure IV-B5-1 illustrates in an example how services are supplied and requested through service interfaces in component software system architectures. The two decisive principles are composition and delegation. A requested service interface can be composed with a supplied service interface, to combine into aggregated components. Service interfaces can be delegated to other service interfaces of the same type, to hand over processing. Supplied service interfaces are available from other components or even from open platforms.

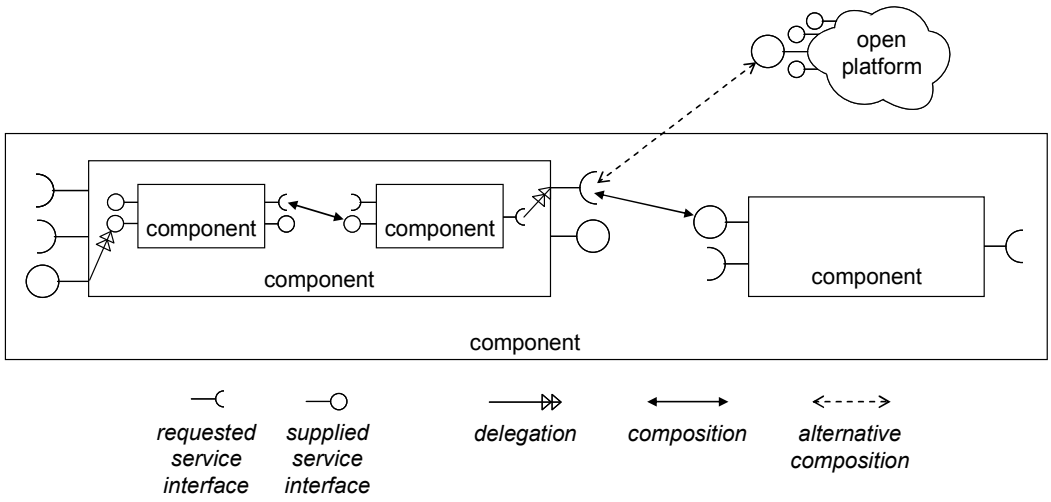

Figure IV-B5-1: Component software architecture example. Based on Shaw and Garlan (1996).

Composition implies to match supplied interfaces with requested interfaces. Figure IV-B5-2 suggests a classification of the possible schemes for matching supplied service options in existing compositions. The set of n supplied service options from external, uncontrolled platforms can be large. The set of m compositions from within the closed software can also be large, but $n >> m$.

- The choice scheme compares one composition – a requested service with its supplied service – against the alternative composition of this same requested service with one different supplied service option. It is the elementary decision whether the present

composition or the alternative composition is better. This scheme is also the basic consideration for the other schemes.

- The allocation scheme checks one particular service option for many or all compositions. Allocation can be seen as a repetition of choice, trying one supplied option in all compositions.
- The search scheme checks many supplied service options for one particular composition. Search can be seen as a repetition of choice, trying one composition with all supplied options.
- The screening scheme checks many or all compositions with many supplied service options each. Screening is the most general approach and can be seen as allocation with search.

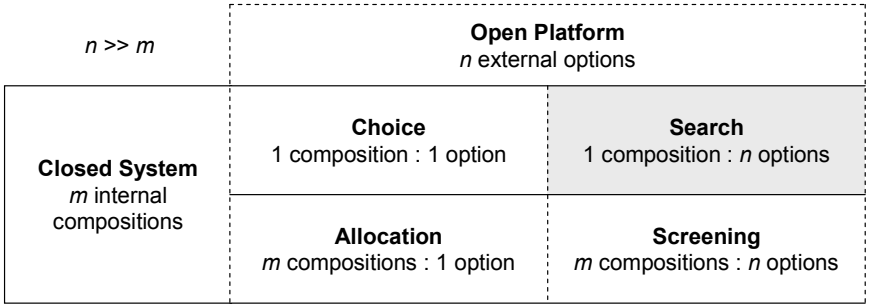

Figure IV-B5-2: Possible matching schemes.

Run-time software self-adaptation aims at dynamically re-composing services. Service options supplied at run-time via open platforms are assumed to fit function wise, but may differ in other ways such as quality, cost, etc. For a given composition it is possible to improve such non-functional system features by selecting a best supplied service option and by adapting the system accordingly (it can be assumed that the set of supplied options is never empty, if we consider the internal service supply as fallback if no external option is chosen).

Run-time software self-adaptation is triggered by the adapting system itself. Before a requested service interface calls the supplied interface at run-time, the system looks for externally supplied options and decides whether to re-compose this service call. This implies matching operations from the search scheme and excludes the allocation scheme. It can be assumed that the utility function is to choose the best available option, and that the actual values for the comparison can be determined by the system. The related computing can be done, for example, in an adaptation component that orchestrates and monitors the re-compositions.

Independent services from open software platforms cannot be controlled. One consequence is that any supplied service can be unavailable or changed in the next moment. This implies that the final decision whether to choose a certain service must be made straightforward. Consequently, for the search scheme it is not possible to memorize a supplied service option and, after further unsuccessful search, get back and use this service option.

Software self-adaptation can be performance critical. With n and m both large and $n >> m$, the screening scheme is not feasible for run-time matching operations (screening is relevant at build-time rather). But even the search scheme on open platforms already requires efficient calculation methods with a large number of available service options n.

3 Optimal stopping in two self-adaptation scenarios

We examine two simple scenarios that avoid exhaustive search and generally enable the application of run-time self-adaptation with uncontrolled external options in the described situation. Firstly, either one limits the number of options to be considered from the many available options. Or, secondly, one allows only a maximum run-time delay. Stopping theory can be used in both scenarios to optimize the run-time software self-adaptation process by determining the best point to stop the search for further alternative options.

Stopping problems are a well known research topic in mathematical statistics. A general solution approach to our problem class can be found already in (Lindley 1961) and within the framework of stopping Markov chains in (Dynkin & Juschkewitsch 1969). Common strategies for optimal stopping under considerations suitable for the two scenarios are described in (Bruss 1984) and (Bruss 2000).

3.1 Limited number of run-time options

The first scenario limits the number of alternative options n which can be considered for self-adaptation. This means that the self-adaptation process evaluates supplied service options at most up to this limited number, and the limit is predefined.

With a predefined, limited number of unknown and independent options, let $I_1, I_2, \ldots, I_n \in \{0;1\}$ be independent indicator functions defined on a probability space (Ω, A, P). An index k is called a success if $I_k = 1$. The indicators are observed in

sequence $I_1, I_2, \ldots$ It is possible to stop at any of them but it is not possible to recall any preceding.

Let T be the class of stopping rules τ so that $\{\tau = k\} \in \sigma(I_1, I_2, \ldots, I_k)$ which represents the sigma-algebra generated by the indicator sequence. The optimal stopping rule $\tau^* \in$ T maximizes the probability of the event $I_\tau = 1$ and $I_{\tau+1} = I_{\tau+2} = \ldots = I_n = 0$.

Now let $p_j = \mathrm{E}(I_j)$ be the probabilities for the independent indicators. Let $q_j := 1 - p_j$ and the so-called odds $r_j := p_j / q_j$. The optimal rule τ^* for stopping on the last success is to stop on the first index (if any) k with $I_k = 1$ and $k \geq s$ where

$$s = \sup\left\{1, \sup\left\{1 \leq k \leq n : \sum_{j=k}^{n} r_j \geq 1\right\}\right\} \quad \text{with } \sup\{\varnothing\} := -\infty \tag{IV-B5-1}$$

Rule (IV-B5-1) is intuitive. The optimal strategy is to add up the odds $r_n + r_{n-1} + \ldots$ ("backwards") until this sum becomes equal to or greater than one, at index s, and then to stop at the first index $k \geq s$ with a success. In other words, it is optimal to stop as soon as the expected number of future successes becomes equal to or less than one. Then, the value (probability for the best choice) is 1/e, given by

$$V(n) = \prod_{j=s}^{n} q_j \sum_{j=s}^{n} r_j = Q_s(n) R_s(n) \tag{IV-B5-2}$$

This is the odds theorem of optimal stopping, proven in (Bruss 2000).

3.2 Limited run-time delay

The second scenario defines a maximum length for the time frame that can be used for a self-adaptation call at run-time, while the number of supplied service options is not known or cannot reasonably be predefined (except that it is known that there are many options).

Then, with a distribution function $F(z)$ on the real time interval $[\,0; t_{\max}\,]$, let $Z_1, Z_2, \ldots$ be independent random variables (each with a continuous distribution function F) where Z_k is the arrival time of option k. Let N be a non-negative integer random variable independent of all Z_k so that N represents the unknown total number of supplied options. With $N = n$, each arrival order $\langle 1\rangle, \langle 2\rangle, \ldots, \langle n\rangle$ is equally likely. Since the best service option needs to be selected, it only makes sense to accept an option that is better than all previous ones, and all previous ones must have been evaluated.

The waiting time x is defined as the time up to which all options are evaluated without accepting, while the value of the leading option is remembered. The first leading option after time x is accepted, if there is one, and all options are refused, if there is none. This is called the x-strategy.

For any distribution with $P(N > 0) > 0$ there exists a waiting time x^* maximizing the success probability for the x-strategy. Moreover, for all $\varepsilon > 0$ there is an integer m where $n \geq m$ implies

$$x^* \in \left[\frac{1}{e_F} - \varepsilon; \frac{1}{e_F} \right] \text{ where } \frac{1}{e_F} = \inf\left\{ x \mid F(x) = \frac{1}{e} \right\} \qquad \text{(IV-B5-3)}$$

Rule (IV-B5-3) is the only waiting time policy with the asymptotically best possible success probability $\geq 1/e$, regardless of the distribution of N. This is the $1/e$ law of optimal stopping, proven in (Bruss 1984).

4 Application and simulation

Optimal stopping can be applied to optimize the run-time self-adaptation of software systems searching for options on open platforms. No literature was found describing this application, except for the authors' previous research (Skroch & Turowski 2007).

4.1 Limited number of run-time options

The odds theorem (Bruss 2000) can be applied to optimize the first scenario of run-time software self-adaptation. Let the best alternative option show up at j and let the stopping index be s. The best service option will therefore be selected only if $j > s$, and only if the "second best" service option before j appears at i with $i \leq s$, which happens in s out of j-1 cases. Each permutation of the trial sequence is equally likely. So the probability for the best service option at position j is $1/n$ and the probability for the second best service option among the first s is $s/(j-1)$. The probability P_n that the best service option is selected summarizes (over all $j = s+1, s+2, \ldots, n$) the probability for the best service option at position j times the probability for the "second best" service option among the first s:

$$P_s = \sum_{j=s+1}^{n} \left(\frac{1}{n} \frac{s}{j-1} \right) = \frac{s}{n} \sum_{j=s}^{n-1} \frac{1}{j} \qquad \text{(IV-B5-4)}$$

(IV-B5-4) yields $R_s = 1/(n-1) + 1/(n-2) + \ldots$ stopped at $R_s = 1$. At the optimum, i.e. the stopping index s, as $n \to \infty$ it can be recognized that $s/n \to 1/e$ and also $V_s(n) \to 1/e$. The value $1/e$ (~0.368) is a typical lower bound well-known from the classical best choice problem. See also Lindley (1961), Bruss (2000).

The implemented algorithm therefore matches service options with the internal composition and rejects all options, while memorizing the value of the best option yet. After a proportion of n/e of the options has passed, the next leading option is chosen, if there is one. Otherwise no alternative choice is made and the original composition remains in place. Since n is predefined in this first scenario, n/e is a constant.

The algorithm is efficient. Additional time complexity is O(n) in the worst case, linear on the number of evaluated options, because for each evaluated option, one single comparison is made against the previously best option. In the best case a leading option appears immediately after n/e, adding constant complexity only. Additional space complexity is constant even in the worst case, because at any time only one value (the best yet) is stored.

For this first "limited options" scenario, an example was simulated with Web services offering currency exchange rates of different age. It can be assumed that more recent exchange rates are better.

Table IV-B5-1: Results from simulation experiments for the "limited options" scenario.

exp.	avg. quality	best was selected
1	879.3	0.29
2	879.3	0.33
3	894.5	0.38
4	906.3	0.29
5	867.6	0.32
avg.	***885.4***	***0.32***

In the simulation experiments, uniformly distributed quality values between 0 (worst) and 999 (best) were randomly assigned to the compositions with external service options. The internal composition was given an assumed fixed value of 700. These assumptions simplify the simulation without loss of generality as to our intended experimental demonstration. The proposed optimization method does not require any particular quality measurement function, except that it has to produce at least ordinal results for the matching operation.

Five experiments with 100 self-adaptations in each run were simulated. The limit n was set to 1000 service options. Table IV-B5-1 shows results from the first simulation.

With a predefined number of evaluated options, the run-time self-adaptation example with optimal stopping outperformed the assumed closed software system by 167.6 to 206.3 quality points. The average improvement was 185.4 points, or 26.5 percent, over all five "limited options" experiments together. The best available service was actually selected in 32 percent of the self-adaptation trials (where the theoretical prediction is 36.8 percent).

4.2 Limited run-time delay

The 1/e law (Bruss 1984) can be applied to optimize the second scenario of run-time software self-adaptation. With many suitable service options available on open platforms, a uniform distribution over time is assumed for service discovery.

Take $x = F(z),\ z \in [\,0; t_{\max}\,]$ with a continuous time scale x between 0 and 1 and with each $X_k = F(Z_k)$ uniform on $[\,0;1\,]$. A stopped search ends optimal if the best service option $\langle 1 \rangle$ arrives in $]\,x;1\,]$ before all other service options arriving in $]\,x;1\,]$ which are better than the best of those which arrived in $]\,0;x\,]$. From the k+1 best options the option $\langle k+1 \rangle$ arrives in $]\,0;x\,]$ and the k best ones in $]\,x;1\,]$ with probability $x(1\text{-}x)^k$. Since $\langle 1 \rangle$ arrives before $\langle 2 \rangle, \ldots, \langle k \rangle$ with probability 1/k one obtains the success probability:

$$P_n(x) \;=\; x \sum_{k=1}^{n} \frac{1}{k}(1-x)^k \;=\; x \sum_{k=1}^{n-1} \frac{1}{k}(1-x)^k + \frac{1}{n}(1-x)^n \quad \text{with } n \geq 2 \qquad \text{(IV-B5-5)}$$

The sum term of (IV-B5-5) contains the Taylor expansion of –ln(x). As $n \to \infty$ one obtains $P_n(x) \to -x \ln(x)$ which has a unique maximum at $x = 1/e$. The value 1/e (~0.368) is the (asymptotically) best possible lower bound. See also Lindley (1961), Bruss (1984).

The respective algorithm therefore matches service options with the internal composition and rejects all options, while memorizing the value of the best option yet. With a uniform distribution of service discovery events, as soon as a proportion of t/e of the predefined time frame has passed the next leading service option is chosen, if there is one. Otherwise no alternative choice is made and the original composition remains in place. Since t is predefined in this second scenario, t/e is a constant.

The algorithm is again efficient. Additional time complexity is constant even in the worst case, because the maximum length of the time frame *t* is preset. Additional space complexity is also constant even in the worst case, because at any time only one value (the best yet) is stored.

For this second "limited delay" scenario, simulation examples were conducted with settings according to the "limited options" case. In the "limited delay" simulation five experiments were conducted with 500 self-adaptations each, 1000 supplied service options were available for each self-adaptation, and the maximum run-time delay *t* was set to 200 milliseconds. Table IV-B5-2 shows results from the second simulation.

Table IV-B5-2: Results from simulation experiments for the "limited delay" scenario.

exp.	avg. quality	best was selected
1	874.0	0.376
2	877.2	0.364
3	870.3	0.354
4	856.7	0.294
5	872.0	0.350
avg.	*870.0*	*0.348*

On a predefined maximum delay of the matching operation, the run-time self-adaptation example with optimal stopping performed 156.7 to 177.2 quality points better than the assumed closed software system. The average improvement was 170.0 points, or 24.3 percent, over all five "limited delay" runs together. The best available service was actually selected in 34,8 percent of the service calls (where the theoretical prediction is 36.8 percent).

5 Summary and conclusion

Stopping theory has been used to optimize the run-time self-adaptation of advanced, dynamic software systems in two scenarios. One scenario predefined the maximum number of options at run-time. The other scenario predefined the maximum run-time delay. For both scenarios, suitable stopping theory was applied and efficient algorithms were derived. Simulation experiments showed that dynamic software systems with run-time self-adaptation and optimal stopping outperform a corresponding static software system.

A major driver for the applicability of the results is the increasing use of open platforms for distributed, service-oriented systems and mashups. Important application areas

already include grid computing, distributed multimedia, mobile computing, and self-healing software. With few changes, the results are applicable also for the run-time self-adaptation of software systems to dynamically changing requirements.

References (B5)

Achermann, F.; Nierstrasz, O. (2005), "A calculus for reasoning about software composition", *Theoretical Computer Science*, 331 (2-3): 367-396.

Atkinson, C.; Bunse, C.; Groß, H.; Kühne, T. (2002), „Towards a General Component Model for Web-Based Applications", *Annals of Software Engineering*, 13 (1): 35-69.

Bernstein, P.; Haas, L. (2008), "Information Integration in the Enterprise", *Communications of the ACM*, 51 (9): 72-79.

Bruss, T. (1984), "A unified approach to a class of best choice problems with an unknown number of options", *The Annals of Probability*, 12 (3): 882-889.

Bruss, T. (2000), "Sum the odds to one and stop", *The Annals of Probability*, 28 (3): 1384-1391.

Dynkin, E.; Juschkewitsch, A. (1969), *Sätze und Aufgaben über Markoffsche Prozesse*, Springer, Heidelberg.

Gamble, T.; Gamble, R. (2008), "Monoliths to Mashups: Increasing Opportunistic Assets", *IEEE Software*, 25 (6): 71-79.

Lindley, D. (1961), "Dynamic programming and decision theory", *Applied Statistics*, 10 (1): 39-51.

Parnas, D. (1972), "On the Criteria To Be Used in Decomposing Systems into Modules", *Communications of the ACM*, 15 (12): 1053-1058.

Shaw, M.; Garlan, D. (1996), *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, Upper Saddle River, USA.

Skroch, O.; Turowski, K. (2007), "Improving service selection in component-based architectures with optimal stopping", *Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications*, IEEE Computer Society, 28-31 August 2007, Lübeck: 39-46.

Szyperski, C.; Gruntz, D.; Murer, S. (2002), *Component Software: Beyond Object-Oriented Programming*, 2nd edition, Addison Wesley, London, UK.

Beitrag B6:	**Reducing domain level scenarios to test component-based software**
Autoren:	Oliver Skroch, Klaus Turowski Lehrstuhl für Wirtschaftsinformatik und Systems Engineering Universität Augsburg, Universitätsstr. 16, D-86159 Augsburg [oliver.skroch|klaus.turowski]@wiwi.uni-augsburg.de
Veröffentlichung:	Journal of Software, 2 (5): 64-73 (2007).

Durch die kompositorische Wiederverwendung von Softwarelösungen werden funktionale „higher-order“ Softwaretests (Myers 1979, S. 103ff) für unabhängige, fachliche Anforderungen von Endbenutzern immer wichtiger. Der folgende Beitrag basiert auf einer Methode, die bereits früher vorgeschlagen wurde (Skroch 2007). Mit ihr können testbare Szenarien durch Abstraktion, Reduktion und Inklusion direkt aus einem Modell der Anwendungsdomäne des Kunden abgeleitet werden. Die dabei entstehenden, verzweigungsfreien Szenarios können damit zu Referenzkriterien für spätere Tests erweitert werden. Angebotene Komponenten und Dienste – genauer, bereits ihre Spezifikationen – können dann einfach auf die Erfüllung dieser Szenarien geprüft werden. Der Beitrag stellt den Ansatz in einem Überblick vor und führt, im Rahmen der vorgeschlagenen Gesamtmethode, den Schritt der Domänenreduktion in seinen Einzelheiten aus. Ein auf der Anforderungsseite weitestgehend reales Beispiel, ein Ausschnitt aus den Kundenverwaltungsprozessen eines Großunternehmens, wird dabei vorgestellt und reduziert. Vorteile der Methode sind erstens ein ihr zugrunde liegendes, klares Geschäftsmodell, zweitens die Gewinnung von Testorakeln, die unabhängig von Software-Entwicklungsprozess sind, und drittens Prüfergebnisse, die früh im Software-Lebenszyklus verfügbar sein können, womöglich sogar bevor die Software selbst für Tests zur Verfügung steht.

1 Introduction

Software engineering is in the process of evolving from a craft to an industry and reuse is one decisive element that supports and propels this evolution. Reuse has even been described as "the only realistic approach" (Mili, Mili & Mili 1995, p. 529) to meet the needs of a software industry. Recently, further increasing needs for reuse have been listed among the top trends that will influence future software processes (Boehm 2005).

Compositional reuse is one of the fundamental software reuse technologies (Biggerstaff & Richter 1987). The approach is to reuse executable artifacts which are found in repositories, and compose them into larger applications (Szyperski, Gruntz & Murer 2002). Compositional reuse of black box business components is part of the overall concept of component-based business applications, where business components are described by multi-layered and semi-formal specifications, implement services from a business domain, and are envisaged to be traded on markets (Turowski 2003). Such compositional reuse includes

- building software for reuse, by creating self contained, marketable, fully described black box artifacts on the supply side,
- building software from reuse, by composing larger applications from these executable stand-alone artifacts on the demand side, and
- trading the associated software artifacts or components on a market (possibly an internal market within a corporation).

In this environment, the demand side – customers and end users of component software – looks for useful software components and does not want to access source code but restricts to a black box view. The demand side focus therefore is on "higher-order" (Myers 1979, pp. 103ff) compliance of domain level pragmatics and semantics, while mere formal and syntactical compliance is often perceived as technical precondition in the responsibility of the supply side.

Software component reuse with parts that can be looked up in catalogues and can then be integrated into large applications similar to electronic parts has been proposed already since long (McIlroy 1968). But non-trivial problems still complicate broad compositional software reuse in theory and in practice today. Among the problems on the demand side is the evaluation of available components against their more complex end user domain requirements: assuming that an offered component complies syntactically, it still needs to be tested if its pragmatics and semantics are useful for a specific domain automation purpose.

In traditional engineering disciplines, the importance of testing is well acknowledged because of a long history of experiences. In software engineering it is on the one hand known that software is fundamentally less reliable than traditional engineering products (Parnas 2001) and that building software will probably always be hard (Brooks 1987). On the other hand the well-known notion of "good-enough software" (Yourdon 1995, p. 79) shows that we have to deal with a pragmatic view on quality aspects of software, in particular with large enterprise applications.

But also good enough software development can profit from testing, especially with enterprise-sized systems if errors are found efficiently and early in the development process. Firstly, it was shown that the effort for error correction grows markedly when the error is detected later. Secondly, the earlier errors are detected the more rectification alternatives are available. Thirdly, studies in science and projects in industry indicate that testing takes more than fifty percent of the effort even with non-safety critical software.

Software testing is even more important whenever prefabricated items such as components are reused. Firstly, a single component made for reuse must be more thoroughly tested than a component made to be used once because it is reused in combinations unknown at the time of development. Secondly, a system based on a configuration of multiple black box components from different suppliers must be more thoroughly tested as compared to large pre-integrated products. (Meyer 2003)

The distinction between technology based supplier testing and domain based customer testing is widely acknowledged, in particular with component-based software (Weyuker 1998; Gao, Tsao & Wu 2003). Recently, an approach was proposed specifically for component validation testing on the domain level of the demand side (Skroch 2007). It is based on testable scenarios which are independently derived from an end user domain and become checked against reuse specifications from suppliers.

The rest of the paper presents and elaborates the method and is structured as follows. Section two of the paper sets out basic assumptions and presents the underlying business model. Section three introduces the approach in an overview, elaborates on the process reduction through an abstracted business domain, and finally applies the method in a small example, non-fictitious on the domain side. Section four elaborates on the current state of the art and on existing solutions, and delimits the contributions of the method. Section five summarizes and concludes this paper.

2 Basic assumptions and business model

Dynamics and pragmatism of real life businesses demand good enough software which is useful to the customer, and therefore support a focus on higher-order domain tests. So our approach is based on the fundamental assumption that the final arbiter of software success is only the customer to whom the component software is useful or not. This most central assumption was stated already in 1979: "*A software error is present when the program does not do what its end user reasonably expects it to do.*" (Myers 1979, p. 103).

The end user domain is the area of intended application for the component-based software. While it usually lacks a fully formal definition or model, we still assume that customer test references from the application domain prevail over test oracles created with mere supplier knowledge from within the component software technology. Higher-order testing on the domain level, without the intention to change or reengineer components or their specifications, initially has as a goal to validate the suppliers' software for reuse and control on the demand side. The argument of assertive and independent consideration of the ontological domain and the supporting technologies can be founded in Ψ (psi) theory (Dietz 2006).

From an end user's domain point of view, it is favorable to test higher-order requirements independently and as early as possible. This supports the identification and assessment of components, if possible at best before the executable software itself is available. The necessary validation knowledge consists of testable business requirements that predefine what the right software solution is supposed to do, and it needs to be constructed.

Our construction approach is embedded into a clear business model assumption derived from the vision of industrialized compositional reuse for software engineering, which has been described in detail in (Turowski 2003). Figure IV-B6-1, notation "e3-value" (Gordijn & Akkermans 2001), introduces the underlying business model assumption with the three actors: component supply, component demand and component market.

In the business model, suppliers create components for an anonymous market to satisfy an assumed demand or requirement on that market. These requirements can typically be acquired from discussions with individual clients but also could very well be entrepreneurial market assumptions. Software components offered to cover the requirements are technically mature and suppliers keep their source code undisclosed. They completely specify their components in black box style by fully defining the interfaces to convey the components' contracts (what the components do) but without

disclosing their implementation details (how the components work) (Meyer 1992). Specifications achieving this are multi-layered and semi-formal today. Respective specification approaches are proposed e.g. in (Overhage 2006; Ackermann et al. 2002) where contract levels and facts to be specified describe the external view onto the component for reuse. These component specifications can serve as black box description for reuse and are put into publicly available component specification libraries.

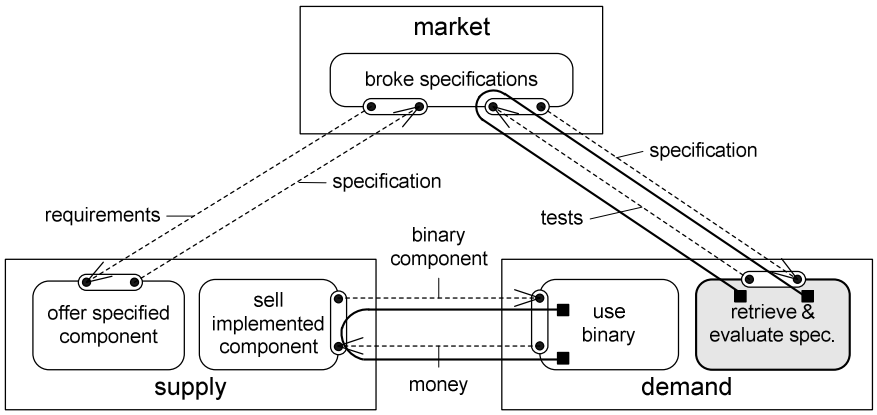

Figure IV-B6-1: Business model assumption.

Component software users on the demand side want support and automation for their requirements and search a wide variety of library components for the right software. The available components are found as specifications e.g. on the Internet. These semi-formal, multi-layered component reuse specifications represent the candidates offered by suppliers for domain testing. Customers query the black box functional specifications with specific predefined criteria, retrieve matches, and then evaluate the retrieved specifications in detail. Both retrieval and evaluation imply a comparison i.e. a test between reference features demanded and specification candidates offered.

Compositional reuse acknowledges the industrial segregation between a supply side offering components for reuse and a demand side requiring software built from reused and properly orchestrated parts. Such industry-style compositional reuse apparently requires advances to established software engineering methods, which also includes the testing stage. An important challenge for black box reuse at this point is how to derive reasonable specification retrieval and evaluation criteria, and that means: how to validate testable end user domain requirements against supplier specifications.

The associated testing may be classified as specification based or program based, and specification based testing can be divided into state based testing and black box testing (Vincenzi et al. 2003). The component paradigm of the described business model

assumes that components are tested on the basis of their specifications, and restrict our approach to black box testing. It is acknowledged that good overall testing will be comprehensive and will employ a set of complementary methods in practice. It is also acknowledged that testing alone can not improve the quality of software, but early and expressive test results can improve decisions.

3 Constructing linear scenarios

3.1 ARIval overview

A precondition for the validation of requirements is that these requirements are stated in testable terms. Figure IV-B6-2, notation "activity diagram" (Object Management Group 2005), gives an overview on the ARIval (abstraction, reduction, inclusion, validation) method (Skroch 2007), where domain level scenarios are used to validate aspects of multi-layered component reuse specifications, if possible showing that the specified software works for the higher-order domain requirements.

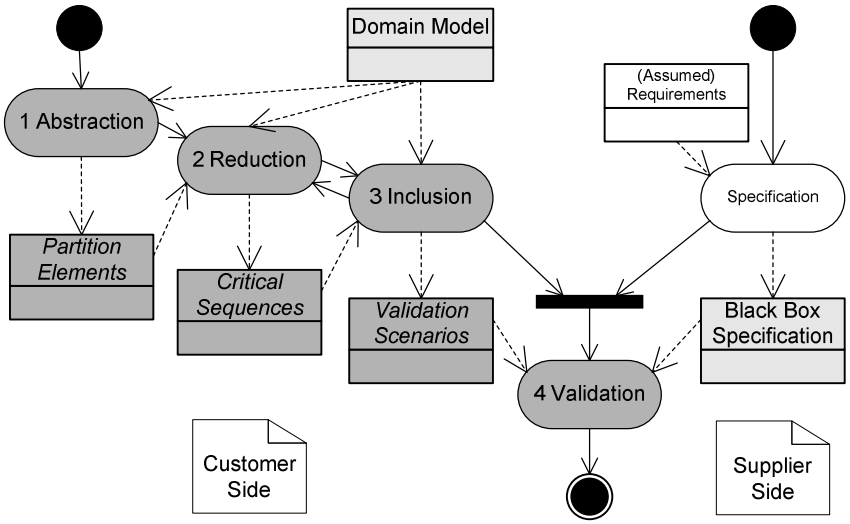

Figure IV-B6-2: ARIval overview.

To construct testable business requirements on the customer side, our first starting point is the observation that also for testing higher-order domain functionality, only a small subset of the full domain is actually relevant for the end users' intended automation with distinct effects on utilized system behavior.

The second starting point is the observation that some kind of domain model is usually available on the customer side, in many cases through prosaic business rules and process descriptions as semi formal or informal models, e.g. activity diagram, event

driven process chain, Petri net, etc. Full or partial automation is required for the model, or parts of it, from ready-made software components.

Relevant parts of the model environment are first abstracted based on the well-known equivalence partitioning and boundary value analysis, which is described and used for program testing since the late 1970s (Myers 1979). This results in domain partition elements which are a discrete representation of the original continuous domain, with one representative element per partition.

The abstracted elements are then reduced, by identifying reasonable and critical sequences. Complexity of typical requirements in real settings will lead to very many possible sequences at this point and prevent an exhaustive testing. This means that with each possible sequence of steps that requires automation on the domain side, and with the corresponding sequence of equivalence classes, a small number of critical sequences need to be selected from the very large number of all possibilities.

Selection criteria are domain centric and come from outside of the software engineering process. They include domain workflow and value flow considerations e.g. on frequency, criticality, financial or other risk, external visibility, etc. instead of software centric objectives such as coverage of all control statements in the source code. Furthermore, the sequences must not contain branching but make up linear paths in order to avoid quantitative evaluation problems during actual testing (cf. state explosion). To achieve this, a critical sequence with branches becomes de-branched until we have a number of linear sequences instead.

The abstracted and reduced domain part then contains value representatives in sequences, with each sequence linear and deemed critical by the customer for the intended application.

An inclusion will use the critical linear sequences to build scenarios, both within a domain part and across a number of different related domain parts, to cover the critical paths in their context as full business transaction flows. These scenarios must again not contain branching but make up linear paths. This can be guaranteed by constructing them accordingly, i.e. instead of a branching scenario we include two or more branch-free scenarios, until all resulting scenarios are linear at the end.

The method provides the possibility to re-iterate the reduction step, e.g. if certain sequences are found missing one can go back and establish them to be available for the construction of the respective end-to-end scenario. In this way each linear scenario is deliberately and consciously included into the validation step, or not. Inclusion criteria,

again, are domain centric and are derived from considerations rooting in domain ontology instead of software technology, as described.

Finally, the actual testing will numerically check applicable parts of the reuse specifications from the supply side using all formally defined and branch-free critical validation scenarios as test cases.

Three basic coverage measures can be defined. Two start from the abstracted domain, which is an equivalent of the original domain. Reduction coverage measures the abstracted domain against reduced sequences requirements. Inclusion coverage measures reduced sequences against included scenarios. The third measure starts from the set of scenarios. Validation coverage measures a scenario's expected results against the actual validation success. The measures could be plain and weighted. The weighted coverage would scale on numeric scores given for each reduction criteria, inclusion criteria and scenario, e.g. by using a simple ranking.

Beneficiaries of the method are mainly customers and end users in the presented business model. The ARIVal method supports them in evaluating the many component specifications from repositories on the basis of their testable requirements, independently derived from their ontological domain, and before actual software is available.

3.2 Process flow transformation and blocking

Through data abstraction, based on equivalence partitioning and boundary value analysis, we prepared a discrete data domain which is an equivalent representation of the complete and continuous original data domain. We now aim at the identification of an incomplete set of branch-free critical sequences through this abstracted, discrete domain model.

At the core is the reduction of the process domain. The approach is a double reduction: first, transformation and block building on process scheme level, and then numerical (de-)selection on the level of process instances (or, test sequences) in the simplified scheme. In this way, we deliberately resign from completeness twice. In other words, we first select the critical scheme parts from the overall process flow that need testing coverage. This leads to a simplified process scheme. The selected scheme parts that are deemed critical by the customer are at the same time numerically unfolded according to the abstracted domain model (i.e. all possible “traces” are listed that can be derived from the business rules). Now we can select a small number out of all possible numerical sequences through this simplified domain process scheme. The result is a

small critical subset out of the very large set of all possible paths through the abstracted domain model.

Criteria to be used are based on aware stakeholder priority decisions, e.g. on business criticality of different process scheme parts and of different "traces" through the simplified model. This could be measured e.g. in terms of monetary value flow per path. If e.g. in a process scheme half of the revenues are generated within a certain small subset of maybe ten percent of the full scheme, and the other half of the revenue generation happens throughout the remaining ninety percent of the scheme, then apparently the smaller subset of the scheme is probably more important for the end-user testing. In this simplified example we could even already calculate a very simple priority value from the figures. The further elaboration of underlying stakeholder criteria would lead away from the scope of this paper. At this point we just need to take the diligent assumption that we are able to prioritize process scheme parts and process instances according to their business value.

From computability theory we can derive the fundamental process flow constructs “sequence”, “join” and “split” selection (joining pre-conditions, splitting post-conditions, also known as “selection”) and “iteration”, which are also described and used as starting point for workflow patterns definitions (van der Aalst et al. 2003). Process flow patterns use constructs ranging from these simple elements up to complex processing primitives. For each of the three basic constructs, we take workflow patterns from van der Aalst et al. (2003) and show how transformation and block building works for the basic construct; the notation used in the figures is UML activity diagram (Object Management Group 2005). The full domain process flow can then be treated iteratively by treating the single basic constructs.

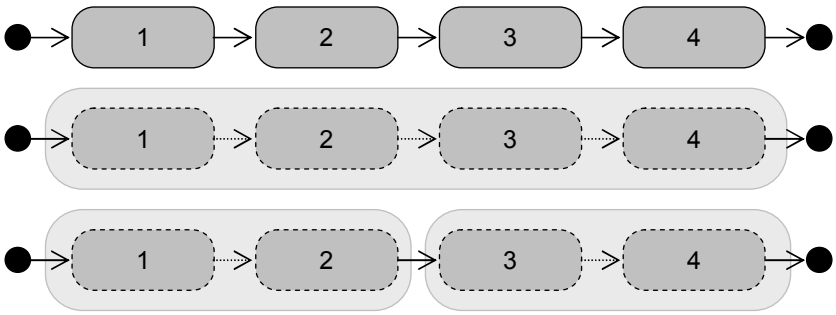

Figure IV-B6-3: Sequence blocking.

A *sequence* of process steps as shown in Figure IV-B6-3 is found in the basic workflow pattern “sequence”. It reflects the fundamental notion of an activity that is enabled after the completion of the preceding activity, and a common interpretation for the pattern is implication or causality.

As a linear sequence this basic construct is already in the form of our intended result, and we can – without further transformation – readily form a block unit by using any continuous subsequence of it; in Figure IV-B6-3, the second line shows a block built from the maximum subsequence, the third line shows an alternative block building. Each block can then be (de-)selected as a whole unit. This means that numerical test, and as a consequence also oracles, will be set at the block boundaries only; in line three of Figure IV-B6-3 before activity 1 and after activity 2, and before activity 3 and after activity 4, but not, say, after activity 1. This implies that even when including the block unit, there will be no consideration of block internals. In the reduced, simplified process scheme, the internal structure of the block is hidden.

A *split* selection of the activities flow into multiple activities as shown in Figure IV-B6-4 is found in the basic workflow patterns "XOR-split" and "AND-split". The patterns reflect the essential notion of branching activities. A common interpretation for the XOR-split or switch pattern shown on the upper left side of the figure is decision. A common interpretation for the AND-split or fork pattern shown on the upper right side of the figure is parallel processing.

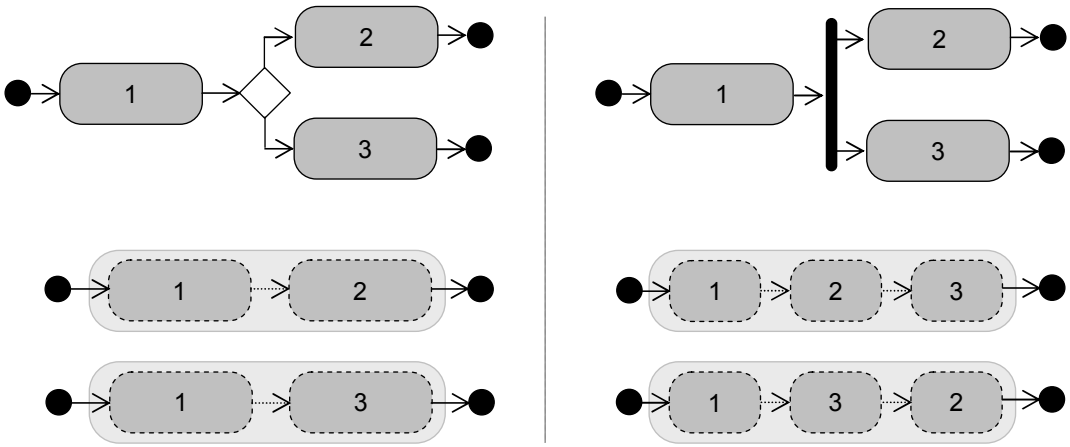

Figure IV-B6-4: Split transformation and blocking.

For both split types, we transform the process scheme into a simpler scheme for blocking as shown in Figure IV-B6-4. On a binary XOR switch, as well as on a binary AND fork, two blocks encompass the construct, one block for each of the two subsequent steps within the scheme part. Splits with more than two following steps can be handled accordingly and result in more than two blocks. The internal block structures become hidden on the simplified scheme level. Numerical selection of the single "traces" in a subsequent step is less complex and establishes linear paths. Note that the concurrency aspect of the AND-split disappears, which seems appropriate for the intended testing against reuse specifications and without executable software.

A *join* selection of the activities flow from multiple activities as shown in Figure IV-B6-5 is found in the basic workflow patterns "XOR-join" and "AND-join". It reflects the essential notion of merging activities. A common interpretation for the XOR-join pattern shown on the upper left side of the figure is trigger. A common interpretation for the AND-join pattern shown on the upper right side of the figure is synchronization.

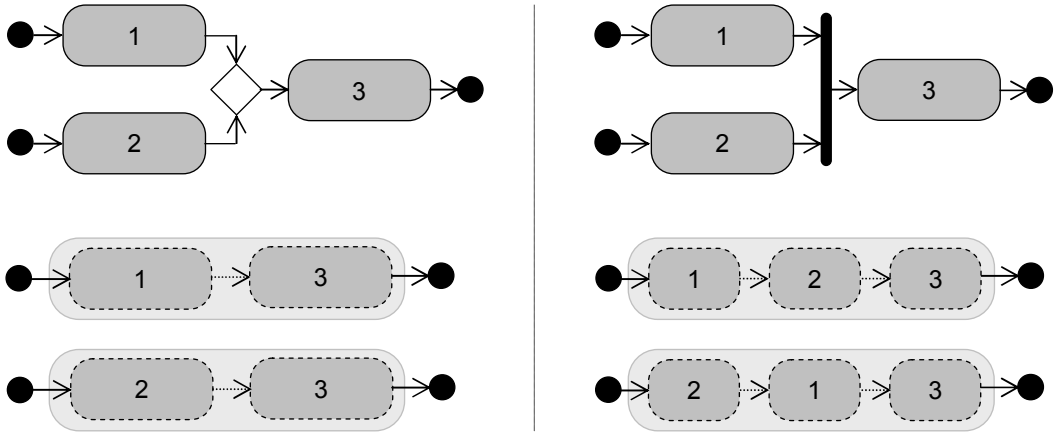

Figure IV-B6-5: Join transformation and blocking.

For both join types, we transform the process scheme into a simpler scheme for blocking as shown in Figure IV-B6-5. On a binary XOR trigger, as well as on a binary AND synchronization, two blocks encompass the whole construct, one block for each of the two preceding steps within the scheme constructs. Joins with more than two preceding steps can be handled accordingly and lead to more than two blocks. Again the internal block structures become hidden to the unfolding in the numerical reduction, when we select the "traces" in a subsequent step. Note also here that the synchronization aspect of the AND-join disappears, which again seems appropriate for the intended testing against reuse specifications and without executable software.

An *iteration* in the activities flow as shown in Figure IV-B6-6 is found in the structural workflow pattern "arbitrary cycles". It reflects the notion of activities that can be done repeatedly. A common interpretation for repeated activities patterns is loop.

For an iteration construct in a real workflow, we transform the process scheme into a simpler scheme for blocking. We use the same approach as with the other constructs and unfold the iteration primitive into single linear paths. The number of possible paths is determined by the business rules ("loop conditions"). The number can be large, even in a non-theoretical workflow, even with an abstracted data domain and single value representatives per equivalence class (as produced from the preceding abstraction step).

Our approach is to bundle equivalence classes for iterations so that as many "traces" through the loop as possible fall into the same equivalence category. We start at the general and known approach to leave out sub-paths from the transformed iteration that are passed more than k times. We argue for our validation purposes that a sub-path that is included in a related larger path needs to be looked at only once, and so we set $k = 1$ (the example given later demonstrates the application of the idea). Together with the iterative sequence blocking in our approach, we still have the possibility to explicitly include also sub-paths that were identified as business critical within the loop, if they are included in a larger path (as suggested in the second and third line in Figure IV-B6-6). So we established a basis for selecting the critical passes that are needed for inclusion as validation scenarios.

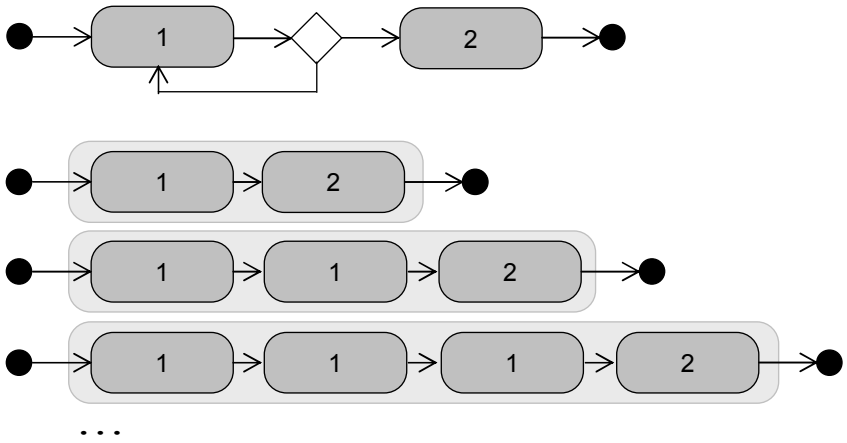

Figure IV-B6-6: Iteration transformation and blocking.

Note that for our higher-order testing of reuse specifications we can omit unsolvable cases from information theory (cf. e.g. halting problem).

With the transformation and blocking procedures we can construct paths through the abstracted domain that are (i) part of the domain under consideration, (ii) critical for the customer and (iii) linear, without branches and without cycles. We call such a path a "Sunshine Path". Sunshine Paths can be serialized by construction, because they are a linear sequence of process steps, or transactions, which produce the same result as in the originating graph, if they were completely selected. They now go into the inclusion step as building blocks for end-to-end validation scenarios.

3.3 Example

The example is non-fictitious on the domain side and is taken from a large company's business rules and processes for the creation of credit items. Figure IV-B6-7 shows one function out of the process diagram and the relevant business rules for the

"authorization level ok?" process step. A credit item has been recorded by an agent of the company at this stage. Now it needs formal release. Everyone involved in the process belongs unambiguously to a certain role, and all roles have limits for releasing (rel) a recorded credit item depending on its amount. If the credit amount is above the role's limit, it is not released by the role but instead submitted (sub) to the next superior role. Above a certain amount, any credit needs release by two different authorized roles (rel-s, rel). The two highest roles are entitled to release all credits. In the described process, credits that shall not be released remain in an undefined, or submitted, state.

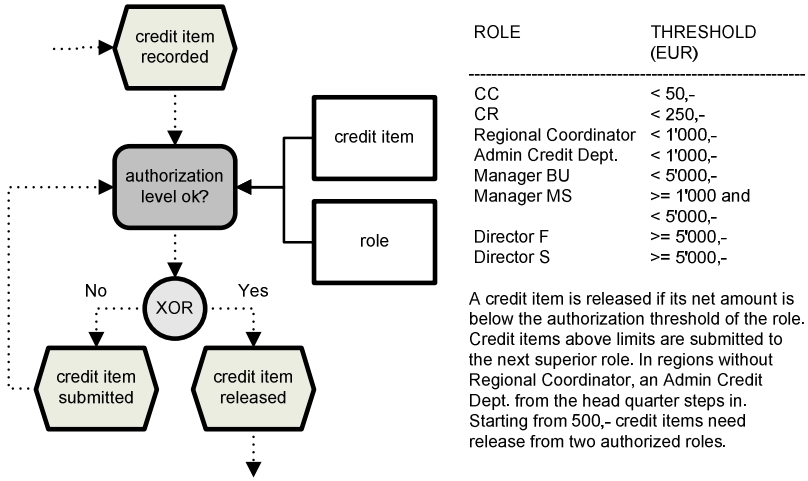

Figure IV-B6-7: Domain model excerpt.

Abstraction maps the original domain model onto an equivalent domain model with defined discrete partitions and distinct value representatives per partition. The example results in seven partitions shown in Table IV-B6-1 together with their values. If the analyzed customer domain section does not define any observable behavior, e.g. for partition P_1 in this example, then tests cannot be derived from this part of the domain model.

Table IV-B6-1: Partitions and values.

Partition	Value	Partition	Value
$P_1 =]-\infty, 0]$	$e_1 = -1$	$P_5 = [500, 1000[$	$e_5 = 500$
$P_2 =]0, 50[$	$e_2 = 25$	$P_6 = [1000, 5000[$	$e_6 = 1000$
$P_3 = [50, 250[$	$e_3 = 50$	$P_7 = [5000, \infty[$	$e_7 = 5001$
$P_4 = [250, 500[$	$e_4 = 250$		

Further simplification of the scheme and its business rules by transformation and blocking is not necessary in the simple example. Reduction can readily identify the "traces" or, data sequences that are critical and reasonable for testing from the full set of possible sequences, from an end user validation point of view.

We restrict to demonstrating positive test sequences here, negative test sequences work according to the same principle. The iteration on the example can be reduced from an end user's business perspective to sequences starting at the least empowered call center (cc) role, which will subsequently cover also superior roles with suitable partition values (i.e. no explicit check for $k > 1$ in a first approach).

This reduction results in Table IV-B6-2 listing ten Sunshine Path sequences, the building blocks for critical business scenarios through the domain section. These sequences are now eligible for inclusion, also with critical sequences from other, interconnected domain parts, to build end-to-end branch-free business scenarios. The approach to connect sequences is the same as it was shown for the steps within a domain part. Joining two scenarios becomes possible by using the preceding scenario's output as the subsequent scenario's input. Inclusion criteria, again, are fully domain centric.

Table IV-B6-2: "Sunshine path" sequences.

Role	cca	cr	rc	acd	mbu	mms	df	ds
S_1	e_2 rel	-	-	-	-	-	-	-
S_2	e_3 sub	e_3 rel	-	-	-	-	-	-
S_3	e_4 sub	e_4 sub	e_4 rel	-	-	-	-	-
S_4	e_4 sub	e_4 sub	-	e_4 rel	-	-	-	-
S_5	e_5 sub	e_5 sub	e_5 rel-s	-	e_5 rel	-	-	-
S_6	e_5 sub	e_5 sub	-	e_5 rel-s	e_5 rel	-	-	-
S_7	e_6 sub	e_6 sub	e_6 sub	-	e_6 rel-s	e_6 rel	-	-
S_8	e_6 sub	e_6 sub	-	e_6 sub	e_6 rel-s	e_6 rel	-	-
S_9	e_7 sub	e_7 sub	e_7 sub	-	e_7 sub	-	e_7 rel-s	e_7 rel
S_{10}	e_7 sub	e_7 sub	-	e_7 sub	e_7 sub	-	e_7 rel-s	e_7 rel

To demonstrate how we check a specification artifact on the basis of the sequences from Table IV-B6-2, it is assumed that a software provider has specified and offered a fictitious `Comparator` software component. Next to other levels and facts, the behavior of this software artifact is described in OCL (Object Constraint Language) from Object Management Group (2006), and a `checkGE` service ("greater or equal") is defined according to Figure IV-B6-8. It is also specified for the `Comparator` component, on the respective layer of a multi-level reuse specification, that a `limits` relation maps a `value` to an `actor`.

To check the behavior specified in Figure IV-B6-8 against customer requirements given as critical sequences, the constraints from the supplier's specification are now numerically compared with one ore more branch-free scenarios. As described, such a scenario can be one path through several subsequent critical customer sequences from interrelated domain parts that are assembled and validated together.

We assume in this example that our customer includes one single Sunshine Path, S_4 from Table IV-B6-2. So we can restrict our example to demonstrate this single sequence. In natural language, S_4 follows a recorded credit item of 250.– from a region without regional coordinator role. The credit item is (i) beyond the credit authorization limit of the call center role and therefore submitted to the customer representative role. It is (ii) beyond the credit authorization limit of the customer representative role and therefore submitted to the administrator credit department role. It is (iii) within the credit authorization limit of the administrator credit department role and released. Validation of this sequence is done by systematically walking through the OCL constraints from Figure IV-B6-8.

```
context Comparator::checkGE( val:Real,
                             act:String ):Boolean

pre:
      ( oclIsUndefined( val )=false )
      and
      ( oclIsUndefined( act )=false )
      and
      self.limits->exists( actor:String | actor=act )

post:
      if self.limits->
       select( actor:String | actor=act ).value >= val
      then result=true   -- greater or equal
      else result=false  -- not (greater or equal)
      endif
```

Figure IV-B6-8: Behavioral specification artifact (OCL).

In step (i) the first two preconditions hold: `val` is 250.– and `act` is cca. The third precondition also holds: once the mapping table is set up with role descriptions and thresholds from the domain model, then cca will be found in the `limits` relation. If the preconditions hold as described, the specification's postcondition will evaluate (50 >= 250) and return false. The work flow can identify this with the meaning that the credit item is not released, and return to the "authorization level ok?" function with a "credit item submitted" state.

In step (ii) the first two preconditions hold: `val` is 250 and `act` is cr. As in the previous step the third precondition also holds for cr. If the preconditions hold as described, the specification's postcondition will evaluate (250 >= 250) and return true. The work flow can identify this with the meaning that the credit item is released, and continue to further parts of the domain model with a "credit item released" state.

In step (iii) the first two preconditions hold: `val` is 250 and `act` is acd. As in the previous steps the third precondition also holds for acd. If the preconditions hold as

described, the specification's postcondition will evaluate (1000 >= 250) and return true. The work flow can identify this with the meaning that the credit item is released, and continue to further parts of the domain model with a "credit item released" state.

Thus, on the bottom line, validation of the `Comparator` component vs. S_4 using ARIval revealed a problem. While steps (i) and (iii) can be performed correctly by the specified software, in step (ii) the `Comparator` component fails check vs. the business rules. In the domain model and its critical sequence S_4, the credit item of 250.– is not released by a customer representative but instead submitted to be checked by the superior role. In the `Comparator` component, the validation shows that the credit item of 250.– is actually released by the customer representative role, which is inconsistent with the requirements from the domain model.

Possible consequences of this result could include looking for a `checkG` service ("greater") of the `Comparator` component, or changing the business rules slightly, or others. In any case the small but on the domain side non-fictitious validation example has shown that the proposed method gives an early hint at the necessity of a respective, aware decision and provides tangible support for it, without using any actual software.

4 Related work

Component software testing theory has become a large area of scientific research (Vincenzi et al. 2003). Important existing approaches with relation to our method have been selected and are shown in Table IV-B6-3 to demarcate original contributions of the method.

Lines in Table IV-B6-3 list the examined approaches which are further described below. Columns list three abstraction levels: component, composition and context. On component level formal program verification of single components with their interfaces is typical research focus. On composition level research from formal and less formal areas deals with architectures of several integrated components. On context level research focus is on the requirements side and less formal, concerned with system architectures in their socio-technical domain and business context. The availability of an approach for different abstraction levels is indicated in the cells. Our method's research contributions on the domain level or context level are: embedding into a clear business model, independent domain based test oracles, and early domain level testing before software is available. The analyzed existing approaches don't seem to cover this.

Built-in test technologies. Built-in technologies for self-testing software components, in analogy to built-in tests from integrated circuits, have extensively been researched, e.g.

in the Component+ project of the European Union (Edler & Hörnstein 2003). Built-in tests come within the component, e.g. as additional test services, and are not intended to represent independent customer specific automation requirements but basic technical checks. Tests built into the component by their vendors are complementary to our domain centric axiom.

Table IV-B6-3: Related approaches.

Approach	**Component (Program)** verification	**Composition (Architecture)** ver. & val.	**Context (Domain)** validation
Built-in test technology	X		
Formal methods	X	(X)	
Scenario-/model-based testing	(X)	(X)	(X)
Specification matching	X		(X)
Tabular notation	X		
Test / composition languages	(X)	X	
Test input data sampling	X	(X)	
Test output oracles	X		
This Approach (ARIval)		(X)	X

Formal methods. Especially in formal model checking, plenty of verification approaches ("are we building the software right?") are discussed, among them the interesting domain reduction abstraction (Choi & Heimdahl 2003). The method proposed here transfers some of the ideas to the domain validation ("are we building the right software?") viewpoint. But fully formal approaches for real components are prevented by computational effort with real systems in practice, decidability problems from computer theory, the absence of complete formal specifications, and the lack of a justifying business case or public interest. Also, formal verification can still be wrong. Formal verification methods provide valuable insight but in a practical sense don't apply to our complex domain level validation.

Scenario based and model based testing. Scenarios can be seen as special entities within the more general notion of a model. In model based testing, test references are generated from a model of the actual system. Many model based test approaches build upon the UML (Unified Modeling Language) today, and derive test references from UML diagrams (Offutt & Abdurazik 1999; Briand & Labiche 2002). Test references in existing approaches are built from artifacts within the component software development – models, design scenarios, etc. – and not from independent and unknown customer

requirements as proposed here. Few if any approaches have yet addressed these model independency issues and its test implications, as does our method on the domain validation side.

Specification matching. Existing approaches are based on fully formal language specifications, focus strongly on technical aspects, and are restricted to the matching of relatively simple functions (Moormann Zaremski & Wing 1997; Yellin & Strom 1997). Semi-formal matching methods from library science have also been described since long (Prieto-Díaz & Freeman 1987; Penix & Alexander 1999), and discussions exist to automatically extract classification attributes from natural language descriptions (Maarek, Berry & Kaiser 1991). Further investigations include in particular relaxations of exact matching, and also contextual refinement theory (Fidge 2002). Discussions started only recently that focus on more complex business domain perspectives for compatibility considerations of multi-layered specifications (Zaha 2004). Our method goes beyond formal technical aspects and aims at checking specifications vs. higher-order requirements represented by domain level scenarios.

Tabular notation. This approach aims at representing requirements fully formal by using a comprehensible, mathematically precise tabular notation of predicate logic for partial functions (Parnas 1993). Domain requirements are successively translated into this tabular form, with promising first practical results (Baber et al. 2005). Tabular notation seems very formal for "good enough" testing as intended in our method.

Test and composition languages. Similar to well known specification languages such as Z or OCL, special languages for testing and for composition have been proposed. One example on the testing side is TTCN-3 for test execution (Grabowski et al. 2003). An example on the composition side is the Piccola calculus for formal component composition (Achermann & Nierstrasz 2005). Test languages make implicit assumptions on their domains and their intended use, and have proven successful for testing software in their respective target areas. Architectural composition languages are formal and powerful but don't seem suitable for defining and evaluating actual test scenarios. Our method suggests a generic, widely applicable domain validation method without actual software but based on reuse specifications.

Test input data sampling. Exhaustive testing on all possible inputs is infeasible in general and inappropriate in particular for large real life enterprise applications. Hence an incomplete but appropriate test has to be determined. Existing approaches achieve this by sampling a domain of the input data according to fault hypotheses i.e. assumptions about which aspects or entities are error prone, allowing the test to reveal as many failures as possible with a minimum effort (Beizer 1995). In our method, tests

are generated not from fault hypotheses within the technological software system or its specification or models, but instead independently from the actual customer's ontological domain and its automation requirements which are unknown to, and detached from, the component software technology provider.

Test output oracles. The test oracle question (Turing 1939; Weyuker 1982) relates to outputs produced by a test: if the actual results differ from the expected results, did a proper test run produce wrong results revealing a software error, or were the expected results and/or the testing and/or basic assumptions wrong in the first place? Particular test outputs need careful analysis if the oracle grounds on the same model as the software (Pretschner & Philipps 2005). Related issues can be observed in the controversial discussions of N-version programming in the 1980s. Sophisticated approaches such as e.g. (Hummel & Atkinson 2005) exist today. Our method instead sets priority to tests created independently from a software user, to deliver the independent oracle and the final judgment about an expected feature of a reused component.

5 Summary and conclusions

Compositional reuse for industry style software production is an important approach pursued to master the ever increasing demands on software intensive systems. Testing black box software components from large repositories for their suitability to be reused in an actual end user situation is among the problems that complicate this approach. The associated validation activities are supported by the ARIval method, offering to the component demand side a domain centric component validation approach. The approach has some core advantages: it is derived from a clear business model assumption, sources test oracles from business domain requirements independent from the technological development process, and produces tangible results early, before the executable software is available, on the basis of suppliers' reuse specifications.

We demonstrated the principle in an example which is non-fictitious on the domain side. By constructing critical scenarios via abstraction, reduction and inclusion from a domain model, we obtain branch-free Sunshine Paths of automation sequences deemed validation critical on the domain level of the demand side. These scenarios represent references against which relevant levels from multi-dimensional supplier black box specifications can be checked very early in the compositional development process, and with oracles that are independent from this development process.

With our approach we support early and independent higher-order black box component software testing on the demand side in industrialized software processes. This can benefit software component customers through earlier and better testing within further decomposed division of work as required for industrialized software engineering processes.

References (B6)

Achermann, F.; Nierstrasz, O. (2005), "A calculus for reasoning about software composition", *Theoretical Computer Science*, 331 (2-3): 367-396.

Ackermann, J.; Brinkop, F.; Conrad, S.; Fettke, P.; Frick, A.; Glistau, E.; Jaekel, H.; Kotlar, O.; Loos, P.; Mrech, H.; Ortner, E.; Raape, U.; Overhage, S.; Sahm, S.; Schmietendorf, A.; Teschke, T.; Turowski, K. (2002), "Standardized Specification of Business Components", Gesellschaft für Informatik, Augsburg.

Baber, R.; Parnas, D.; Vilkomir, S.; Harrison, P.; O'Connor, T. (2005), "Disciplined Methods of Software Specification: A Case Study", *Proceedings of the International Symposium on Information Technology: Coding and Computing*, IEEE Computer Society, 4-6 April 2006, Las Vegas, USA: 428-437.

Beizer, B. (1995), *Black-Box Testing: Techniques for Functional Testing of Software and Systems*, Wiley, New York, USA.

Biggerstaff, T.; Richter, C. (1987), "Reusability Framework, Assessment, and Directions", *IEEE Software*, 4 (2): 41-49.

Boehm, B. (2005), "The Future of Software Processes", *Unifying the Software Process Spectrum: Proceedings of the International Software Process Workshop: Revised Selected Papers*, Lecture Notes in Computer Science 3840, Springer, 25-27 May 2005, Beijing, China: 10-24.

Briand, L.; Labiche, Y. (2002), "A UML-Based Approach to System Testing", *Journal of Software and Systems Modeling*, 1 (1): 10-42.

Brooks, F. (1987), "No Silver Bullet: Essence and Accidents of Software Engineering", *IEEE Computer*, 20 (4): 10-19.

Choi, Y.; Heimdahl, M. (2003), "Model Checking Software Requirement Specifications using Domain Reduction Abstraction", *Proceedings of the 18th IEEE International*

Conference on Automated Software Engineering, IEEE Computer Society, 6-10 October 2003, Montreal, Canada: 314-317.

Dietz, J. (2006), *Enterprise Ontology: Theory and Methodology*, Springer, Berlin.

Edler, H.; Hörnstein, J. (2003), Component+ Final report 1.1., retrieved 12 October 2005, <http://www.component-plus.org/pdf/reports/Final report 1.1.pdf>.

Fidge, C. (2002), "Contextual Matching of Software Library Components", *Proceedings of the 9th Asia-Pacific Software Engineering Conference*, IEEE Computer Society, 4-6 December 2002, Gold Coast, Australia: 297-306.

Gao, J.; Tsao, H.; Wu, Y. (2003), *Testing and quality assurance for component-based software*, Artech House, Boston, USA.

Gordijn, J.; Akkermans, H. (2001), "Designing and Evaluating E-Business Models", *IEEE Intelligent Systems*, 16 (4): 11-17.

Grabowski, J.; Hogrefe, D.; Réthy, G.; Schieferdecker, I.; Wiles, A.; Willcock, C. (2003), "An introduction to the testing and test control notation (TTCN-3)", *Computer Networks*, 42 (3): 375-403.

Hummel, O.; Atkinson, C. (2005), "Automated Harvesting of Test Oracles for Reliability Testing", *Proceedings of the 29th Annual International Computer Software and Applications Conference*, IEEE Computer Society, 25-28 July 2005, Edinburgh, UK: 196-202.

Maarek, Y.; Berry, D.; Kaiser, G. (1991), "An Information Retrieval Approach For Automatically Constructing Software Libraries", *IEEE Transactions on Software Engineering*, 17 (8): 800-813.

McIlroy, M. (1969), "Mass Produced Software Components", *Software Engineering: Report on a conference sponsored by the NATO Science Committee*, NATO Scientific Affairs Division, 7-11 October 1968, Garmisch: 138-155.

Meyer, B. (1992), "Applying 'Design by Contract'", *IEEE Computer*, 25 (10): 40-51.

Meyer, B. (2003), "The Grand Challenge of Trusted Components", *Proceedings of the 25th International Conference on Software Engineering*, IEEE Computer Society, 3-10 May 2003, Portland, USA: 660-667.

Mili, H.; Mili, F.; Mili, A. (1995), "Reusing Software: Issues and research directions", *IEEE Transactions on Software Engineering*, 21 (6): 528-562.

Moormann Zaremski, A.; Wing, J. (1997), "Specification Matching of Software Components", *ACM Transactions on Software Engineering and Methodology*, 6 (4): 333-369.

Myers, G. (1979), *The Art of Software Testing*, Wiley, New York, USA.

Object Management Group (2005), Unified Modeling Language: Superstructure version 2.0, retrieved 31 March 2006, <http://www.omg.org/docs/formal/05-07-04.pdf>.

Object Management Group (2006), UML 2.0. OCL Specification, retrieved 24 October 2006, <http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf>.

Offutt, J.; Abdurazik, A. (1999), "Generating Tests from UML Specifications", *The Unified Modeling Language – Beyond the Standard: Proceedings of the Second International Conference*, Lecture Notes in Computer Science 1723, Springer, 28-30 October 1999, Fort Collins, USA: 416-429.

Overhage, S. (2006), "Vereinheitlichte Spezifikation von Komponenten: Grundlagen, UNSCOM Spezifikationsrahmen und Anwendung", Dissertation, Universität Augsburg.

Parnas, D. (1993), "Predicate Logic for Software Engineering", *IEEE Transactions on Software Engineering*, 19 (9): 856-862.

Parnas, D. (2001), "Software Aspects of Strategic Defense Systems", in Hoffman, D.; Weiss, D. (eds.), *Software Fundamentals: Collected Papers by David L. Parnas*, Addison Wesley, Boston, USA: 497-518.

Penix, J.; Alexander, P. (1999), "Efficient Specification-Based Component Retrieval", *Automated Software Engineering*, 6 (2): 139-170.

Pretschner, A.; Philipps, J. (2005), "Methodological Issues in Model-Based Testing", in Broy, M.; Jonsson, B.; Katoen, J.; Leucker, M.; Pretschner, A. (eds.), *Model-Based Testing of Reactive Systems: Advanced Lectures*, Lecture Notes in Computer Science 3472, Springer, Berlin: 281-291.

Prieto-Díaz, R.; Freeman, P. (1987), "Classifying Software for Reusability", *IEEE Software*, 4 (1): 6-16.

Skroch, O. (2007), "Validation of Component-based Software with a Customer Centric Domain Level Approach", *Proceedings of the 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, IEEE Computer Society, 26-29 March 2007, Tucson, USA: 459-466.

Szyperski, C.; Gruntz, D.; Murer, S. (2002), *Component Software: Beyond Object-Oriented Programming*, 2nd edition, Addison Wesley, London, UK.

Turing, A. (1939), "Systems Of Logic Based On Ordinals", *Proceedings of the London Mathematical Society*, s2-45 (1): 161-228.

Turowski, K. (2003), *Fachkomponenten: Komponentenbasierte betriebliche Anwendungssysteme*, Shaker, Aachen.

van der Aalst, W.; ter Hofstede, A.; Kiepuszewski, B.; Barros, A. (2003), "Workflow Patterns", *Distributed and Parallel Databases*, 14 (1): 5-51.

Vincenzi, A.; Maldonado, J.; Delamaro, M.; Spoto, E.; Wong, W. (2003), "Component-Based Software: An Overview of Testing", in Cechich, A.; Piattini, M.; Vallecillo, A. (Eds.), *Component-Based Software Quality: Methods and Techniques*, Lecture Notes in Computer Science 2693, Springer, Berlin: 99-127.

Weyuker, E. (1982), "On Testing Non-testable Programs", *The Computer Journal*, 25 (4): 465-470.

Weyuker, E. (1998), "Testing Component-Based Software: A Cautionary Tale", *IEEE Software*, 15 (5): 54-59.

Yellin, D.; Strom, R. (1997), "Protocol Specifications and Component Adaptors", *ACM Transactions on Programming Languages and Systems*, 19 (2): 292-333.

Yourdon, E. (1995), "When good enough software is best", *IEEE Software*, 12 (3): 79-81.

Zaha, J. (2004), "Automated compatibility tests for business related aspects of software components", *On the Move to Meaningful Internet Systems: Workshop Proceedings*, Lecture Notes in Computer Science 3292, Springer, 25-29 October 2004, Agia Napa, Cyprus: 834-841.

V Fazit und Ausblick

V.1 Fazit

Die vorgestellten Beiträge haben Forschungsfragen der Wirtschaftsinformatik im komponenten- und serviceorientierten Leitbild der Entwicklung betrieblicher Anwendungssysteme (Turowski 2003, S. 9-15) behandelt. Das Kernthema war, aus unterschiedlichen Perspektiven, die Gestaltung von Software-Entwicklungsaufgaben. Die Beiträge haben – im Sinne einer durchgängigen Betrachtung – innerhalb des strategischen Rahmens der Wiederverwendung (langfristig) unterschiedliche Aspekte der Spezifikation (taktisch) und Auswahl (operativ) von Komponenten bzw. von durch diese implementierten Services berührt.

Die ersten beiden Beiträge haben strategische Rahmenbedingungen der Wiederverwendung behandelt und sind dabei vom Multipfad-Vorgehensmodell (Ortner 1998, S. 332; Overhage 2006, S. 136) ausgegangen.

In Beitrag B1 gelang es zunächst, zwei grundsätzliche verschiedene Wiederverwendungsstrategien – kompositorische und generative Wiederverwendung – voneinander abzugrenzen sowie zwei idealtypische Marktumfelder zu beschreiben, stabile Märkte der „old economy“ und turbulente „high tech“ Marktbedingungen. Darauf aufbauend, und unterstützt durch die Analyse dreier Praxisprojekte mit Wiederverwendungsanteilen, konnte eine neue Theorie zur Präferenz von bestimmten Wiederverwendungsansätzen in bestimmten Marktmilieus in zwei wohlbegründeten und konkreten Hypothesen formuliert werden. Die Forschungsziele des Beitrags sind damit erreicht worden. Um die konkrete Hypothesenformulierung zu ermöglichen musste allerdings ein „reduktionistischer“ Ausgangspunkt eingenommen sowie einschränkende Annahmen getroffen werden. So besitzen die beschriebenen Marktbedingungen stark idealtypischen Charakter, ebenso die strategische Entscheidungsalternativen, von der die Theorie ausgeht. Weiterhin betreffen die beschriebenen Strategien nur jeweils ein einziges Marktumfeld und nicht gleichzeitig unterschiedliche Marktumfelder. Darüber hinaus gibt es auch weitere Faktoren, die neben den Marktmilieus Einfluss auf die Art der Wiederverwendung haben können. Der Beitrag begründet sinnvolle Hypothesen, die nicht als umfassender Wiederverwendungsleitfaden, aber als ein Schritt in Richtung der rationalen Identifikation strategischer Wiederverwendungspräferenzen in Abhängigkeit vom Marktumfeld dienen können.

In Beitrag B2 gelang es, die „make-or-buy“ Problematik an großen Referenzprojekten – mit einer in dieser Art und Größe bislang selten beschriebenen Realitätsnähe – zu

untersuchen. Dabei konnte in einem Referenzprojekt detailliert analysiert werden, zu welchem Grad die Anforderungen an ein komplexes, intra- und interorganisationales Anwendungssystem durch die Entwicklung von Individualsoftware erfüllt sind. In einem zweiten Schritt konnte, ebenso detailliert, verglichen werden, wie weit dieselben Anforderungen von einer Kombination von Softwarepaketen erfüllt sind, die zur Wiederverwendung und Integration am Markt erhältlich waren. Ergänzend konnten entsprechende Erfahrungswerte aus weiteren Großprojekten untersucht werden, die dem Referenzprojekt in bestimmten Aspekten ähnelten. Während der Vergleich der Anforderungsüberdeckung durch verschiedene Entwicklungsstrategien im detailliert untersuchten Referenzprojekt zugunsten der Wiederverwendung ausfiel, ergab die Analyse der weiteren Projekte kein eindeutiges Bild. Die Forschungsziele des Beitrags sind damit teilweise erreicht worden. Die Ergebnisse deuten zunächst darauf hin, dass es neben der Frage nach Individualentwicklung oder Kauf von der Stange weitere wichtige Erfolgsfaktoren gibt. Hinzu kommt, dass die naheliegende Option einer kombinierten „make-*and*-buy“ Strategie im Beitrag nicht untersucht werden konnte, da die betrachteten Projekte ohne diese Alternative gesteuert wurden. Weiterhin ist möglich, dass die im Beitrag untersuchten Projekte nicht in jeder Hinsicht repräsentativ sind.

Die beiden Beiträge zu den taktischen Aspekten haben sich auf die Spezifikation als besonders herausfordernden und vielleicht wichtigsten Teil der komponenten- und serviceorientierten Softwareentwicklung konzentriert (Alpar et al. 2008, S. 294; Sommerville 2001, S. 107).

In dem Beitrag B3 wurde zunächst die zentrale Rolle von Anforderungsspezifikationen und den daraus abgeleiteten Leistungsbeschreibungen in der betrieblichen Praxis der arbeitsteiligen Softwareentwicklung dargestellt. Danach gelang es, kritische Erfolgsfaktoren zu identifizieren, die die Anfertigung hochwertiger Anforderungsspezifikationen in der Praxis begünstigen. Schließlich wurden noch die Risiken dargestellt, die sich aus unklaren Leistungsbeschreibungen ergeben können – bis hin zum ungünstigsten Fall, dem Rechtstreit. Diese ausdrückliche Verbindung interdisziplinärer Aspekte an der Schnittstelle zwischen Wirtschaftsinformatik und Recht stellt das Novum des Beitrags dar, der seine Forschungsziele damit erreicht hat. Die Ergebnisse des Beitrags beruhen dabei auf der Einschätzung von wenigen, langjährig in Theorie und Praxis erfahrenen Experten, ohne umfassende Literaturanalysen und kontrollierte Experimente zur empirischen Bestätigung oder Widerlegung.

In dem Beitrag B4 konnte ein Ansatz zur umfassenden Bewertung von Anforderungsspezifikationen für spätere Offshore-Entwicklungsschritte wissenschaftlich begründet und im Rahmen einer großen Einzelfallstudie kontrolliert angewandt werden. Die

Praxistauglichkeit des vorgestellten Ansatzes konnte anhand seiner reibungslosen Durchführung im realen industriellen Kontext gezeigt werden. Die mit dem Ansatz erzielten Ergebnisse wurden anhand des tatsächlichen, weiteren Projektverlaufs validiert. Damit ist ein Schritt in Richtung auf ein bislang in dieser Art nicht existierendes, systematisches Verfahren zur Planungs- und Entscheidungsunterstützung in Offshore-Entwicklungssituationen auf der Grundlage von Anforderungsspezifikationen getan. Die Forschungsziele des Beitrags sind somit erreicht worden. Die empirische Bestätigung beruht dabei auf einer Einzelfallstudie. Dieser Einzelfall ist zwar groß, real und relevant, und der vorgestellte Ansatz ist unproblematisch auch in weiteren Fallstudien anwendbar. Trotzdem kann noch nicht von seiner uneingeschränkten Generalisierbarkeit ausgegangen werden.

Die abschließenden beiden Beiträge haben sich mit der operativen Serviceauswahl aus der Sicht der Nachfrager im Leitbild der komponenten- und serviceorientierten Entwicklung betrieblicher Anwendungssysteme (Turowski 2003, S. 9-15) befasst.

In Beitrag B5 gelang es zunächst, Besonderheiten der opportunistischen ad hoc Suche nach geeigneten (Web-)Services im Internet zu bestimmen. Darauf aufbauend konnten Ergebnisse aus der mathematisch-statistischen Theorie des optimalen Stoppens zur Verbesserung dieser Suche erstmals in zwei Szenarien für selbst-rekonfigurierende, serviceorientierte Systeme angewandt werden. Dafür wurden die den Anwendungsszenarien entsprechenden Stoppalgorithmen hergeleitet und implementiert. Der operative Vorteil offener, dynamischer, durch optimales Stoppen verbesserter Systeme gegenüber geschlossenen, statischen Systemen wurde in Simulationsexperimenten gemessen und bestätigt. Die Forschungsziele des Beitrags sind damit erreicht worden. Die Grundannahme des Beitrags ist dabei, dass es mehrere funktional äquivalente Dienste im Internet geben wird, die geeignet sind, eine bestimmte, kleine betriebliche Teilaufgabe innerhalb eines komponenten- und serviceorientierten Gesamtsystems zu erfüllen. Diese Annahme wird derzeit nicht von allen Experten geteilt. Weiterhin beschreibt der Beitrag Verbesserungen für eine optimierende Suche, die semantische und pragmatische Suchkriterien zu berücksichtigen hat, welche derzeit zwar zu den viel diskutierten, aber noch nicht abschließend gelösten Forschungsfragen gehört.

In Beitrag B6 wurde gezeigt, wie ein komplexer Geschäftsprozess in einzelne, lineare Durchläufe zerlegt werden kann, damit diese Abläufe als Gerüst für die Bestimmung durchgängiger Prüfszenarien verwendet werden. Mit Hilfe solcher Szenarien ist die Brauchbarkeit von wiederzuverwendenden Komponenten und Diensten bereits aufgrund ihrer Spezifikation überprüfbar. Bisher in dieser Form selten beschriebene Vorteile ergeben sich dabei aus der Möglichkeit zur frühzeitigen „higher-order“ Prüfung mit

Testorakeln, die vom Entwicklungsprozess unabhängig sind, einschließlich der Möglichkeit zur Bestimmung von Testabdeckungsmaßen. Die Forschungsziele des Beitrags sind damit teilweise erreicht worden. Einschränkungen ergeben sich, da der Beitrag von der systematischen Linearisierbarkeit von Geschäftsprozessmodellen ausgeht, diese hängt jedoch u.a. wesentlich von der für die Modellierung verwendeten Notationsgrammatik ab. Weiterhin geht der Beitrag von der Prüfbarkeit von Spezifikationen aus, die aber von der genauen Art und dem Formalisierungs- und Detailgrad der Spezifikation abhängt. Der Beitrag enthält ein reales Beispiel zur Linearisierung eines EPK-Ausschnitts, zur Überführung in ein Testszenario und zur entsprechenden Überprüfung eines OCL-Spezifikationsartefakts. Der Beitrag erläutert aber nicht allgemein, wie aus einem verzweigungsfreien Durchlauf ein Testszenario entsteht, und welche zusätzlichen Angaben dafür noch zu erstellen sind, d.h. die Generalisierbarkeit der Vorgehensweise wird nicht abschließend diskutiert.

Mit den vorgestellten Ergebnissen zur Spezifikation und Auswahl von Services im strategischen Rahmen der wiederverwendungsgetriebenen Entwicklung komponenten- und serviceorientierter betrieblicher Anwendungssysteme wurden mit der Arbeit sechs Beiträge geleistet.

V.2 Ausblick

Vier der sechs Beiträge dieser Arbeit haben ihre Forschungsziele erreicht (B1, B3, B4, B5), bei den weiteren beiden Beiträgen ist dies zumindest teilweise gelungen (B2, B6). Interessante Anknüpfungspunkte für die weitere Forschung lassen sich aus jedem Beitrag ableiten.

In Beitrag B1 sind zwei Hypothesen zur strategischen Präferenz für Software-Wiederverwendungsansätze in Abhängigkeit vom jeweiligen Marktumfeld begründet worden. In zukünftigen Forschungsvorhaben könnten diese Hypothesen weiter verfeinert bzw. erweitert werden. Die Hypothesen sollten letztlich auch einer – vermutlich aber nicht einfach zu leistenden – empirischen Überprüfung zugeführt werden.

In Beitrag B2 konnten die strategischen Vorteile der Wiederverwendung im Vergleich zur Individualentwicklung in einem detailliert untersuchten, großen Referenzprojekt bestätigt werden. Die Analyse weiterer, zum Referenzprojekt ähnlich gelagerter Fälle hat allerdings kein klares Bild für oder gegen die Wiederverwendung mehr ergeben. Dieser zweiten Beobachtung könnte in weiterführenden Forschungsarbeiten nachgegangen werden, etwa mit dem Ziel weitere, bislang noch nicht entdeckte Erfolgsfaktoren zu isolieren.

In Beitrag B3 wurde aufgrund von Experteneinschätzungen die (auch interdisziplinäre) Bedeutung klarer Anforderungsspezifikationen in der arbeitsteiligen Entwicklung dargestellt. Kritische Erfolgsfaktoren ihrer Anfertigung wurden beleuchtet und Risiken ihrer Vernachlässigung geschildert. Diese Expertenaussagen aus der Praxis könnten in weiterführenden Forschungsaktivitäten theorieseitig mit den Ergebnissen einer umfassenden Literaturanalyse ergänzt werden. Weiterhin könnte die besonders hohe Bedeutung der Anforderungsspezifikation in den zwei aktuellen Forschungsrichtungen komponenten- und serviceorientierte Architekturen bzw. Offshore-Softwareentwicklung vertieft und empirisch untersucht werden.

In Beitrag B4 konnte gezeigt werden, wie mittels einer theoretisch fundierten, systematischen Methode umfangreiche Anforderungsspezifikationen auf ihre Tauglichkeit im Offshore-Entwicklungsumfeld hin untersucht werden können. An die hierfür in einem industriellen Kontext mit positiven Ergebnissen durchgeführte Einzelfallstudie können sich interessante und viel versprechende Forschungsaktivitäten anschließen. Eine mögliche Richtung ist die stärkere empirische Überprüfung und gegebenenfalls Anpassung der Methode, indem diese noch in anderen möglichst realen und aussagekräftigen Praxisfällen kontrolliert angewandt wird. Eine andere mögliche Forschungsrichtung ist die weitere analytische Detaillierung, Formalisierung und gegebenenfalls Erweiterung des vorgestellten Bewertungsansatzes.

In Beitrag B5 wurde die dynamische Suche nach geeigneten (Web-)Services im Internet in zwei Szenarien durch die Herleitung und Anwendung von Algorithmen aus der optimalen Stopptheorie verbessert. Hierbei wurde von stabilen Anforderungen und der opportunistischen Suche nach einem jeweils optimal geeigneten Dienst ausgegangen. Für die Entdeckung von Diensten wurde dabei, ohne Beschränkung der Allgemeinheit, von der Gleichverteilung der Ereignisse („Service-Entdeckungen“) über die Zeit ausgegangen. Nachfolgende Forschungsarbeiten könnten sich der Ermittlung empirischer Verteilungsfunktionen widmen. Ebenfalls könnte der Ansatz auf seine Anwendbarkeit in Situationen mit mehreren Optimierungskriterien und Zielkonflikten untersucht werden, also etwa Servicequalität gegen Kosten der Serviceverwendung. Eine anders gelagerte, ebenfalls interessante Erweiterung der Forschungsfragen könnte sich aus dem Versuch ergeben, den beschriebenen Ansatz auf Systeme anzuwenden, die sich dynamisch an instabile funktionale Anforderungen anpassen. Dabei kann es ein Vorteil für die künftige Forschung sein, dass einfach anpassbare, leistungsfähige und wiederverwendbare Simulationskomponenten implementiert wurden.

In Beitrag B6 gelang es an einem realen Beispiel, verzweigungsfreie Abläufe aus komplexen Geschäftsprozessmodellen zu extrahieren, auf deren Grundlage durchgängige

Testszenarien zu bilden und bestimmte Spezifikationsartefakte wiederzuverwendender Komponenten gegen diese Testszenarien zu prüfen. Es schließt sich eine Reihe interessanter Forschungsfragen an. So könnte untersucht werden, unter welchen Voraussetzungen das Modell eines Geschäftsprozesses überhaupt linearisierbar im Sinne der Methode ist. Dabei sollten Barrieren der theoretischen Berechenbarkeit durch pragmatische Annahmen und empirische Erhebungen über wirklich existierende Geschäftsprozessmodelle ergänzt werden. Weiterhin könnte beschrieben werde, wie genau aus den verzweigungsfreien Durchläufen fertig ausdefinierte Testszenarien erzeugt werden, darunter beispielsweise die Frage, welche Testdaten benötigt werden und wie diese zu bestimmen sind. Ebenfalls könnte untersucht werden, welchen Voraussetzungen Spezifikationen genügen müssen, damit sie überprüfbar sind. Schließlich könnte auch die Bestimmung von Metriken zur Messung des Abdeckungsgrades von „higher-order" Tests eine vor allem für die Praxis interessante Weiterentwicklung sein.

Über den direkt aus den einzelnen Beiträgen identifizierten weiteren Forschungsbedarf hinaus kann festgestellt werden, dass die gestalterische Aufgabe von Software-Entwicklungsvorhaben im komponenten- und serviceorientierten Leitbild betrieblicher Anwendungssysteme bei weitem noch nicht vollständig untersucht ist. Wie auch die vorliegende Arbeit gezeigt hat, lässt gerade diese Forschungsrichtung auf Ergebnisse und Fortschritte hoffen, die die Wettbewerbsfähigkeit von Unternehmen stärken können und die auch zukünftig ein wissenschaftlich herausforderndes und ergiebiges Forschungsfeld innerhalb der Wirtschaftsinformatik mittragen.

Literatur (Fazit und Ausblick)

Alpar, P.; Grob, H.; Weimann, P.; Winter, R. (2008), *Anwendungsorientierte Wirtschaftsinformatik: Strategische Planung, Entwicklung und Nutzung von Informations- und Kommunikationssystemen*, 5. Auflage, Vieweg, Wiesbaden.

Ortner, E. (1998), „Ein Multipfad-Vorgehensmodell für die Entwicklung von Informationssystemen – dargestellt am Beispiel von Workflow-Management Anwendungen", *Wirtschaftsinformatik*, 40 (4): 329-337.

Overhage, S. (2006), „Vereinheitlichte Spezifikation von Komponenten: Grundlagen, UNSCOM Spezifikationsrahmen und Anwendung", Dissertation, Universität Augsburg.

Sommerville, I. (2001), *Software Engineering*, 6. Auflage, Pearson Studium, München.

Turowski, K. (2003), *Fachkomponenten: Komponentenbasierte betriebliche Anwendungssysteme*, Shaker, Aachen.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ACM	Association for Computing Machinery
AHP	Analytic Hierarchy Process
ARIval	Abstraction, Reduction, Inclusion and Validation
avg.	average
BGH	Bundesgerichtshof
BSS	Business Support System
BU	Business Unit
CAPEX	Capital Expenditure
CC	Call Center
COTS	Commercial Off-The-Shelf
CR	Customer Representative
CRM	Customer Relationship Management
CUA	Cost Utility Analysis
DSP	Digital Signal Processing
DV	Datenverarbeitung
DW	Data Warehousing
dept.	department
e. g.	exemplum gerendi
EPK	Ereignisgesteuerte Prozesskette
exp.	experiment
GI	Gesellschaft für Informatik e. V.
GI-FB WI	Fachbereich Wirtschaftsinformatik der GI
GoBS	Grundsätze ordnungsgemäßer DV-gestützter Buchführungssysteme
HMD	Handbuch der modernen Datenverarbeitung
IEEE	Institute of Electrical and Electronics Engineers
ICT	Information and Communication Technology
IN	Intelligent Network

IP	Internet Protocol
IS	Information System
ITU	International Telecommunication Union
ITU-T	ITU – Telecommunication Standardization Sector
MIS	Management Information System
NGN	Next Generation Network
OCL	Object Constraint Language
OMG	Object Management Group
OSS	Operations Support System
P1 bis P9	Phasen im Multipfad-Vorgehensmodell
PDP-6	Programmed Data Processor 6
QoS	Quality of Service
RdNr.	Randnummer
SEAA	Software Engineering and Advanced Applications
SIG	Special Interest Group
spec.	specification
TCO	Total Cost of Ownership
TTCN	Testing and Test Control Notation
UK	United Kingdom
UML	Unified Modeling Language
UNSCOM	Unified Specification of Components
val.	validation
ver.	verification
VHB	Verband der Hochschullehrer für Betriebswirtschaft e. V.
VoIP	Voice over IP
WI	Wirtschaftsinformatik
WKWI	Wissenschaftliche Kommission Wirtschaftsinformatik
Z	Zermelo-Fraenkel Notation

Symbolverzeichnis

#	Beitragsnummer
$\rightarrow$	konvergiert gegen
$\varnothing$	leere Menge
∞	unendlich
~	ungefähr
>>	viel größer als
A	σ-Algebra
E	Erwartungswert
e	Eulersche Zahl
ε	beliebig kleine reelle Zahl größer als Null
F	Verteilungsfunktion
I	Indikatorfunktion
inf	Infimum
ln	natürlicher Logarithmus
max	Maximum
O	Landau-Symbol
Ω	Menge aller Elementarereignisse
P	Wahrscheinlichkeitsmaß
sup	Supremum
σ	Sigma-Operator
T	Klasse von Stoppregeln
τ	Stoppregel