Architekturzentrierte agile Anwendungsentwicklung in global verteilten Projekten

Dissertation zur Erlangung des Doktorgrades

an der Fakultät für Mathematik, Informatik
und Naturwissenschaften,
Fachbereich Informatik der Universität Hamburg

vorgelegt von
Joachim Sauer

Hamburg, Dezember 2010

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Erster Gutachter: Prof. Dr.-Ing. Heinz Züllighoven, Universität Hamburg
Zweiter Gutachter: Prof. Dr. Volker Wulf, Universität Siegen

Die mündliche Prüfung fand am 8. Dezember 2010 statt.

Für meine Eltern

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Zusammenfassung

Die global verteilte Anwendungsentwicklung hat international und auch in Deutschland in den letzten Jahren an Bedeutung gewonnen. Von der Verlagerung auf mehrere verteilte Entwicklungsstandorte versprechen sich die Auftraggeber insbesondere geringere Kosten, einen leichteren Zugriff auf qualifiziertes Personal, das in einem Land nicht ausreichend verfügbar ist, und eine größere Nähe zum Endanwender. Eine besondere Form der global verteilten Entwicklung ist das Offshoring, bei dem Aufträge an einen geografisch entfernten, unabhängigen IT-Dienstleister vergeben werden, der die Entwicklung überwiegend auf einem anderen Kontinent erledigt. Die Erfahrungen mit global verteilter Entwicklung sind eher negativ. In vielen Projekten konnten die avisierten Kosteneinsparungen nicht realisiert werden und mehrere Projekte sind gescheitert.

In dieser Arbeit wird untersucht, inwiefern der Einsatz von agilen Methoden in global verteilten Entwicklungsprojekten Probleme vermeiden und die Erfolgswahrscheinlichkeit erhöhen kann. Bei agiler Entwicklung wird Wert auf Flexibilität, Kommunikation und ein iteratives Vorgehen sowie eine offene Kommunikation zwischen allen Beteiligten gelegt. Anhand der Analyse veröffentlichter Erfahrungsberichte und dem Abgleich mit wissenschaftlichen Erkenntnissen wird aufgezeigt, dass sich in der Praxis charakteristische Probleme aus dem Einsatz agiler Methoden in verteilten Entwicklungsprojekten ergeben. Insbesondere ist es schwierig, eine effiziente Kommunikation und Koordination zwischen allen Beteiligten zu gewährleisten.

Um den Problemen zu begegnen, wird in dieser Arbeit agile verteilte Entwicklung mit weiteren Ansätzen kombiniert. Dies ist zum einen das Dual-Shore-Modell, bei dem sowohl onshore in der Nähe des Auftraggebers als auch offshore beim ausländischen Dienstleister entwickelt wird. Zum anderen die architekturzentrierte Entwicklung, bei der die Softwarearchitektur im Mittelpunkt vieler Aktivitäten der Softwareentwicklung steht und u. a. zur Arbeitsteilung und -koordination, Fortschrittsprüfung, Wissensvermittlung und Qualitätssicherung dient.

Durch Auswertung von eigenen Erfahrungen, veröffentlichten Erfahrungsberichten und Feldstudien werden Vor- und Nachteile der Ansätze herausgearbeitet und ihre Eignung für die agile verteilte Entwicklung diskutiert. Die Ergebnisse werden in Musterform festgehalten. Die Einsatzmöglichkeiten der Muster werden in zwei global verteilten Entwicklungsprojekten untersucht. So kann in der Praxis nachgewiesen werden, dass die in dieser Arbeit herausgearbeiteten Ansätze eine gute Grundlage für die agile verteilte Entwicklung bilden.

Abstract

Over the past years, global software development has gained in importance both internationally and in Germany. Customers hope to profit from the relocation to several distributed development sites by lower costs, an easier access to qualified staff that is not available in one country, and closer proximity to end users. A special type of global software development is offshoring where orders are placed with a geographically distant, independent IT service provider who carries out the development mainly on another continent. Present experience with global software development is rather negative. Many projects missed the anticipated cost savings. Several projects failed.

In this paper we investigate how the application of agile methods to globally distributed development projects helps to avoid problems and increases the probability of success. Agile software development emphasizes flexibility, communication and an iterative proceeding as well as open communication among all participants. By means of an analysis of published experience reports and an alignment with scientific findings, we show that the application of agile methods to distributed development projects in practice leads to characteristic problems. It is particularly difficult to provide for efficient communication and coordination among all participants.

In order to avoid these problems, we combine agile distributed development with further approaches in this paper. One of these approaches is the dual shore model which connects onshore development close to the customer with offshore work at the foreign service provider's development site. Another one is architecture-centric development which places the software architecture center-stage of many tasks during the software development and aids in the separation and coordination of tasks, progress monitoring, knowledge transfer, and quality management, among other things.

Through the evaluation of our own experiences, published experience reports, and field studies, we identify advantages and disadvantages of the approaches and discuss their suitability for agile distributed development. The results are recorded as patterns. The possible applications of these patterns are evaluated in two globally distributed development projects. Thereby we demonstrate in practice that the approaches which are developed in this paper form a good basis for agile distributed development.

Danksagung

Von den vielen Menschen, die mich unterstützt und zum Gelingen dieser Arbeit beigetragen haben, möchte ich Einzelne herausheben und mich bei ihnen besonders bedanken.

Als Erstes danke ich den beiden Gutachtern dieser Arbeit. Heinz Züllighoven hat mich auf meinem akademischen Weg seit dem Studium begleitet und war der Hauptbetreuer dieser Dissertation. Volker Wulf hat sich über seine Gutachtertätigkeit hinaus eingebracht und insbesondere dabei geholfen, die Arbeit zu einem runden Abschluss zu bringen.

Mehrere Menschen haben Teile der Arbeit durchgesehen und mit mir diskutiert. Carola Lilienthal hat große Teile gelesen, gerade im Bereich Softwarearchitektur, und war mir stets ein Vorbild für zielgerichtetes Promovieren in der Softwaretechnik. Ronald Krick hat die erste umfassende Version dieser Arbeit revidiert und mit mir betriebswirtschaftliche Aspekte des Offshoring diskutiert. Alexander Boden war mir eine Hilfe beim Verständnis der kulturellen Dimension der global verteilten Arbeit.

Meinen Kollegen am Arbeitsbereich Softwaretechnik verdanke ich eine angenehme, motivierende Arbeitsatmosphäre. Christiane Floyd hat mir als Arbeitsbereichsleiterin den Weg bereitet und mir Mut gemacht. Ihre Ausführungen zu Forschungsmethodik haben mich konzeptionell vorangebracht. In einer selbst organisierten Diskussionsrunde haben wir Promovierende uns in regelmäßigen Treffen gegenseitig moralisch und inhaltlich unterstützt. Zu dieser Gruppe gehörten neben meinen langjährigen Zimmergenossen Petra Becker-Pechau und Jörg Rathlev auch Stefan Hofer und Eugen Reiswich. Bei den beiden „Senioren“ Axel Schmolitzky und Wolf-Gideon Bleek bedanke ich mich neben ihrer Vorbildfunktion auch für hilfreiche Ratschläge und Aufmunterung, wenn es mal nicht so flüssig lief.

Nur durch die Rücksicht und die Unterstützung vieler Kollegen der C1 WPS war es mir möglich, meine Dissertation neben dem Beruf abzuschließen. Mit Guido Gryczan konnte ich über alle Themen offen und vertrauensvoll sprechen. Andreas Kornstädt hat mir bei gemeinsamen Veröffentlichungen nicht nur den Spannungsbogen von Artikeln, sondern auch ein besseres Englisch nähergebracht. Bei Jörn Koch, Aleksander Koleski und Thomas Dorka bedanke ich mich für ihre Geduld bei langen, mehrfachen Interviews. Auch allen weiteren Interviewpartnern meiner Feldstudien aus Deutschland, Irland, Polen, Russland, den USA und Indien bin ich zu Dank verpflichtet.

Danksagung

Meine Familie hat die Zeit des Arbeitens an der Dissertation meist geduldig ertragen und mich liebevoll unterstützt. Meinen Eltern bin ich sehr dankbar, dass sie mich von klein auf gefördert und mir den Weg zum Studium bereitet haben. Inma, meine geliebte Verlobte, hat wohl am direktesten erfahren, wie viel Arbeit in dieser Dissertation steckt, wie viel Freude mir aber auch der wissenschaftliche Austausch und die Reisen zu entfernten Konferenzen und Interviews bereitet haben, auf denen sie mich oft begleitet hat.

Hamburg, im Dezember 2010
Joachim Sauer

1 Einleitung

Um Kosten bei der Entwicklung von Anwendungssoftware zu sparen und um näher am Endanwender zu sein, verteilen viele Unternehmen Entwicklungsarbeiten auf mehrere globale Standorte. Eine besondere Form der verteilten Entwicklung ist das Offshoring, das spätestens seit Mitte der 90er Jahre international und auch in Deutschland ein für die Praxis sehr relevantes Thema ist. Hauptmotive für die Verlagerung von Entwicklungsarbeiten sind neben der Kostensenkung durch ein niedrigeres Lohnniveau und die Nähe zum Endanwender auch eine Steigerung der Flexibilität, die Konzentration auf das Kerngeschäft und der Zugriff auf qualifiziertes Personal, das im eigenen Land nicht ausreichend verfügbar ist.

1.1 Motivation

Global verteilte Entwicklung und insbesondere das Offshoring der Softwareentwicklung werden in der Literatur hauptsächlich aus den Perspektiven der Betriebswirtschaft und der Wirtschaftsinformatik diskutiert, weniger aus technischer Sicht. Eine Betrachtung aus softwaretechnischer Sicht ist jedoch wichtig, da durch das verteilte Arbeiten und die Verlagerung von Entwicklungsaufgaben in Niedriglohnländer nicht nur tief greifende und nachhaltige Auswirkungen auf die Beschäftigten [Nicklisch et al. 2008], sondern auch auf die Technologien [Meyer 2006] erwartet werden (vgl. Abschnitt 2.4.2).

Diese Forschungslücke wurde mittlerweile erkannt. Global verteilte Softwareentwicklung wird in der neueren Forschungsliteratur als ein umfangreiches, interdisziplinäres Themengebiet mit beträchtlichem weiteren Forschungsbedarf gesehen [Damian u. Moitra 2006]. Obwohl erste Forschungsergebnisse vorliegen, fehlen immer noch eine gründliche konzeptionelle Aufarbeitung des Themas und ein genaues Verständnis der Mechanismen und Zusammenhänge.

Konzeptionelle und methodische Hilfestellungen sind dringend nötig, da die bisherigen praktischen Erfahrungen mit global verteilter Entwicklung eher negativ sind. Mehrere Projekte haben zu unbefriedigenden Ergebnissen geführt oder sind gescheitert. Die Softwaretechnik hat sich mit dem allgemeinen Phänomen von scheiternden Projekten in den letzten Jahren intensiv auseinandergesetzt. Es konnten Vorgehensmodelle, Methoden und Techniken entwickelt werden, welche die Softwareentwicklung flexibler und besser plan- und steuerbar machen. Diese Entwicklungen sollten auch in der global verteilten Entwicklung genutzt werden, um die Erfolgsquote zu erhöhen.

Diese Arbeit möchte einen Beitrag dazu leisten, die Forschungslücke bei global verteilter Softwareentwicklung weiter zu verringern. Es sollen auch praktisch nutzbare Empfehlungen bereitgestellt werden.

1.2 Themen und Fragestellungen

Diese Arbeit beschäftigt sich mit der Frage, wie softwaretechnische Konzepte eingesetzt werden können, um Schwierigkeiten der global verteilten Entwicklung von Anwendungssoftware zu bewältigen. Dabei konzentriert sich die Betrachtung auf den Einsatz von agilen Methoden, wie beispielsweise Extreme Programming [Beck 1999]. Davon erhofft man sich eine Kombination der Vorteile beider Ansätze: Flexibilität durch agile Methoden bei gleichzeitiger Kosteneinsparung durch Nutzung von global verteilter Entwicklung mit Offshore-Ressourcen.

Agile Methoden wurden schon in einigen global verteilten Entwicklungsprojekten eingesetzt. Dabei haben sich besondere Probleme ergeben (siehe z. B. [Ramesh et al. 2006], [Šmite et al. 2010]). So kann es schwierig sein, eine für agiles Arbeiten wichtige offene und häufige Kommunikation zwischen allen Beteiligten zu gewährleisten. Gleiches gilt für den Wissenstransfer und die Schaffung einer gemeinsamen Vision der zu entwickelnden Anwendung. Hinzu kommen spezielle Probleme mit einzelnen agilen Techniken, z. B. mit dem Programmieren in Paaren und der Anwesenheit des Kunden vor Ort [Sauer 2006b]. Daher besteht noch beträchtlicher Forschungsbedarf.

Mit der Bearbeitung dieses Themenkomplexes sollen Antworten auf die folgenden Forschungsfragen zur global verteilten Entwicklung mit agilen Methoden gefunden werden:

(FF 1) Welche Themen sind in der Praxis besonders problematisch?
(FF 2) Mit welchen Mitteln kann der Einsatz von agilen Methoden unterstützt werden?
(FF 3) Wie können Empfehlungen für eine erfolgreiche praktische Umsetzung dokumentiert und evaluiert werden?

Im Verlauf dieser Arbeit wird eine Fokussierung auf bestimmte Themen vorgenommen, damit die Ergebnisse konkret und in ähnlichen Kontexten anwendbar sind. Dies ist wichtig, da Verallgemeinerungen von Forschungsergebnissen in der Regel mit steigendem Komplexitätsgrad der untersuchten Fälle schwieriger werden. Gerade in der Softwaretechnik ist unklar, inwieweit empirische Forschungsergebnisse verallgemeinerbar sind [Prechelt 2001].

Im Einzelnen geht es in dieser Arbeit um

- Neuentwicklung von Anwendungssoftware, im Gegensatz zu z. B. Application Service Providing (ASP), Application (Lifecycle) Management und Business Process Outsourcing (BPO) (siehe Abschnitt 2.2.3),

- softwaretechnische Fragestellungen, nicht um wirtschaftliche oder rechtliche Themen (siehe Abschnitt 2.4.2) und
- kleine und mittlere Projekte mit knappen Ressourcen (siehe Abschnitt 3.4.3).

Trotz dieser Fokussierung sollten sich auch Anregungen für Forschungsgebiete ergeben, die in dieser Arbeit nicht direkt erfasst werden.

1.3 Methodik

In dieser Arbeit wird ein qualitatives, empirisch fundiertes Forschungsdesign verfolgt. Das bedeutet, dass Methoden und Theorien dem Untersuchungsgegenstand angemessen ausgewählt und angewendet werden, dass unterschiedliche Perspektiven berücksichtigt und analysiert werden und dass die Reflexion des Forschers über die Forschung als Teil der Erkenntnis verstanden wird [Flick 2006, S. 16 ff]. Bei den Untersuchungen werden sozialwissenschaftliche Methoden eingesetzt, um die Forschungsfragen zu beantworten. Aus den empirischen Erfahrungen werden möglichst generelle Ergebnisse abgeleitet und somit neues Wissen geschaffen.

Heinzl argumentiert in [Heinzl 2001], dass ein nach diesen Prinzipien gestalteter Ansatz eine synergetische Ergänzung zum im deutschsprachigen Raum vorherrschenden konstruktiven Forschungsparadigma darstellt, das sich an einer ingenieurwissenschaftlichen Vorgehensweise orientiert. Prechelt betont, dass in der Softwaretechnik ein empirisches Vorgehen für einen Wissenszuwachs erforderlich sei, da „die betrachteten Prozesse zu komplex für eine mathematische Analyse sind und sich ohnehin nur unvollständig abstrakt erfassen lassen“ [Prechelt 2001, S. 33]. Fast immer sei nur durch Empirie ein Verständnis über den Vorgang der Softwarekonstruktion zu gewinnen. Daher sind auch in dieser Arbeit theoretische Fundierung und konstruktive Umsetzung eng verwoben.

Die Arbeit folgt den Prinzipien des Collaborative Practice Research, das ein empirisch fundiertes Forschungsdesign ermöglicht, so wie es oben beschrieben wurde [Mathiassen 2000]. Collaborative Practice Research bietet Methoden, um formale, strukturierte Forschung mit praktischen Erfahrungen zu verbinden. Dabei werden verschiedene Erkenntnisbereiche mit unterschiedlichen wissenschaftlichen Methoden ausgewertet und zu einem Gesamtergebnis verbunden.

In dieser Arbeit werden die folgenden Erkenntnisbereiche betrachtet: Es werden veröffentlichte Erfahrungsberichte systematisch ausgewertet, um ein Verständnis für die generellen Themen, Probleme und Lösungsansätze der global verteilten Entwicklung mit agilen Methoden zu gewinnen. In einer Feldstudie werden eigene Erfahrungen des Autors eingebracht. Die gewonnenen Erkenntnisse werden mit der Auswertung von weiteren Feldstudien abgesichert. Die in diesen Erkenntnisbereichen jeweils eingesetzten Methoden werden in den nächsten Abschnitten detailliert dargestellt.

1.3.1 Erfahrungsberichte

Erfahrungsberichte aus der Praxis bieten den gewünschten empirischen Bezug. Sie werden mit qualitativen Methoden ausgewertet. Dabei kommen Verfahren aus der Grounded Theory zum Einsatz. Der Grounded-Theory-Ansatz wurde von Barney Glaser und Anselm Strauss in den 1960er Jahren begründet [Glaser u. Strauss 1967]. In späteren Jahren wurde der Ansatz überwiegend von Strauss weiterentwickelt [Strauss u. Corbin 1996]. Der Grounded-Theory-Ansatz ist „eine Auswertungstechnik zur Entwicklung und Überprüfung von Theorien, die eng am vorgefundenen Material arbeitet bzw. in den Daten verankert (grounded) ist.“ [Bortz u. Döring 2006, S. 332]. Der Grounded-Theory-Ansatz ist wissenschaftstheoretisch begründet und soll helfen, die Lücke zwischen Theorie und empirischer Forschung zu schließen. Es lassen sich unterschiedliche Perspektiven berücksichtigen und realitätsnahe Theorien erarbeiten. Im Gegensatz zu anderen Ansätzen entsteht der theoretische Rahmen erst im Verlauf der Forschung.

Der Grounded-Theory-Ansatz besteht aus einer Reihe von aufeinander aufbauenden Verfahren. Diese enthalten Konzepte und Techniken, um Daten zu gewinnen, zu analysieren und zu neuen Theorien zu verdichten. Ein wesentliches Verfahren ist das Kodieren. Das Kodieren ist ein Teil der Datenanalyse. Es dient der Konzeptualisierung von empirisch gewonnenen Daten. Dadurch wird von Einzelfällen abstrahiert und es können generellere Aussagen getroffen werden.

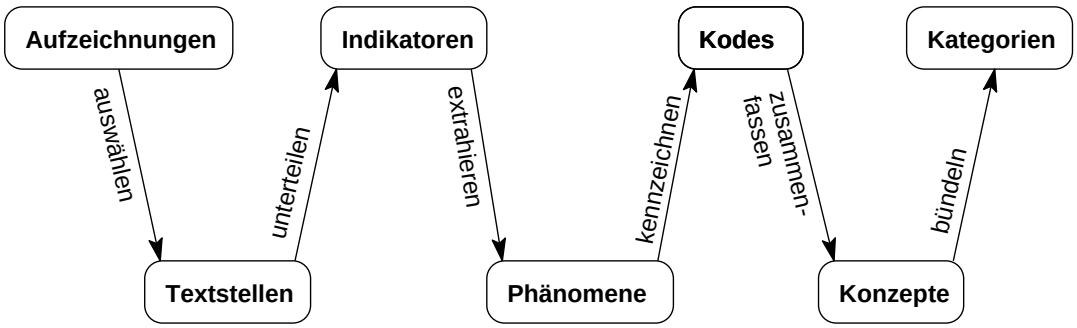


Abbildung 1.1: *Illustration der Schritte beim offenen Kodieren*

Beim *offenen* Kodieren (engl. open coding), das in dieser Arbeit angewendet wird, gehen die Forscher mit offenen Erwartungen an die Gewinnung von Daten, im Gegensatz zu anderen Kodierungsformen wie z. B. dem axialen oder dem selektiven Kodieren. Aus den Aufzeichnungen werden Textstellen ausgewählt und in sinnvolle Analyseeinheiten (Wörter, Satzteile oder Sätze) unterteilt, vgl. Abbildung 1.1. Aus diesen Einheiten, die auch Indikatoren genannt werden, werden beim Kodieren Phänomene, d. h. bestimmte Gegenstände oder Themen, extrahiert. Um die wichtigsten Phänomene zu finden, werden Fragen an den Text gestellt: zu den handelnden Personen, den Rollen, in denen sie auftreten und ihren Interaktionen. Die Handlungen werden bezüglich der

zugrunde liegenden Absichten und ihren Zwecken untersucht. Anschließend werden weitere Textstellen gesucht, in denen diese, ähnliche oder kontrastierende Phänomene beschrieben werden [Muckel 2007]. Die gefundenen Phänomene werden mit einem Kode gekennzeichnet, der die Phänomene zusammenfassend bezeichnet. Man unterscheidet natürliche Kodes, die Originalzitate und Fachbegriffe aus den Texten übernehmen, von Kodes, die von den Forschern selbst auf Basis der Texte und ihres Wissens konstruiert werden. Anschließend werden die Kodes miteinander verglichen und zu abstrakteren Einheiten mit gleichen oder zumindest ähnlichen Inhalten zusammengefasst. Diese werden Konzepte genannt. Dieser Analyseprozess wird iterativ auf Basis weiterer Aufzeichnungen durchgeführt. Die Konzepte sind dabei nicht fest, sondern werden an neu gewonnene Erkenntnisse angepasst.

Ein weiterer Aggregierungsschritt besteht im Bilden von Kategorien. Kategorien bündeln zusammengehörige Konzepte. Sie werden meist vom Forscher formuliert, sind abstrakter als Konzepte und sollen diese in einen Zusammenhang stellen. Die Kategorien entstehen folglich erst bei der Auswertung, was die gewünschte Offenheit ermöglicht [Pandit 1996].

1.3.2 Eigene Erfahrungen

Erfahrungen des Autors aus einem verteilten Projekt werden im Rahmen einer Feldstudie eingebracht. Nach [Prechelt 2001, S. 43] ist eine Feldstudie in der Softwaretechnik „jede empirische Studie, die ihre Beobachtungen direkt einem realen Softwareprojekt entnimmt und nicht die Form eines kontrollierten Experiments hat.“ Feldstudien unterscheiden sich von *Fall*studien dadurch, dass sie Werkzeuge und Methoden anhand realer Projekte beschreiben und nicht „anhand eines konkreten Anwendungsbeispiels, das eigens zu diesem Zweck unter künstlichen oder typischen Bedingungen ausgeführt wird.“ [Prechelt 2001, S. 41]

Gegenstand der Feldstudie ist ein agiles Entwicklungsprojekt mit iterativ-inkrementeller Vorgehensweise[1]. Die Vorgänge im Entwicklungsprojekt werden permanent analysiert. Verbesserungen, die sich aus der wissenschaftlichen Betrachtung ergeben, fließen direkt wieder in den Entwicklungsprozess zurück.

Den wissenschaftstheoretischen Hintergrund des Vorgehens in der Feldstudie bildet die Forschungsmethodik des Action Research (auf Deutsch übersetzbar mit Aktions- oder Handlungsforschung). Diese geht auf den deutschen Psychologen Kurt Lewin zurück, der im Jahr 1946 den Begriff prägte [Lewin 1946]. Er setzte den Akzent auf partizipative Gruppenprozesse in Organisationen. Die Forscher sollen mehrfach die Schritte Planung, Handlung und Reflexion über die Ergebnisse der Handlung wiederholen, um Erkenntnisse zu gewinnen.

[1] Bei einer iterativ-inkrementellen Vorgehensweise erfolgt die Softwareentwicklung schrittweise *(inkrementell)*. Der Entwicklungszyklus wird dabei mehrfach wiederholt *(iterativ)*.

Bei der modernen Form des Action Research werden Theorie und Praxis gleichberechtigt behandelt. Die Forscher sind in das Projekt eingebettet und aktiv beteiligt. Sie arbeiten mit Praktikern zusammen, um das Projekt durch Änderungen und Verbesserungen voranzubringen [Mathiassen 1998]. Phasen der aktiven Projektteilnahme wechseln mit Phasen der kritischen Reflexion. Der sich daraus ergebende Erkenntnisgewinn führt zu neuem, die Wirklichkeit beeinflussendem Handeln. Dieser iterative Forschungsprozess passt sehr gut zu einem agilen Vorgehensmodell in Entwicklungsprojekten.

Nach Susman und Evered [Susman u. Evered 1978] besteht Action Research aus den folgenden Stufen:

1. Analyse des Problems (engl. diagnosing),
2. Planen von Handlungen, um das Problem anzugehen (action planning),
3. Ausführen der Handlungen (taking action),
4. Reflexion des Erfolges (evaluating) und
5. Dokumentation der Erfahrungen (specifying learning).

Bei der Dokumentation der Erfahrungen werden Forschungstagebücher als flexibles, schwach strukturiertes Medium eingesetzt [Jepsen et al. 1989].

1.3.3 Ergänzende Feldstudien

Mit zwei weiteren Feldstudien wird die empirische Basis verbreitert. Die Feldstudien stützen sich auf Interviews mit Projektbeteiligten und die Analyse von Entwicklungsdokumenten.

Durch die mündlichen Interviews mit Personen, die an der Softwareentwicklung in den jeweiligen Projekten beteiligt waren, werden relevante Vorkommnisse und Beobachtungen aus den untersuchten Projekten ermittelt. Die wichtigsten Fälle werden ausgewählt. Sie werden zunächst ohne Wertung beschrieben. Anschließend werden sie jeweils kritisch reflektiert. Dadurch wird der Entdeckungskontext methodisch vom Rechtfertigungskontext getrennt. Am Ende erfolgt eine Gesamtbewertung.

Das Vorgehen bei der Datenerhebung wird vorher systematisch anhand einschlägiger Literatur geplant, siehe z. B. [Flick 2006]. Als wesentliches Mittel zur Informationsermittlung werden mündliche Befragungen durch Leitfaden-Interviews eingesetzt. Während der Interviews wird eine weiche bis neutrale Gesprächsatmosphäre angestrebt. Bei einem weichen Interview ist die Gesprächsführung einfühlsam und entgegenkommend. Dies dient dazu, „dem Befragten auf diese Art seine Hemmungen zu nehmen und ihn zu reichhaltigeren und aufrichtigeren Antworten anzuregen“ [Bortz u. Döring 2006, S. 239]. Bei einem neutralen Interview ist der Befragte ein gleichwertiger Partner des Interviewers, der „unabhängig von seinen Antworten und ohne Vorbehalte voll akzeptiert“ wird [Bortz u. Döring 2006, S. 239].

Durch einen vom Autor spezifisch für das jeweilige Interview erstellten Leitfaden werden die Art und die Inhalte des Gesprächs festgelegt. In den Leitfäden werden offene Fragen (freie Antwort möglich) und halb offene Fragen (gewisse Elemente der Antworten werden vorgegeben) verwendet. Geschlossene Fragen (Antwortmöglichkeiten werden vorgegeben) und sehr offene Fragen werden in Anlehnung an Cockburn bewusst vermieden. Wie dieser ausführt, besteht bei geschlossenen Fragen die Gefahr, viele Nuancen und damit wichtige Informationen zu verpassen. Komplett offene Fragen können dagegen die Gesprächspartner zum Abschweifen bringen und es erschweren, Gemeinsamkeiten zwischen Interviews zu erkennen [Cockburn 2003]. Die Gespräche werden mit Zustimmung der Interviewten aufgezeichnet. Nach Abschluss der Interviews werden ihre wichtigsten Teile als Basis für die spätere Auswertung transkribiert. Zusätzlich werden während der Interviews und direkt im Anschluss Gesprächsnotizen erstellt. Auf Basis dieser Unterlagen erfolgt die weitere Auswertung der Feldstudien.

1.4 Weiterer Aufbau der Arbeit

Diese Arbeit ist in sieben Kapitel und einen Anhang eingeteilt. Nach diesem Einleitungskapitel folgen drei Teile. Vergleiche die grafische Darstellung in Abbildung 1.2.

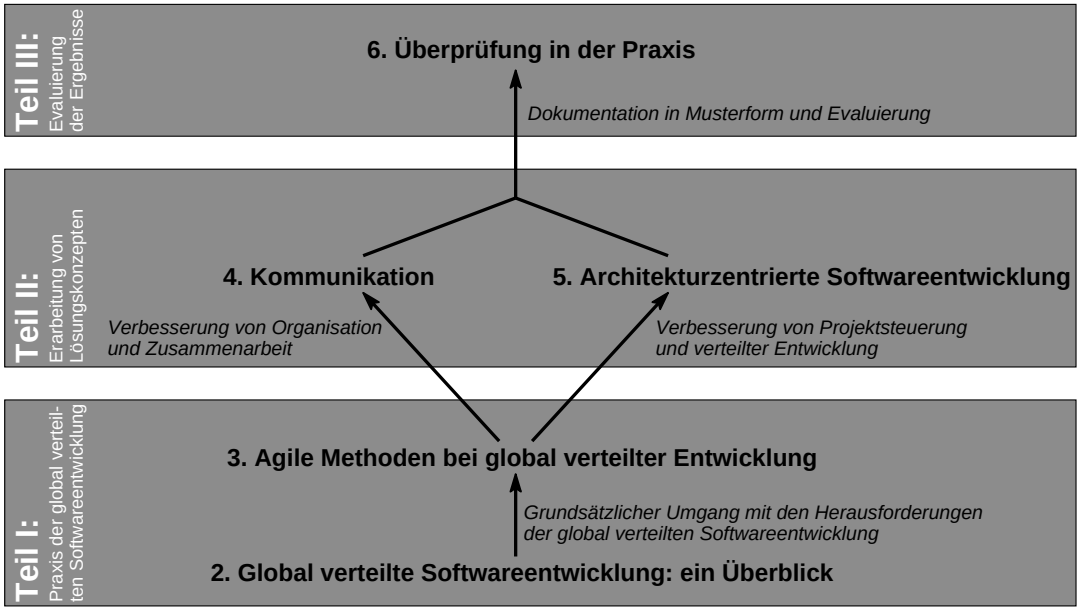


Abbildung 1.2: *Vereinfachte Darstellung des weiteren Aufbaus der Arbeit*

Teil I: Praxis der global verteilten Softwareentwicklung

In Teil I werden Grundlagen, Besonderheiten und Schwierigkeiten der global verteilten Entwicklung aufgezeigt.

Das **zweite Kapitel** gibt einen Überblick über global verteilte Entwicklung, Outsourcing und Offshoring und schafft damit eine begriffliche und konzeptionelle Grundlage für die weitere Arbeit. Nach grundlegenden Begriffsdefinitionen und Abgrenzungen zu verwandten Gebieten werden die mit global verteilter Entwicklung verbundenen Erwartungen untersucht. Aus den Gründen, die für global verteilte Entwicklung und Offshoring sprechen, und den sich in der Praxis ergebenden charakteristischen Herausforderungen werden wissenschaftliche Fragestellungen abgeleitet. Der Schwerpunkt der Diskussion liegt dabei auf der Anwendungsentwicklung.

Im **dritten Kapitel** werden agile Methoden als eine Möglichkeit des Umgangs mit den Schwierigkeiten der global verteilten Softwareentwicklung betrachtet. Dazu werden die Merkmale von agilen Methoden diskutiert und ihr praktischer Einsatz bei der verteilten Entwicklung durch Auswertung von veröffentlichten Erfahrungsberichten untersucht. So wird ein Katalog von praktischen Themen der global verteilten Entwicklung mit agilen Methoden gebildet. Die Erfahrungen, charakteristische Probleme und Lösungsansätze werden kritisch reflektiert und mit wissenschaftlichen Erkenntnissen abgeglichen.

Teil II: Erarbeitung von Lösungskonzepten

In Teil II werden die zuvor entwickelten Fragestellungen aufgegriffen und wissenschaftlich untersucht.

Im **vierten Kapitel** werden die Themen Kommunikation, Koordination und Kooperation aufgegriffen, die im entwickelten Katalog von praktischen Themen eine zentrale Rolle spielen. Wichtige Kooperationsmodelle für Offshoring-Projekte werden anhand des im dritten Kapitel erstellten Katalogs analysiert und bewertet. Der Schwerpunkt liegt auf dem Dual-Shore-Modell, bei dem sowohl onshore in der Nähe des Kunden als auch offshore beim ausländischen Auftragnehmer entwickelt wird. Dabei kommunizieren Onshore- und Offshore-Teams intensiv miteinander. Zeitweise arbeiten einige Akteure im anderen Team mit.

Im **fünften Kapitel** werden weitere zentrale Themen der Zusammenarbeit aus dem Katalog aufgegriffen. Für deren Lösung wird der Einsatz von architekturzentrierter Softwareentwicklung bei der global verteilten Entwicklung mittels einer Feldstudie in einem Offshoring-Projekt mit einem deutschen und einem indischen Team untersucht. Architekturzentrierte Softwareentwicklung begreift die Softwarearchitektur als Blaupause für alle Aktivitäten der Softwareentwicklung, die Modellierern, Entwicklern, Managern und weiteren Akteuren zur Abstimmung dient. Als Ergebnisse der

Untersuchung werden Problemfelder, Aufgaben und Nutzen herausgearbeitet und Empfehlungen für den praktischen Einsatz von architekturzentrierter Entwicklung in global verteilten Entwicklungsprojekten erarbeitet.

Teil III: Evaluierung der Ergebnisse

In Teil III wird aufgezeigt, wie sich die Ergebnisse in die Praxis zurückführen und dort anwenden lassen.

Im **sechsten Kapitel** werden die wichtigsten der neu gewonnenen Empfehlungen für die global verteilte Anwendungsentwicklung in Musterform festgehalten. Diese Muster werden anschließend anhand zweier Feldstudien auf ihre Einsetzbarkeit in der Praxis hin untersucht, um die Ergebnisse abzusichern.

Im Resümee im **siebten Kapitel** werden die Ergebnisse der Arbeit zusammengefasst und kritisch gewürdigt. In einem Ausblick werden offene Fragen und weiterer Forschungsbedarf beschrieben.

Der Anhang enthält eine Diskussion von agilen Methoden für die verteilte Entwicklung sowie Auszüge aus wichtigen Dokumenten der Empirieprojekte und der Interviews.

1.5 Allgemeine Bemerkungen

Die Orthografie dieser Arbeit richtet sich nach dem amtlichen Regelwerk des Rats für deutsche Rechtschreibung, das zum August 2006 in Kraft trat. Bei Wahlmöglichkeiten wird die im Rechtschreibduden vorgeschlagene Schreibweise (siehe [Duden 2009]) verwendet. Englische Begriffe werden vermieden, sofern sinngemäße deutsche Wörter etabliert sind. Ausnahmen von dieser Regel werden gemacht, wenn die englischen Begriffe in der deutschsprachigen Literatur weit verbreitet und ihre deutschen Übersetzungen sperrig sind. Zitate sind in der Originalsprache belassen. Dies gilt auch für die Interviews aus den Feldstudien. Bei englischsprachigen Zitaten werden englische Anführungszeichen verwendet, ansonsten deutsche.

Alle Personen- und Rollenbezeichnungen im Maskulinum beziehen sich in gleicher Weise auf Frauen und Männer.

Alle in dieser Arbeit angegebenen Internetadressen wurden zuletzt am 23. April 2010 auf Erreichbarkeit überprüft.

Der Text wurde unter Eclipse 3 mit TEXlipse- und CVS-Plugins erstellt und mit LaTeX und KOMA-Script gesetzt.

Teil I

Praxis der global verteilten Softwareentwicklung

2 Global verteilte Softwareentwicklung: ein Überblick

Dieses Kapitel gibt einen Überblick über die wichtigsten Konzepte der global verteilten Softwareentwicklung. Es legt mit Definitionen die Grundlage für die weitere Arbeit. Das Offshoring der Anwendungsentwicklung ist ein bedeutender Fall von verteilter Softwareentwicklung. Daher werden in diesem Kapitel Outsourcing und Offshoring im Detail behandelt. Es wird diskutiert, warum Offshoring für die Softwareentwicklung interessant ist und was sich Unternehmen vom Auslagern ihrer Anwendungsentwicklung versprechen.

In Abschnitt 2.1 werden verschiedene Ausprägungen von verteilter Entwicklung vorgestellt. Im darauf folgenden Abschnitt 2.2 geht es um die Herkunft des Begriffes Offshoring, die Beziehung zum Outsourcing und die Besonderheiten des Offshoring. Abschnitt 2.3 beschäftigt sich mit Erwartungen, die mit verteilter Anwendungsentwicklung im Allgemeinen verbunden sind. Diesen Erwartungen werden Erfahrungen und Probleme aus der Praxis gegenübergestellt. Daraus werden in Abschnitt 2.4 wissenschaftliche Fragestellungen im Allgemeinen und für die Softwaretechnik im Speziellen abgeleitet, die als Ausgangspunkt für die weiteren Untersuchungen in dieser Arbeit dienen.

2.1 Ausprägungen von global verteilter Entwicklung

Bei der verteilten Softwareentwicklung befinden sich nicht alle Entwickler am selben Ort. Schon die Arbeit in verschiedenen Gebäuden, in verschiedenen Stockwerken oder auch nur in verschiedenen Räumen kann als verteilte Entwicklung aufgefasst werden. Die Definition von global verteilter Softwareentwicklung geht noch darüber hinaus (in Anlehnung an das Global Software Development Handbook [Sangwan et al. 2006]):

Definition Global verteilte Softwareentwicklung:

Global verteilte Softwareentwicklung (englisch Global Software Development, GSD) ist eine besondere Form der Softwareentwicklung, bei der sich die an der Entwicklung Beteiligten an verschiedenen Standorten befinden, die durch erhebliche räumliche Entfernungen getrennt sind.

Des Öfteren findet man dafür auch die Bezeichnungen **Global Software Engineering** und **Globally Distributed System Development**. Sie werden meist synonym gebraucht.

Neben diesen Begriffen gibt es weitere, die unterschiedliche Ausprägungen von verteilter Entwicklung bezeichnen. Bei der Kooperativen Softwareentwicklung findet die Entwicklung mit mehreren Beteiligten an verschiedenen Orten statt. Altmann und Pomberger definieren den Begriff in [Altmann u. Pomberger 1999, S. 652] wie folgt: „**Kooperative Softwareentwicklung** umfaßt die Abdeckung der Kommunikations- und Koordinationsbedarfe innerhalb eines Softwareentwicklungsprozesses, die für die Planung, Durchführung und Abstimmung aller aufgabenbezogenen, zeitlich und räumlich verteilten Aktivitäten erforderlich sind. Kooperative Softwareentwicklung umfaßt dementsprechend alle prozeß- und produktbezogenen Aktivitäten aller Beteiligten, deren gemeinsames Ziel die Erstellung eines Softwareproduktes ist.“ Untersuchte Fragestellungen im Bereich der Kooperativen Softwareentwicklung sind die Kommunikation, Koordination und die Kooperation bei verteilter Teamarbeit. Diese Aspekte sind bei jeder Art von verteilter Entwicklung sehr wichtig. Daher greifen wir sie in Abschnitt 4.1 wieder auf.

Bei der oft synonym zu Kooperativer Softwareentwicklung verwendeten **Kollaborativen Softwareentwicklung** (KSE oder CSD, von engl. Collaborative Software Development, oder CSE, von engl. Collaborative Software Engineering [Mistrík et al. 2010]) sind die Akteure organisatorisch, zeitlich und räumlich verteilt. Sie müssen trotzdem bei der Entwicklung einer Software eng zusammenarbeiten und intensiv untereinander Wissen austauschen [Hildenbrand et al. 2007]. Im englischsprachigen Raum benutzt man für solche Projekte auch häufig die Bezeichnung **Distributed (Software) Development**. Möchte man betonen, dass die Entwickler räumlich auf mehrere Standorte verteilt sind, so spricht man von **Multi-Site Development** [Sengupta et al. 2006].

Beim **Dispersed Development** arbeitet jeder Entwickler einzeln, räumlich von den anderen getrennt. Dies ermöglicht den Mitarbeitern z. B. die häusliche Kinderbetreuung im Sinne der Telearbeit. Dieses Modell ist nur begrenzt für größere Entwicklungsvorhaben geeignet [Daniels u. Dyson 2004]. Es ist auch bezeichnend für viele Open-Source-Projekte. Nach Alistair Cockburn ist typisches **Open-Source-Development** mit den meisten anderen Softwareentwicklungsprojekten nicht vergleichbar, obwohl auch hier mehrere Entwickler kooperieren. Es unterscheidet sich u. a. durch kaum vorhandenen Marktbezug, eine prinzipiell unbegrenzte Anzahl Entwickler mit einer starken intrinsischen Motivation (Mitarbeit zum Vergnügen, nicht zum Profit) und eine besondere Teamstruktur. Diese besteht aus einem Kernteam, das die Quelltextbasis kontrolliert, und mehreren dazu mit neuen Funktionen, Fehlermeldungen und -behebungen und Dokumentation beitragenden Entwicklern [Cockburn 2002].

Allgemein um den Einsatz von Technologien zur Unterstützung der verteilten Gruppenarbeit geht es bei dem interdisziplinären Themengebiet **Computer-Supported Cooperative Work** (CSCW). Bei der Gruppenarbeit wirken mehrere Menschen am

selben Arbeitsgegenstand zusammen. Im Themengebiet CSCW werden Erkenntnisse aus der Informatik, Psychologie, Soziologie, den Arbeits-, Organisations- und Wirtschaftswissenschaften und weiteren Disziplinen zusammengetragen. Ein Ergebnis der Forschung auf diesem Gebiet ist die Entwicklung von Groupware-Systemen – Software, Hardware und Services, die Gruppen unterstützen, z. B. Wikis, Gruppenkalender und Blogs [Gross u. Koch 2007].

Agile Ansätze zur verteilten Entwicklung nehmen in dieser Arbeit eine wichtige Rolle ein. Sie werden in Kapitel 3 ausführlich behandelt.

2.2 Offshoring: Historie und Definition

Verteilte Entwicklung erfolgt häufig im Rahmen des Offshoring. Umgangssprachlich wird der Begriff Offshoring oft für jede Form von global verteilter Entwicklung verwendet. Bei genauerer Betrachtung ist Offshoring jedoch nur eine mögliche Organisationsstruktur für global verteilte Entwicklung, welche die Beteiligung mehrerer Unternehmen am Entwicklungsvorhaben berücksichtigt.

Im nächsten Abschnitt werden grundlegende Begriffe im Umfeld des Offshoring diskutiert und für diese Arbeit definiert. Im darauf folgenden Abschnitt werden strategische Gestaltungsoptionen beim IT-Sourcing betrachtet. Anschließend wird das zu untersuchende Gebiet eingegrenzt.

2.2.1 Begriffsbestimmung „Offshoring“

Beim Offshoring handelt es sich um eine besondere Form des Outsourcings, das daher zuerst betrachtet wird.

Outsourcing

Der Begriff **Outsourcing** fand im 1996 erschienenen Ergänzungsband erste Aufnahme in die Brockhaus Enzyklopädie mit folgender Definition [Brockhaus 1996, S. 559]: „Kw. aus engl. *out*side (außen, außerhalb), re*sourc*e (Hilfsmittel) und *-ing* [in anderen Quellen: us*ing*]. Bez. für den Übergang von der eigenen betriebl. Leistungserstellung zum Fremdbezug mit der primären Absicht, die Kosten zu senken.“ Der englische Begriff wurde ins Deutsche übernommen. Im aktuellen Rechtschreib-Duden findet sich folgende Definition [Duden 2009, S. 755]: „Out|sour|cing, das, -s <engl.> (*Wirtsch.* Übergabe von bestimmten Firmenbereichen an spezialisierte Dienstleistungsunternehmen)“ mit den Verbformen out|sour|cen, etw. wird outgesourct. In der Fachliteratur finden sich weitere Definitionen, u. a. bei [Ruiz Ben u. Claus 2005], [Schwarze u. Müller 2005], [Cheon et al. 1995], [Bruch 1998].

Der allgemeine Outsourcing-Begriff umfasst nicht nur die Auslagerung der Informationstechnik (IT), sondern auch die Auslagerung weiterer Felder, wie z. B. des Fuhrparks oder der Betriebskantine. Wenn explizit die IT gemeint ist, spricht man von **IT-Outsourcing**. Cullen und Willcocks heben hervor, dass sich IT-Outsourcing in mehreren Punkten von anderen Formen des Outsourcings wesentlich unterscheidet [Cullen u. Willcocks 2003]:

1. Die Aufgaben der IT sind vielfältig und heterogen.

2. Die Leistungsfähigkeit der IT entwickelt sich in hohem Tempo weiter. Zukünftige Aufgaben und Anforderungen können nicht genau vorhergesagt werden.

3. Die Wirtschaftlichkeit der IT kann nicht genau gemessen werden. Daher fehlt die Grundlage, um die mit einer Auslagerung einsparbaren Kosten zu beziffern.

4. Bei der IT lassen sich Kostenvorteile schlecht durch Massenproduktion erreichen. Viel wichtiger sind geeignete Prozesse und Techniken.

5. Das bedeutendste Unterscheidungsmerkmal ist, dass mit der Auslagerung der IT in der Regel große Umstellungskosten verbunden sind.

Das IT-Outsourcing hoher Auftragsvolumina begann 1963 mit einem Vertrag zwischen EDS und dem Blue Cross of Pennsylvania über die Auslagerung der Datenverarbeitung. Im Laufe der 70er Jahre konnte EDS weitere bedeutende Unternehmen als Kunden gewinnen, u. a. Continental Airlines und Enron. Dies führte zu einer größeren Akzeptanz von Outsourcing. Der nächste große Schritt erfolgte Ende der 80er Jahre, als IBM mit seinem International Shared Service Center ins Outsourcing-Geschäft einstieg und Kodak als ersten Kunden gewinnen konnte. Dieser Vertrag, der mit 1 Mrd. US$ notiert war, erregte viel Aufsehen. In der Folge wurden viele weitere Outsourcing-Verträge geschlossen, nicht nur in den USA, sondern beispielsweise auch von der Deutschen Lufthansa [Dibbern et al. 2004].

Im Jahr 2008 lag der Gesamtumsatz mit IT- und Business Process Outsourcing in Deutschland bei 13,6 Mrd. €, nach einer Steigerung von 7,4% im Vergleich zum Vorjahr. Trotz der wirtschaftlich schwierigen Situation erwartet der Bundesverband BITKOM auch für 2009 ein ähnliches Wachstum von 7,2% auf dann 14,6 Mrd. €. Wegen der Krise würden auch Unternehmen über Outsourcing nachdenken, die bisher noch Vorbehalte hätten. Unternehmen würden zudem schneller über Outsourcing-Projekte entscheiden als vorher [BITKOM 2009].

Wir definieren den Begriff IT-Outsourcing in Anlehnung an die Definition der BITKOM [Weber 2006, S. 87] wie folgt:

Definition IT-Outsourcing:

„Information Technology (IT) Outsourcing ist die vollverantwortliche Übertragung von IT-Funktionen oder Geschäftsprozessen mit hohem IT-Anteil [von einem Klienten] an rechtlich selbständige – d. h. externe – Dienstleister über einen definierten Zeitraum."

Im Gegensatz zu einigen anderen Definitionen muss die übertragene Leistung vorher nicht im eigenen Unternehmen erbracht worden sein. Die Definition fordert die Vergabe der Leistungen an rechtlich selbstständige Unternehmen („Auslagerung"). Bei anderen Definitionen wird auch ein Leistungsbezug von kapitalmäßig verbundenen Unternehmungen eingeschlossen („Ausgliederung"), vgl. z. B. [Bliesener 1994]. Ersteres wird auch als Outsourcing im engeren Sinne, Letzteres als Outsourcing im weiteren Sinne verstanden [Bruch 1998].

Aufgrund dieser Definition ist das Outsourcing eine spezielle Variante einer Make-or-Buy-Entscheidung. Normativ wird für Outsourcing-Entscheidungen gefordert, dass es sich um Entscheidungen mit strategischer Bedeutung handelt, dass mittel- und längerfristige Kooperationen angestrebt werden und dass die outgesourcten Funktionen vorher besonders umfassend analysiert wurden. Dies steht im Gegensatz zum spontanen Fremdbezug mit keiner oder einer kurzfristigen Bindungswirkung.

Der Klient wird je nach Kontext auch als Auftraggeber oder Servicenehmer bezeichnet. Der Dienstleister entsprechend auch als Auftragnehmer oder Servicegeber.

Offshoring als Offshore Outsourcing

Während beim Outsourcing die Frage beantwortet wird, *wer* eine Dienstleistung erbringt, steht beim Offshoring der geografische Aspekt im Mittelpunkt; es ist wichtig, *wo* die Dienstleistung erbracht wird.

Im aktuellen Rechtschreib-Duden von 2009 [Duden 2009] sind die Begriffe **Offshoring** oder **offshore** nicht aufgeführt. Die Brockhaus Enzyklopädie kennt zwar den Begriff offshore, jedoch wird er definiert als „(engl.) offshore = in einiger Entfernung von der Küste" [Brockhaus 2002] und im Kontext der Gewinnung von Erdöl und Erdgas aus dem Meeresboden verwendet. Offshoring selbst ist auch dort noch nicht als eigenständiger Begriff enthalten. Da eine Übersetzung ins Deutsche, beispielsweise „Verlagerung des Standorts ins Ausland" sehr sperrig ist und sich in der deutschsprachigen Literatur schon die englischen Begriffe durchgesetzt haben, werden sie auch in dieser Arbeit verwendet.

Wir definieren IT-Offshoring wie folgt:

Definition IT-Offshoring:

IT-Offshoring ist eine besondere Form des IT-Outsourcings, bei der IT-Funktionen oder Geschäftsprozesse mit hohem IT-Anteil an Unternehmen in geografisch entfernte Niedriglohnländer vergeben werden.

Mit dieser Definition ist IT-Outsourcing ein Oberbegriff, der IT-Offshoring beinhaltet. Als Abgrenzungsmerkmal dient die geografische Entfernung. Nach einer Untersuchung von Definitionen des Begriffs Offshoring in 25 für ein Journal eingereichten Artikeln im Bereich Offshoring von Informationssystemen, unterscheidet sich Offshoring vom Outsourcing hauptsächlich darin, dass die Diensterbringung außerhalb des Heimatlandes des Kunden erfolgt [King u. Torkzadeh 2008]. In den meisten Fällen findet sie sogar auf einem anderen Kontinent statt.

Das unterschiedliche Lohnniveau begründet hauptsächlich, warum IT-Offshoring praktiziert wird. Zusätzliche Gründe werden in Abschnitt 2.3.2 diskutiert. Weitere Definitionen von IT-Offshoring lassen sich z. B. bei [Aspray et al. 2006], [Weber 2006] und [Schaaf 2004] nachlesen. Eine kurze Historie des IT-Offshoring findet sich bei Wiener [Wiener 2006].

Einige Definitionen von Offshoring fordern, dass „sich die benötigten Ressourcen oder das zur Projektbearbeitung notwendige Personal vollständig am Standort des Offshore-Dienstleisters [befinden]“ [Nicklisch 2006, S. 48]. In der hier verwendeten Definition wird diese Einschränkung nicht vorgenommen. Teile der Dienstleistung können (und sollten in vielen Fällen) in räumlicher Nähe zum Auftraggeber erbracht werden.

Wenn in dieser Arbeit in der Folge die Begriffe Outsourcing und Offshoring verwendet werden, sind immer das IT-Outsourcing bzw. das IT-Offshoring gemeint.

Zahlen zum Offshoring

Offshoring ist in Deutschland nicht so verbreitet, wie man aufgrund der recht umfangreichen Diskussion in den Medien vermuten könnte. Nach einer Schätzung der IDC wurden im Jahr 2006 weltweit insgesamt 14 Mrd. US$ für das Offshoring klassischer IT-Dienstleistungen ausgegeben. Davon entfallen 11 Mrd. US$ auf die USA und nur 2,5 Mrd. US$ auf Westeuropa. Die deutschsprachigen Länder kommen nur auf einen Anteil von 9% an den europäischen Ausgaben [Mason et al. 2005]. Diese Zahlen werden durch Umfragen untermauert. In [Schaaf u. Weber 2005] wird deutlich, dass nur wenige Projekte von Deutschland aus verlagert werden und dass deren Umfang gering ist. Knapp die Hälfte der auslagernden Unternehmen wendet weniger als 1 Mio. € dafür auf.

Es ist schwierig, belastbarere Zahlen und Tendenzen zum Umfang des Offshoring in Deutschland zu finden. Nach Angaben des Statistischen Bundesamts gibt es generell

für die „Verlagerung wirtschaftlicher Aktivitäten deutscher Unternehmen ins Ausland“ [Statistisches Bundesamt 2009, S. 15] bisher keine verlässlichen Zahlen. Stattdessen wurde mit Einzelfallbeispielen argumentiert. Dies wird sich in Zukunft ändern, da die EU in der FATS-Verordnung vom Juni 2007 (FATS steht für Foreign AffiliaTes Statistics) die Erhebung solcher Daten vorschreibt [Statistisches Bundesamt 2009].

Ein interessantes Thema betrifft die Auswirkungen des Offshoring auf Arbeitsplätze in den auslagernden Ländern. Die Auswirkungen hängen wesentlich davon ab, wie flexibel die Arbeitsmärkte sind. Hier gibt es bedeutende Unterschiede. Im Vergleich zu den USA steht Deutschland schlechter da (siehe z. B. die Analyse von McKinsey [Farrell 2004] und die Entgegnung von Peter Mertens [Mertens 2005]). Eine Prognose von A. T. Kearney aus dem Jahr 2007 zeichnet ein düsteres Bild für Deutschland: „In fünf Jahren fallen durchschnittlich 80% der IT-Service-Ausgaben auf externe Dienstleister und es werden 120.000 Arbeitsplätze abgebaut. 30% der existierenden Stellen werden in Niedriglohnländer verlagert.“ [Röder et al. 2007]

Verwandte Begriffe

In der Fachliteratur und in dieser Arbeit werden weitere Begriffe im Zusammenhang mit Outsourcing, Offshoring und global verteilter Entwicklung gebraucht:

Die Leistungserbringung findet **on-site** statt, wenn sie direkt oder sehr nahe beim Auftraggeber erfolgt (auf dessen Firmengelände oder zumindest in derselben Stadt oder Region); das Gegenteil wird als **off-site** bezeichnet. Die Leistungserbringung findet **onshore** statt, wenn sie in geografischer Nähe des Auftraggebers erfolgt (typischerweise auf demselben Kontinent), ansonsten – analog zur Definition von Offshoring – **offshore**. Diese Begriffe lassen sich kombinieren: Onshore on-site bedeutet Leistungserstellung beim Auftraggeber. Onshore off-site kann z. B. die Leistungserstellung in den Räumen des Auftragnehmers im gleichen Land bedeuten. Offshore off-site steht für Leistungserstellung auf einem anderen Kontinent. Offshore on-site ist ein Widerspruch. Die Personen, die onshore arbeiten, werden als **Onshore-Team** bezeichnet, die offshore arbeitenden Mitarbeiter entsprechend als **Offshore-Team**. Diese Bezeichnung erfolgt unabhängig davon, ob das Personal zum Auftragnehmer oder -geber gehört und welche Aufgaben es wahrnimmt. Die Rollen der Teams werden in Kapitel 4 detailliert diskutiert.

Beim **Insourcing** wird eine Leistung explizit durch einen Auftragnehmer innerhalb desselben Unternehmens erbracht, eventuell in einer rechtlich selbstständigen Tochtergesellschaft. Insourcing kann eine Folge von **Backsourcing** sein, bei dem eine vorher outgesourcte Leistung wieder ins eigene Unternehmen zurückverlagert wird. Beim **Nearshoring** werden Ressourcen im benachbarten Ausland auf demselben Kontinent genutzt, von Deutschland aus gesehen beispielsweise in Europa. Nearshore-Länder weisen meist eine dem Ausgangsland ähnliche Kultur auf (vgl. die Begriffserläuterungen in [Weber 2006]). Nearshoring wird aufgrund ähnlicher Eigenschaften in der

Literatur oft unter Offshoring subsumiert. In dieser Arbeit verwenden wir Offshoring ebenfalls als Oberbegriff und differenzieren die Begriffe nur an den Stellen explizit, an denen es bemerkenswerte Unterschiede gibt. Multinationale Konzerne sprechen oft von **Global Sourcing** und meinen damit die kontextabhängige Nutzung verschiedener Sourcing-Varianten zur Ausnutzung und Kombination verschiedener nationaler Standortvorteile.

Neben den hier eingeführten Begriffen werden gerade in der Praxis viele weitere im Umfeld des Outsourcings benutzt. Oft werden diese nicht oder nur unscharf definiert, nicht klar voneinander abgegrenzt und anscheinend hauptsächlich aus vertriebsorientierten Überlegungen verwendet. Daher beschränken wir uns im Folgenden auf die hier eingeführten Begriffe.

2.2.2 Dimensionen des IT-Sourcings

Nach der Betrachtung von Offshoring als Offshore Outsourcing werden hier allgemeinere Sourcing-Modelle vorgestellt. Unter **Sourcing** verstehen wir in dieser Arbeit wie in [Weber 2006, S. 95] den „Prozess der Beschaffung von unternehmensinternen oder -externen Ressourcen“. Wenn es sich um Ressourcen der Informationstechnologie handelt, spricht man von **IT-Sourcing**. Es lassen sich zwei Sichten auf das Sourcing unterscheiden. Bei der statischen Sicht geht es um die Beschreibung der gegenwärtigen organisatorischen Gestaltung des Ressourcenbezugs. Bei der dynamischen Sicht geht es um den Entscheidungsprozess und die Wahl zwischen alternativen Gestaltungsmöglichkeiten [Dibbern 2004]. Die Entwicklung einer geeigneten Sourcing-Strategie, d. h. insbesondere die Entscheidung für In- oder Outsourcing von Ressourcen, kann bedeutende Auswirkungen für ein Unternehmen haben.

Nachfolgend wird zunächst eine zweidimensionale Matrix mit den wichtigsten Optionen des IT-Sourcings behandelt. Im Anschluss werden weitere Dimensionen in einer Übersichtskarte des IT-Sourcings vorgestellt.

IT-Sourcing Matrix

Abbildung 2.1 zeigt zwei grundlegende Dimensionen zu Sourcing-Entscheidungen und deren Geschäftsmodelle und ggf. zu erwartende Auswirkungen auf Arbeitsplätze. Die erste Dimension betrifft die unternehmensinterne (Insourcing) oder -externe (Outsourcing) Beschaffung von Ressourcen. Die zweite Dimension gibt den Standort der Leistungserstellung an: onshore oder offshore.

Der Fall **Onshore Insourcing** beschreibt die herkömmlichen internen IT-Einheiten, die alle Anwendungen und IT-Systeme selber erstellen bzw. beschaffen und warten. Multinationale Firmen betreiben oft mehrere regionale Außenstellen in verschiedenen Ländern mit niedrigerem Lohnniveau. Dieses sogenannte **Offshore Insourcing** stellt

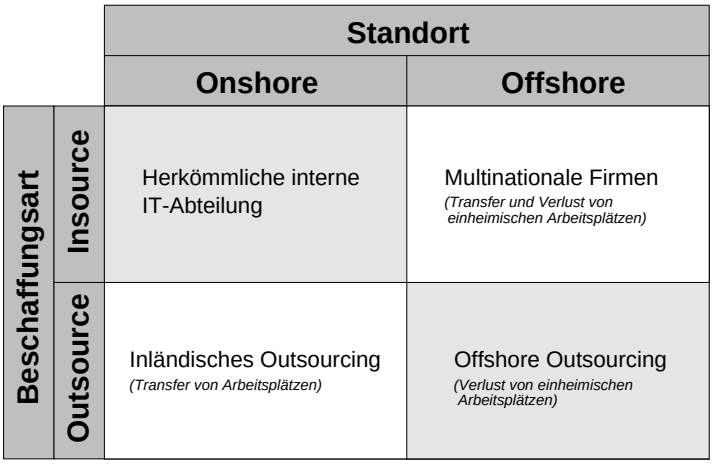


Abbildung 2.1: *Offshore Outsourcing-Matrix, angelehnt an [B. M. Shao u. Smith David 2007, S. 90]*

einen häufigen Fall der global verteilten Entwicklung dar, bei dem nicht mehrere unabhängige Unternehmen beteiligt sind.

Wenn die Leistungserstellung outgesourct wird, ergeben sich zwei Optionen. Beim **Onshore Outsourcing** (meist verkürzt zu Outsourcing) werden die Arbeiten einem inländischen Dienstleister übergeben. Arbeitsplätze werden verlegt. Beim **Offshore Outsourcing** (kurz: Offshoring) werden Arbeiten ins entfernte Ausland verlagert.

Übersichtskarte des IT-Sourcings

Eine weitere Detaillierung der Sourcing-Optionen erhält man durch die Betrachtung zusätzlicher Dimensionen. Eine oft zitierte und verbreitete Darstellung von Jouanne-Diedrich, die IT-Sourcing-Map, findet sich in [Jouanne-Diedrich 2004]. Eine aktualisierte Version dieser Übersichtskarte kann unter [Jouanne-Diedrich 2007] eingesehen werden. Wir verwenden hier eine etwas vereinfachte und angepasste Darstellung, siehe Abbildung 2.2.

Die Dimensionen der IT-Sourcing-Map erleichtern den Vergleich verschiedener Outsourcing-Unterfangen. Bei den empirischen Untersuchungen in den Abschnitten 5.3 und 6.2 wird die IT-Sourcing-Map daher genutzt, um einen Überblick über die Rahmenbedingungen der jeweiligen Feldstudie zu geben.

Die im vorangegangenen Abschnitt betrachteten Dimensionen finden sich auch in diesem Modell: Beim **Grad des externen Leistungsbezugs** geht es um die Frage, welcher Anteil der IT-Leistungen outgesourct wird. Die Spanne reicht vom Totalen Insourcing über das Selektive Sourcing (nach Lacity und Willcocks bei 20–80% Fremdbezug [Lacity u. Willcocks 2001]) zum Totalen Outsourcing. Bei der Auswahl des **Standorts der**

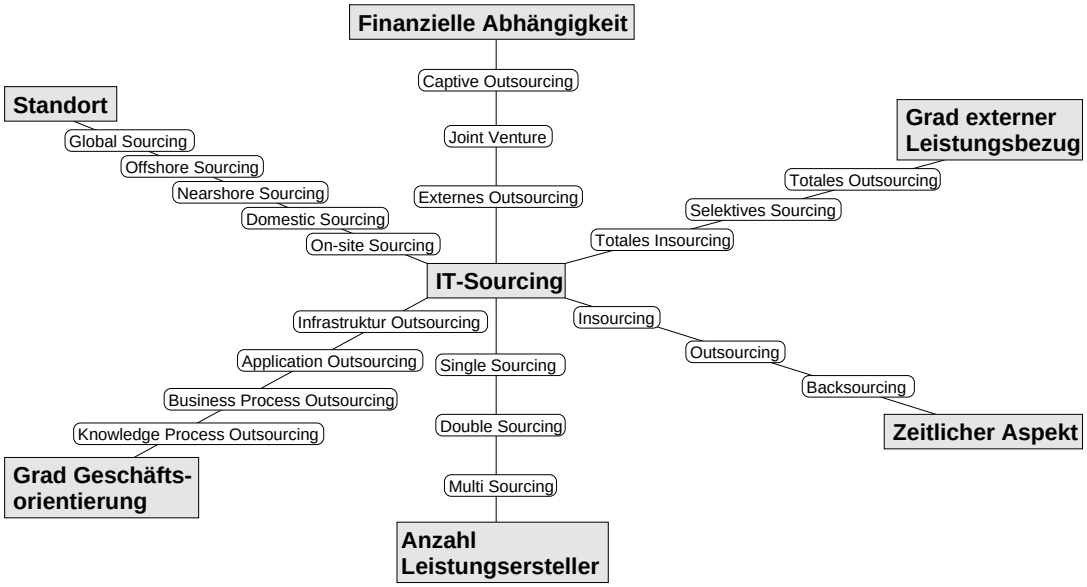


Abbildung 2.2: *IT-Sourcing-Map, angelehnt an [Jouanne-Diedrich 2007]*

Leistungserstellung bieten sich Wahlmöglichkeiten von direkt beim Auftraggeber (On-site Sourcing) über das Inland (Domestic Sourcing) und benachbarte Ausland (Nearshore Sourcing) hin zum Offshore und Global Sourcing.

Hinzu kommen weitere Dimensionen:

Anzahl der Leistungsersteller: Beim Single Sourcing werden alle Leistungen aus einer Hand bezogen. Beim Double Sourcing sind zwei Leistungsersteller (Dienstleister) beteiligt, beim Multi Sourcing mehrere.

Zeitlicher Aspekt: In der Reihenfolge der Sourcing-Entscheidungen lassen sich drei Stufen unterscheiden: vom Insourcing (Leistungserstellung im eigenen Unternehmen) über das Outsourcing (Fremdvergabe der Leistungserstellung) zum Backsourcing (eine zuvor outgesourcte Leistung wird wieder im eigenen Unternehmen erbracht). Diese Dimension spiegelt ein enges Begriffsverständnis des Outsourcings wider, bei dem eine Leistung vorher zwingend selbst erbracht werden musste (vgl. die Diskussion auf Seite 17).

Finanzielle Abhängigkeit: Wenn die Leistungserstellung von einem rechtlich selbstständigen externen Unternehmen erbracht wird, spricht man vom Externen Outsourcing. Bei einem Joint Venture gründen Leistungsnehmer und Leistungsersteller eine gemeinsame Firma zum Zweck der Leistungserstellung. Wenn die Leistungserstellung konzernintern erbracht wird, handelt es sich um Captive Outsourcing.

Grad der Geschäftsorientierung: Unternehmen können unterschiedliche Aufgaben verlagern. Die Spanne reicht vom Outsourcing der Infrastruktur über die Entwicklung von Anwendungen (Application Outsourcing) bis hin zur Abgabe kompletter Geschäftsprozesse, die nicht zum Kerngeschäft gehören (Business Process Outsourcing). Auch wissensorientierte, komplexe Geschäftsprozesse, wie z. B. Forschung & Entwicklung, können dazugehören. Dies bezeichnet man als Knowledge Prozess Outsourcing.

In der Praxis sind viele Ausprägungen dieser Dimensionen anzutreffen. Umfang und Art des Sourcings sind abhängig von den jeweiligen Unternehmenszielen und -strategien. Gerade beim Global Sourcing gibt es keinen Sourcing-Ansatz, der für alle Unternehmen passend ist [Hazra 2006].

2.2.3 Offshoring der Anwendungsentwicklung

Das IT-Outsourcing kann viele Bereiche der Informationstechnik in Unternehmen betreffen. Anhand der Dimension „Grad der Geschäftsorientierung“ lassen sich verschiedene Leistungsebenen unterscheiden, üblicherweise Infrastruktur, Anwendungen und Geschäftsprozesse [Weber 2006]:

Das **Infrastructure Outsourcing** umfasst Betrieb und Wartung von (Teilen der) IT-Infrastruktur sowie Support-Dienstleistungen. Wichtige Felder sind Rechenzentren (Data Center Outsourcing), Netze (Network Outsourcing), Arbeitsplatzsysteme (Desktop Outsourcing), Endbenutzer Support (User Help Desk) und Sicherheit (Security Outsourcing). Beim **Application Outsourcing** kann der gesamte Lebenszyklus von Anwendungen abgedeckt werden: Softwareentwurf, Implementierung und Test; Betrieb, Support und Wartung; Erweiterung und Migration. Im Rahmen des **Business Process Outsourcing** verlagerbare Geschäftsprozesse sind z. B. Personalwesen, Beschaffung, Finanzwesen und Buchhaltung, Logistik, Ausbildung und Training sowie Forschung & Entwicklung (Knowledge Process Outsourcing). Meist wird dabei auch die unterstützende IT-Infrastruktur mit ausgelagert.

Alle diese Aufgaben eignen sich potenziell auch für das Offshoring. Nicht nur in Deutschland hat daran die Anwendungsentwicklung den größten Anteil. Wie Abbildung 2.3 zeigt, gaben bei einer Umfrage unter 570 Unternehmen im deutschsprachigen Raum im Jahr 2005 über die Hälfte der befragten Unternehmen an, die Anwendungsentwicklung zumindest teilweise an Offshore-Partner vergeben zu haben [Schaaf u. Weber 2005]. Weitere 30% der Unternehmen planten dies. Zu vergleichbaren Ergebnissen kamen Dibbern und Heinzl schon 2001 bei einer Umfrage unter kleinen und mittleren Unternehmen in Süddeutschland [Dibbern u. Heinzl 2001]. In dieser Arbeit konzentrieren wir uns auf die Entwicklung von Anwendungssoftware.

Es gibt eine Reihe von Eigenschaften, die gut und wirtschaftlich sinnvoll verlagerbare Prozesse kennzeichnen [Boos et al. 2005]:

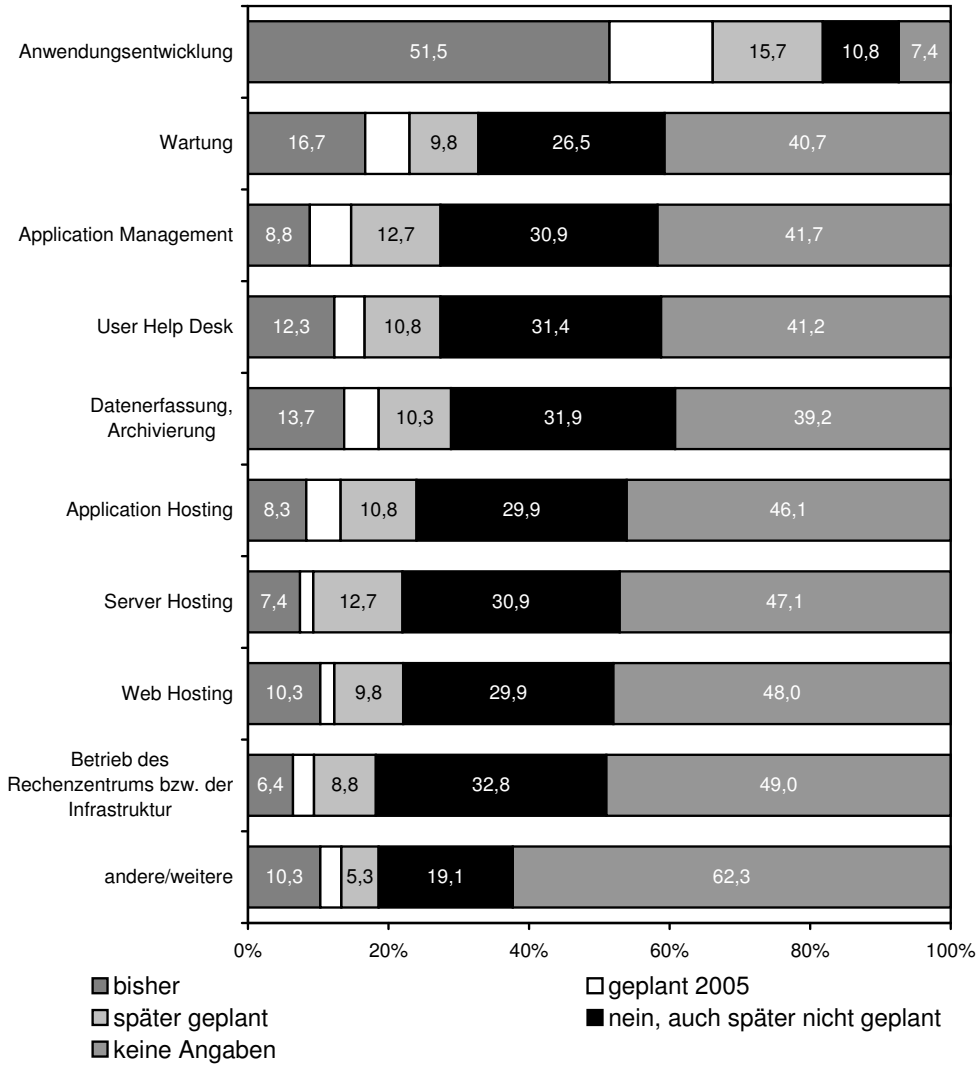


Abbildung 2.3: *Für Offshoring geeignete Geschäftsprozesse, nach [Schaaf u. Weber 2005, S. 14]: „Welche Geschäftsprozesse aus den Bereichen IT & TK werden von Ihrem Unternehmen zumindest teilweise an Offshore-Partner vergeben?“*

- Es handelt sich um abgeschlossene, gut definierbare und überprüfbare Aufgaben.
- Zur Durchführung ist erheblicher Personalaufwand nötig.
- Sie werden häufig durchgeführt (bei wiederkehrenden Aufgaben) bzw. haben einen größeren Umfang (bei Projekten).
- Im eigenen Unternehmen sind sie nur unbefriedigend realisierbar.
- Es existieren keine komplexen Wechselwirkungen mit anderen Bereichen.
- Das Risiko ist überschaubar. Bei Störung oder Ausfall ist der Unternehmenserfolg nicht gefährdet.

Da die Entwicklung von Anwendungssoftware diese Anforderungen häufig erfüllt, scheint eine Auslagerung prinzipiell gut möglich zu sein. In der Praxis ergeben sich jedoch Probleme, deren Diskussion und Bewältigung mit ausgewählten Konzepten und Methoden im Mittelpunkt dieser Arbeit stehen.

Das *Outsourcing* der Anwendungsentwicklung bezeichnet man als **Application Development Outsourcing (ADO)**. Dies ist nicht mit Application Service Providing (ASP) zu verwechseln. Bei diesem werden Softwareanwendungen über eine Online-Verbindung bereitgestellt [Küchler 2004]. Für diese Arbeit ist das *Offshoring* der Anwendungsentwicklung wichtiger, bei dem ein entfernt arbeitendes Team von Entwicklern eingebunden wird. Bei den Beteiligten kann es sich um gleichberechtigte Teams innerhalb eines Unternehmens handeln (Offshore Insourcing, siehe 2.2.2) oder um ein Onshore-Team beim Auftraggeber und ein Offshore-Team eines externen Dienstleisters, das vorgegebene Anforderungen umsetzt. Bei letzterer Organisationsform spricht man auch von **Offshore Software Development (OSD)** [Wiener 2006].

Die Art der Leistungen des Offshoring-Dienstleisters beim Offshoring der Anwendungsentwicklung ist abhängig vom Softwaretyp. Bei Standardsoftware kann der Dienstleister in die Auswahl des Produkts einbezogen werden und individuelle Anpassungen (z. B. durch Parametrisierung und Konfiguration) und Erweiterungen durch Neuprogrammierung vornehmen. Bei Individualsoftware können die komplette Neuentwicklung oder Teile davon an den Dienstleister abgegeben werden [Küchler 2004].

Es lassen sich zwei Varianten der Aufgabenteilung betrachten. Abbas, Kazmierczak, Dart und O'Brien unterscheiden die Aufteilung nach Komponenten einer Anwendung (Application Component Outsourcing) von der Aufteilung nach den Phasen des Entwicklungsprozesses (Outsourcing Development Phases) [Abbas et al. 1998b]. Bei Ersterem gibt der Auftraggeber Anforderungen und Schnittstellen für einzelne Komponenten vor. Der Dienstleister muss diese Anforderungen erfüllen und die Schnittstellen einhalten. Er hat jedoch in der Regel Wahlfreiheit beim Vorgehen und seiner Entwicklungsmethodik. Zum Komponentenbegriff in der Softwaretechnik siehe [Szyperski 2002].

Bei der phasenweisen Aufteilung werden eine oder mehrere Phasen des Entwicklungsprozesses vom Dienstleister übernommen. Nach einer 2006 unter deutschen Unternehmen

durchgeführten Umfrage wird dabei die Realisierungsphase am häufigsten übertragen [Prieß 2006]. Mehr als drei Viertel der Befragten können sich vorstellen, die Implementierung auszulagern. Das Outsourcing der Vorstudie ist für 77% ebenso kein Thema wie das des Fachkonzepts (79%) und der Integration (68%). Bei der Festlegung der IT-Architektur und bei der Testphase gibt es keine klaren Tendenzen.

2.2.4 Besonderheiten des Offshoring

Was macht das Offshoring der Anwendungsentwicklung zu etwas Besonderem? Viele der vorgestellten Ansätze im Umfeld des Outsourcings unterscheiden sich nur unwesentlich. Dies heißt auch, dass Ergebnisse aus verwandten Gebieten womöglich auf das Offshoring der Anwendungsentwicklung übertragen und in einem anderen Gebiet entwickelte Techniken dort eingesetzt werden können. Ein gutes Beispiel sind Groupwaresysteme aus dem Bereich CSCW, die auch zur Kommunikation zwischen verteilten Teams beim Offshoring der Anwendungsentwicklung verwendet werden können.

Aus dem bisher Geschriebenen lassen sich die folgenden besonderen Eigenschaften des Offshoring der Anwendungsentwicklung ableiten:

Anwendungsentwicklung: Entwickelt werden Anwendungen, die meist schwierig im Voraus zu spezifizieren sind und daher Abstimmungen mit den Anwendern erfordern.

Outsourcing-Beziehung: Auftraggeber und Offshore-Dienstleister haben unterschiedliche Interessen sowie explizite und implizite Erwartungen.

Global verteilte Teams: Die beteiligten Teams sind geografisch verteilt und durch mehrere Zeitzonen getrennt. Sie müssen sich bei gemeinsamen Aufgaben abstimmen.

Kulturelle Unterschiede: Die Beteiligten kommen von verschiedenen Kontinenten und damit in der Regel auch aus unterschiedlichen Kulturen.

Aus diesen Eigenschaften ergeben sich direkt spezifische Herausforderungen auf verschiedenen Ebenen, insbesondere Outsourcing, Arbeitsteilung und Kommunikation in verteilten Teams und der Umgang mit kulturellen Unterschieden.

2.3 Erwartungen und Ergebnisse in der Praxis

In diesem Abschnitt beschäftigen wir uns mit der Frage, weshalb global verteilte Entwicklung so ein viel diskutiertes Thema geworden ist und welche Bedeutung es in der Praxis weltweit und in Deutschland besitzt. Wir untersuchen Erwartungen, die daran geknüpft werden und Erfahrungen aus der Praxis. Daran anschließend wird der Forschungsbedarf für die Softwaretechnik abgeleitet.

2.3.1 Offshoring weltweit und in Deutschland

Wie konnte sich Offshoring zu so einem bedeutenden Trend entwickeln? Kalakota und Robinson nennen in [Robinson u. Kalakota 2004] sieben Gründe für das schnelle Wachstum:

- anhaltender Kostendruck auf Unternehmen in den USA und Europa,
- rapider Rückgang der Kommunikations- und EDV-Kosten,
- dramatische Verbesserungen der Internetzuverlässigkeit und -funktionalität,
- mehr Offshore-Dienstleister mit größerem Leistungsvermögen,
- hochwertiges Angebot an Offshore-Dienstleistungen durch Onshore-Anbieter,
- Zugriff auf kostengünstige, gut ausgebildete Mitarbeiter, insbesondere für arbeitsintensive Aufgaben und
- durch Pioniere erfolgreich etablierte Offshoring-Geschäftsmodelle.

Deutsche Unternehmen greifen beim Outsourcing am häufigsten auf heimische Dienstleister zurück. Wenn ins Ausland verlagert wird, dann eher innerhalb Europas: nach Osteuropa (23% in EU-Länder, 15% in Nicht-EU-Länder) und nach Westeuropa (15%). Auf China und Indien entfallen nur 21% (nach einer Prognose von IDC für die Jahre 2005–2009 [Mason et al. 2005]).

Bei den Zielländern des Offshoring liegt Indien nach einer Übersicht von A. T. Kearney mit Abstand an der Spitze der attraktivsten Offshoring-Länder, siehe Tabelle 2.1 [A. T. Kearney 2007]. Mehrere der für Offshoring attraktiven Länder verfügen über gut ausgebildete Arbeitskräfte und ein gutes Geschäftsklima (nach der Studie z. B. Kanada, Singapur, Irland und Australien), mehrere bieten ein finanziell reizvolles Umfeld (z. B. Vietnam, Philippinen, Thailand und China). Indien ist jedoch das einzige Land, das beide Vorteile vereinen kann.

Auch weitere Untersuchungen bestätigen, dass Indien ein attraktives Ziel ist (vgl. z. B. [Aspray et al. 2006]). Beispielsweise verstehen es indische Firmen, bewährte Verfahren der Softwareentwicklung (insbesondere bei der Anforderungsspezifikation und bei Reviews und formaler Qualitätssicherung) mit flexibleren Techniken (insbesondere Iterationen und frühe Releases) zu verbinden [Cusumano et al. 2003]. Die NASSCOM (National Association of Software and Services Companies), die für Software zuständige indische Handelskammer, geht von einem jährlichen Wachstum der indischen IT- und BPO-Industrie von mehr als 25% aus. Es wird erwartet, dass im Jahr 2010 60 Mrd. von insgesamt 110 Mrd. US$ Exporterlöse im Offshoring-Bereich auf Indien fallen [NASSCOM 2005]. Im ersten Quartal 2006 wurden die Erwartungen mit einem Wachstum von etwa 33% gegenüber dem Vorjahr sogar übertroffen [Ribeiro 2006]. Dabei ist die Produktivität des Einzelnen in Indien deutlich geringer als in anderen Ländern. Nach einer im Jahr 2007 durchgeführten Studie der Economist Intelligence

Rank	Country	Financial attractiveness	People and skills availability	Business environment	Total score
1	India	3.22	2.34	1.44	7.00
2	China	2.93	2.25	1.38	6.56
3	Malaysia	2.84	1.26	2.02	6.12
4	Thailand	3.19	1.21	1.62	6.02
5	Brazil	2.64	1.78	1.47	5.89
6	Indonesia	3.29	1.47	1.06	5.82
7	Chile	2.65	1.18	1.93	5.76
8	Philippines	3.26	1.23	1.26	5.75
9	Bulgaria	3.16	1.04	1.56	5.75
10	Mexico	2.63	1.49	1.61	5.73
11	Singapore	1.65	1.51	2.53	5.68
12	Slovakia	2.79	1.04	1.79	5.62
13	Egypt	3.22	1.14	1.25	5.61
14	Jordan	3.09	0.98	1.54	5.60
15	Estonia	2.44	0.96	2.20	5.60
16	Czech Republic	2.43	1.10	2.05	5.57
17	Latvia	2.64	0.91	2.00	5.56
18	Poland	2.59	1.17	1.79	5.54
19	Vietnam	3.33	0.99	1.22	5.54
20	United Arab Emirates	2.73	0.86	1.92	5.51

Tabelle 2.1: *Übersicht über die 20 attraktivsten Offshoring-Länder, nach [A. T. Kearney 2007, S. 2]*

Unit beträgt die jährliche Produktivität von IT-Angestellten in Indien nur 39.033 US$ gegenüber 82.255 US$ in Deutschland, 154.173 US$ in den USA und sogar 386.413 US$ in Taiwan [Thomas 2007].

Die beispielhafte Betrachtung der Situation in Indien verdeutlicht auch typische volkswirtschaftliche Probleme. Steigende Lohnkosten (etwa 13% jährlich) führen zu zunehmendem Wohlstand und Inflexibilität. Eine starke Fluktuation von Arbeitskräften beeinträchtigt Projekte ebenso wie eine Entwicklung der Infrastruktur, die nicht Schritt hält, und akute Energienöte in den Technikzentren [Schaaf 2005]. In der Folge werden schon jetzt Aufträge von Indien nach China und in andere Länder outgesourct.

2.3.2 Gründe für das Offshoring der Anwendungsentwicklung

Bei den Begriffsdefinitionen von Outsourcing und Offshoring haben wir gesehen, dass Kostensenkung ein Hauptgrund für das Offshoring ist. Mit Offshoring verbinden sich in der Regel weitere Hoffnungen. Die nachfolgenden Ausführungen beruhen auf Daten aus empirischen Umfragen unter Unternehmen, die Offshoring betreiben (z. B. [Schaaf u. Weber 2005], [Moczadlo 2003], [Moczadlo 2005], [Hatch 2005]), und zusammenfassenden Darstellungen in der Fachliteratur (z. B. [Carmel u. Tjia 2006], [Nicklisch 2006], [Bruch 1998], [Boos et al. 2005], [Albayrak et al. 2007], [Schwarze u. Müller 2005]).

Allgemeine Outsourcing-Erwartungen

Ein häufiger strategischer Grund für Outsourcing-Entscheidungen ist eine Konzentration auf die Kernkompetenzen. Das Ziel bei dieser Strategie ist, dass Unternehmen nur diejenigen Funktionen mit eigenen Mitarbeitern ausführen, in denen sie ihr Hauptgeschäft und wichtige Unterscheidungsmerkmale zur Konkurrenz sehen. Der Rest wird als austauschbar angesehen und ausgelagert. Wenn es sich dabei um Teile von Produktionsprozessen handelt, spricht man in der Betriebswirtschaftslehre auch vom Reduzieren der Fertigungstiefe.

Vom Outsourcing versprechen sich viele Unternehmen auch finanztechnische Vorteile. Durch die Verlagerung an externe Dienstleister können einige fixe Kosten zu variablen Kosten werden. Die Kosten einer Dienstleistung werden durch Outsourcing explizit gemacht. Dadurch kann eine größere Transparenz und Planbarkeit entstehen. Outsourcing kann auch günstiger als eine Eigenerstellung sein, wenn der Dienstleister Vorteile durch seine Größe, Marktmacht oder eine gute Auslastung bei parallelem Arbeiten an verschiedenen Projekten weitergeben kann (der sogenannte Skaleneffekt). Zu Kostenvorteilen beitragen können auch Marktmechanismen durch eine Konkurrenz mehrerer Dienstleister. Mit geringen eigenen Investitionen können Unternehmen über einen Dienstleister Zugriff auf moderne Technologien erhalten. In Grenzen ist auch eine Risikoteilung mit dem Dienstleister möglich.

Oft ist Outsourcing keine langfristige strategische Entscheidung, sondern dient zur Entschärfung von kurzfristigem Personalmangel. Kapazitätsengpässe aufgrund von Marktschwankungen sollen überbrückt und der Personaleinsatz flexibilisiert werden. Outsourcing kann auch eine Alternative sein, wenn ein Unternehmen nicht über ausreichend qualifizierte Mitarbeiter verfügt und geeignetes Personal nicht selber anstellen möchte. Wenn sich Outsourcing-Dienstleister auf bestimmte Bereiche spezialisieren, können sie diese spezifischen Leistungen effizienter und mit höherem Standard anbieten. Daher erwarten viele Auftraggeber auch eine höhere Qualität von outgesourcten Dienstleistungen als bei der Erbringung im eigenen Unternehmen.

Spezifische Offshoring-Erwartungen

Viel stärker noch als beim herkömmlichen Outsourcing stehen beim Offshoring für die beauftragenden Unternehmen potenzielle Kosteneinsparungen im Vordergrund. Während nach einer Umfrage des Handelsblattes unter deutschen Unternehmen zu Outsourcing allgemein die erzielten Spareffekte in der Regel mit 5 bis 20% beziffert wurden und nur sehr wenige Befragte Werte über 30% angaben [Koenen 2004], sind beim Offshoring größere Einsparungen möglich. In einer Umfrage von 2004–2005 gaben mehr als zwei Drittel der befragten deutschen Unternehmen an, 20% oder mehr einzusparen. Gut 19% konnten sogar Kostenvorteile von mehr als 40% realisieren [Moczadlo 2005].

Diese Einsparmöglichkeiten ergeben sich hauptsächlich aus den geringeren Lohnkosten von Programmierern in Offshore-Ländern. Nach einer 2004 veröffentlichten Übersicht kommen indische Programmierer auf einen Stundenlohn von 7 €, im Vergleich zu 44 € in den USA und 54 € in Deutschland, siehe Tabelle 2.2.

Land	**€/h**
Indien	7
Tschechien	8
Russland	9
Portugal	14
China	14
USA	44
Deutschland	54

Tabelle 2.2: *Arbeitskosten von Programmierern im Jahr 2004, nach [Schaaf u. Weber 2005, S. 6]*

Die Kostenvorteile in Bezug auf die Gehälter sind durch die schon angesprochenen Lohnsteigerungen, gerade in Indien, mittlerweile nicht mehr so extrem. Nach einer Studie von neoIT wird die jährliche Wachstumsrate der Gehälter bis 2010 in Indien

bei 8,7% liegen, gegenüber 6,5% in Tschechien, 7,2% in China und Russland und nur 3,6% in den USA [Eswaran u. Satpathy 2006]. Dennoch ist immer noch ein deutlicher Gehaltsunterschied vorhanden. Gerade in der Softwareentwicklung machen die Lohnkosten einen sehr großen Teil der Gesamtkosten aus. Dadurch ergibt sich in der Regel ein immer noch beträchtlicher Kostenvorteil, auch unter Berücksichtigung von höheren Overhead-Kosten durch das Offshoring (Management, Abstimmung, Bürokratie) und unabhängig vom Standort ähnlichen Kostenfaktoren (insbesondere Hardware-, Infrastruktur-, Kapital- und Immobilienkosten).

Auch ein einheimischer Fachkräftemangel ist eine wichtige Motivation für das Offshoring. Wenn der Zugriff auf IT-Experten mit guter Ausbildung durch einen eingeschränkten heimischen Arbeitsmarkt schwierig ist, lassen sich über Offshoring zusätzliche Personalressourcen akquirieren. In Deutschland werden in der IT-Branche nach einer Prognose des Branchenverbands BITKOM zwischen 12.000 und 17.000 IT-Nachwuchskräfte jährlich benötigt. Da nicht genügend ausgebildet werden, droht ein Fachkräftemangel. Die Experten sehen insbesondere ein wachsendes Defizit bei Absolventen mit IT- und Wirtschafts-Know-how. Wenn nicht gegengesteuert wird, könnten im Jahr 2011 etwa 6.000 qualifizierte Mitarbeiter fehlen [Röder et al. 2007].

Als ein spezifischer Vorteil der Offshore-Entwicklung wird die Möglichkeit gesehen, durch Ausnutzung unterschiedlicher Zeitzonen rund um die Uhr entwickeln zu können. Dies wird im englischen Sprachraum als 24/7- oder „Follow the Sun“-Entwicklung bezeichnet [Yap 2005]. Während der Zeitzonenunterschied beispielsweise zwischen Deutschland und Indien nur 3,5 (Sommerzeit) oder 4,5 Stunden (Winterzeit) beträgt, beträgt er zwischen den USA und Indien etwa einen halben Tag. In der Praxis wird es aufgrund von Koordinationsaufwänden aber schwierig sein, mit täglichen Übergaben ohne zwischenzeitliche Abstimmungen zu arbeiten.

2.3.3 Herausforderungen und realer Nutzen

In diesem Abschnitt werden dem Nutzen von Offshoring-Projekten spezifische Herausforderungen und Risiken gegenübergestellt.

Erfahrungen zum Erfolg von Offshoring-Projekten

Es ist schwierig, genaue Daten zum Erfolg von Offshoring-Projekten zu finden. Umfragen und Studien zeigen ein uneinheitliches Bild. Ein Problem dieser Untersuchungen scheint zu sein, dass fast ausschließlich Manager befragt werden und nicht die direkt beteiligten Analysten und Entwickler, die besser Auskunft über die in der Praxis auftretenden Probleme von Offshoring-Projekten geben könnten.

Bei einer Umfrage unter 5.000 Führungskräften in Nordamerika und Europa über den Erfolg ihrer Offshore-Strategie im Jahr 2004 werteten sie 36% als gescheitert. In mehr als einem Drittel der Fälle mussten Aufgaben vom Offshore-Team ins eigene

Unternehmen zurückverlagert werden (Backsourcing) [Hatch 2005]. Nach einer Studie im deutschsprachigen Raum sind zwei Drittel der befragten Unternehmen zufrieden bis sehr zufrieden mit den bisherigen Offshoring-Kooperationen. Allerdings sind auch 15% damit unzufrieden bis sehr unzufrieden [Schaaf 2004, S. 23]. Bei einer anderen Umfrage unter deutschen Unternehmen waren 91,6% der Befragten mit ihrem Onshore-Dienstleister zufrieden, gegenüber 84,6% bei Nearshore-Dienstleistern und nur gut einem Drittel bei Offshore-Dienstleistern [Prieß 2006].

Kalakota und Robinson erklären diese gemischten Erfahrungen durch ein Phasenmodell für die Verbreitung des Offshoring, siehe Abbildung 2.4 [Kalakota u. Robinson 2004]. Wie bei jeder Innovation, die durch das Management getrieben wird, folgt auf eine Phase

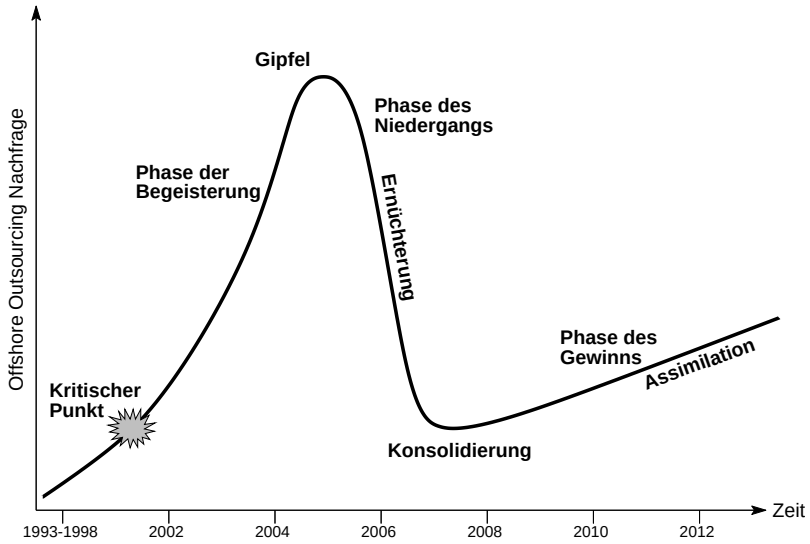


Abbildung 2.4: *Verbreitungsphasen des Offshoring, nach [Kalakota u. Robinson 2004]*

der Begeisterung und der stark ansteigenden Nachfrage eine Phase des Zweifelns und des Niedergangs. Wenn darauf anschließend eine Konsolidierung gelingt, steigt die Nachfrage langsam wieder an und die Innovation wird als gängige Erkenntnis übernommen. Nach der Schätzung aus dem Jahre 2004 erreichte der Offshoring-Bedarf bis 2005/2006 seinen Höhepunkt. Viele Unternehmen haben Offshoring-Projekte durchgeführt, um Erfahrungen in diesem Bereich zu sammeln oder weil sie sich durch Marktdruck dazu getrieben sahen. Diese Phase endete dann abrupt, als viele übereilt begonnene Projekte scheiterten und mehrere Unternehmen sich wieder aus dem Markt zurückzogen. Momentan befindet sich der Markt nach diesem Modell in der Konsolidierungsphase. Schwächere Anbieter werden verdrängt. Tragfähige Lösungen müssen entwickelt werden, um einen Weg aus der Krise zu finden.

Herausforderungen und Risiken des Outsourcings

Häufige Herausforderungen und Risiken bei Outsourcing-Projekten sind aus der Fachliteratur (vgl. die Literaturangaben auf Seite 29) gut bekannt. Hier werden die wichtigsten kurz genannt.

Mit einer Entscheidung für Outsourcing machen sich Unternehmen zu einem gewissen Teil von einem externen Dienstleister abhängig. Sie werden durch die feste vertragliche Bindung weniger flexibel. Dies kann bei der Anwendungsentwicklung dazu führen, dass Konflikte um die Anforderungen und die konkrete Umsetzung entstehen. Auch an Transparenz mangelt es, wenn der Dienstleister wenig Einblicke in den Prozess der Leistungserstellung gewährt. Mangelnde Transparenz gefährdet das Vertrauen des outsourcenden Unternehmens in den Dienstleister und erschwert eine sinnvolle Einflussnahme auf die Leistungserstellung durch den Auftraggeber.

Die Koordination und Steuerung eines externen Dienstleisters ist aufwendig und wird in der Regel unterschätzt. Unternehmensintern muss eine Steuerungsorganisation aufgebaut werden, die Schnittstelle zum Servicegeber wird. Der Koordinationsaufwand dabei kann beträchtlich und die benötigten Ressourcen höher als erwartet sein, beispielsweise für das Change Management.

Für die Beteiligten sind Outsourcing-Projekte nicht einfach. Das wechselseitige Verständnis von Anwendern und Entwicklern wird erschwert, wenn sie aus verschiedenen Unternehmen mit unterschiedlichen Kulturen und Wertvorstellungen stammen. Dadurch kann sich die Anforderungsermittlung in die Länge ziehen und die Wahrscheinlichkeit für Missverständnisse steigt. Das interne IT-Personal kann leicht durch Wegfall von Arbeitsplätzen demotiviert und verunsichert werden. Die Veränderung der internen Aufgaben von ausübenden hin zu steuernden Tätigkeiten kann problematisch sein, da diese den Mitarbeitern andere Qualifikationen abverlangen. Beim Servicegeber ist oft eine fehlende Identifikation mit dem Kunden und geringe Motivation zu beobachten. Es ist daher trügerisch, Qualitätsverbesserungen zu erwarten, insbesondere bei der Entwicklung von Anwendungssoftware [Grover et al. 1996]. Hinzu kommen ein Umsetzungsrisiko während der Verlagerung (z. B. bezüglich der kontinuierlichen Verfügbarkeit von Ressourcen) und häufig auftretende Probleme bei Datenschutz und IT-Sicherheit.

Herausforderungen der global verteilten Arbeit

Bei der global verteilten Entwicklung verschärfen sich einige der beschriebenen Probleme und es kommen neue hinzu. Sie ergeben sich u. a. durch unterschiedliche Kulturen, abweichende Zeitzonen und geopolitische Risiken (vgl. z. B. [Abbas et al. 1998b], [Carmel u. Tjia 2006], [Sahay et al. 2004], [Ågerfalk et al. 2005]).

Erran Carmel stellt in [Carmel 1999] ein anschauliches Modell für globale Teams vor, bei dem diese fünf Kräften ausgesetzt sind. So wie das Team verteilt wird, wirken sie

quasi als zentrifugale Kräfte aus der Mitte heraus und führen zu Problemen, siehe Abbildung 2.5.

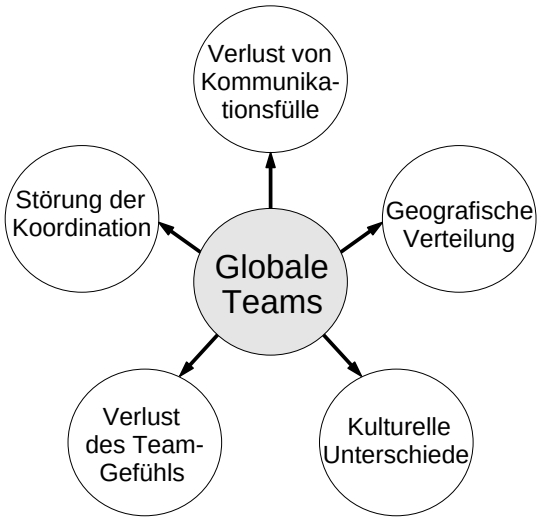


Abbildung 2.5: *Zentrifugale Kräfte bei globalen Teams, nach [Carmel 1999, S. 39ff.]*

Durch die verteilte Entwicklung wird die Kommunikation zwischen den Teams beeinträchtigt. Die Koordination von Aktivitäten und der Austausch von Wissen fällt durch die räumliche und temporale Trennung (unterschiedliche Zeitzonen) schwerer. Es leidet insbesondere die informelle Kommunikation der Beteiligten, da man sich selten persönlich begegnet. Ein verteiltes Entwicklungsprojekt ist weniger transparent und es ist daher schwierig, den Status akkurat zu beurteilen, insbesondere den Entwicklungsfortschritt.

Die durch das Arbeiten in unterschiedlichen Ländern gegebenen Kulturunterschiede können sich in einer Reihe von Aspekten niederschlagen. Durch eine andere Sprache oder einen anderen Dialekt kann es ebenso zu Missverständnissen kommen wie durch unterschiedliche Arbeitsweisen und die Unkenntnis fremden Geschäftsgebarens. Dadurch ist es zusätzlich schwierig, ein Team-Gefühl mit gemeinsamen Vorstellungen und Zielen zu entwickeln. Hinzu kommen landesspezifische Risiken, die mehr oder weniger stark ausgeprägt sein können: politische Instabilität, Währungsrisiken, infrastrukturelle Mängel und Sicherheitsprobleme wie der Umgang mit gewerblichen Schutz- und Urheberrechten. Auch der Arbeitsmarkt kann sehr speziell beschaffen sein und z. B. durch eine höhere Fluktuation der Entwickler Probleme für Projekte bringen (vgl. die Ausführungen über die Entwicklung in Indien in Abschnitt 2.3.1).

Die meisten dieser Punkte wirken sich auch direkt auf die Kosten von global verteilten Entwicklungsprojekten aus. Zusätzlich entstehen versteckte Kosten, die erst später sichtbar werden, insbesondere bei Offshoring-Projekten. Nach einer Hochrechnung des

CIO Magazine betragen diese versteckten Kosten für Offshoring-Projekte zwischen 16 und 65% [Overby 2003]: 1–10% für die Auswahl eines geeigneten Händlers und anfängliche Reisekosten, 2–3% für die Übergangsphase, 3–5% für Kündigungen nicht mehr benötigter on-site Mitarbeiter, 3–27% für schlechtere Produktivität durch hohe Fluktuation, mangelhafte Koordination und andere kulturell bedingte Probleme, 1–10% für die Verbesserung der Softwareentwicklungsprozesse, z. B. CMM-Zertifizierungen, und 6–10% für das Management des Offshore-Dienstleisters. Dies führt oft dazu, dass die Kosten der Softwareentwicklung weniger transparent und planbar als eigentlich beabsichtigt sind.

Zur Strukturierung der Herausforderungen bei Offshoring-Projekten und ihrer Folgen haben wir in [Kornstädt u. Sauer 2007b] ein dreistufiges Modell vorgestellt. Aus grundlegenden Herausforderungen können Folgeprobleme auf einer zweiten Ebene entstehen, die sich auf einer dritten Ebene in unbefriedigenden Ergebnissen ausdrücken, siehe Abbildung 2.6.

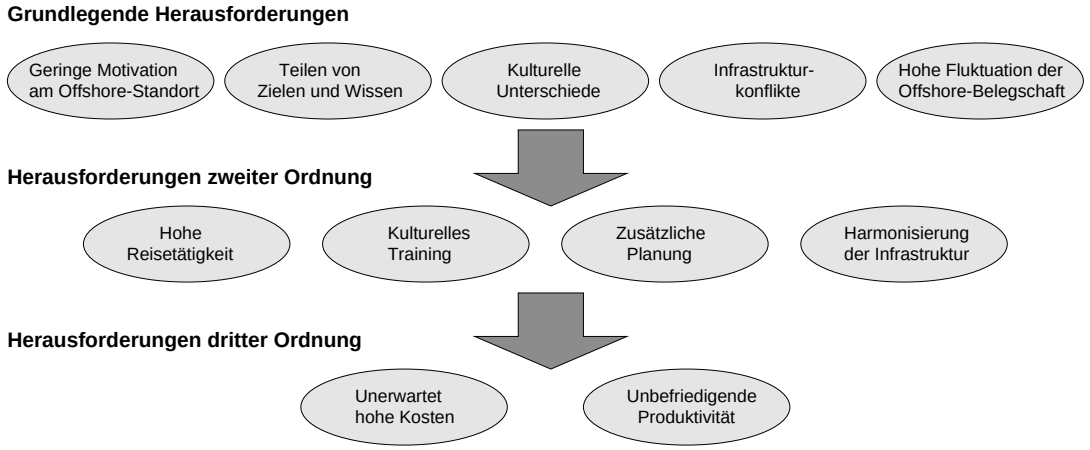


Abbildung 2.6: *Hierarchie von Herausforderungen des Offshoring der Anwendungsentwicklung, nach [Kornstädt u. Sauer 2007b]*

Einige grundlegende Herausforderungen der Offshore-Entwicklung wurden oben schon genannt. Es handelt sich dabei um Probleme, die direkt aufgrund der Charakteristika von Offshoring-Projekten entstehen können. Als wichtigste Herausforderungen können eine geringe Motivation von Mitarbeitern des Auftraggebers, die Entwicklung gemeinsamer Ziele, Wissenstransfer, kulturelle Unterschiede, Infrastrukturkonflikte und die Fluktuation von Offshore-Entwicklern angesehen werden. Aus den Bemühungen, mit diesen grundlegenden Herausforderungen umzugehen, entstehen abgeleitete Herausforderungen. Um die Teams näher zusammenzubringen, die Kommunikation zu stärken und der Kündigung durch gute Entwickler vorzubeugen, können viele Reisen und gegenseitige Arbeitsbesuche sowie ein beidseitiges kulturelles Training erforderlich sein. Gegenüber herkömmlichen Projekten ist mehr Planung nötig. Die technischen Probleme erfordern eine Abstimmung und eine Angleichung der Infrastruktur. Diese

und weitere Maßnahmen können dazu führen, dass die Kosten unerwartet hoch sind und damit die erhofften Einsparungen konterkariert werden. Dieser Effekt kann durch eine unbefriedigende Produktivität verstärkt werden.

Diese Herausforderungen sorgen dafür, dass Offshore-Projekte im Grundsatz schwieriger zu meistern sind als traditionelle Projekte. Matthew Simons drückt das wie folgt aus [Simons 2004, S. 3 f.]: "Offshore distributed projects belong at the top of the complexity scale because they must overcome not only the challenges of distance and of multiple organizations but also of language, cultural, and temporal inhibitors."

Beispiel: Kundenzufriedenheit

Am Beispiel der Kundenzufriedenheit sollen Herausforderungen von Outsourcing und Offshoring bei konkreten Fragestellungen aufgezeigt werden.

Nach Kano wird die Kundenzufriedenheit durch drei Faktoren beeinflusst, siehe [Kano 1984] und Abbildung 2.7:

Basisanforderungen sind implizite, grundlegende Erwartungen. Sie verursachen Unzufriedenheit beim Kunden, wenn sie nicht erfüllt werden. Werden sie erfüllt, wächst seine Zufriedenheit nicht, da ihm die Anforderungen nicht bewusst sind.

Leistungsanforderungen sind dem Kunden bewusst. Proportional zum Erfüllungsgrad entsteht bei ihm Zufriedenheit. Analog dazu entsteht bei ihm proportional zum Nichterfüllungsgrad Unzufriedenheit.

Begeisterungsanforderungen bedienen unausgesprochene Bedürfnisse und Wünsche des Kunden, mit deren Erfüllung er nicht rechnet. Daher entsteht bei ihm Zufriedenheit, wenn sie erfüllt werden, aber keine Unzufriedenheit, wenn nicht.

Beim **Outsourcing** sind mehrere Organisationen beteiligt. Dadurch wird der Entwicklungsprozess weniger flexibel, weniger transparent und aufwendiger. Leistungsanforderungen werden in Dokumenten explizit festgehalten. Nicht erfüllte Basisanforderungen müssen durch Rücksprache mit den Anwendern aufgedeckt werden. Das Risiko, Basisanforderungen zu übersehen, ist beim **Offshoring** noch größer. Durch die Entfernung und die andere Kultur fällt es den offshore arbeitenden Entwicklern in der Regel schwerer, die fachliche Domäne zu verstehen, sich auf den Kunden einzustellen und seine Bedürfnisse zu verstehen. Daher sollten nach Möglichkeit alle Anforderungen als Leistungsanforderungen explizit definiert und dem geografisch entfernten Team übermittelt werden.

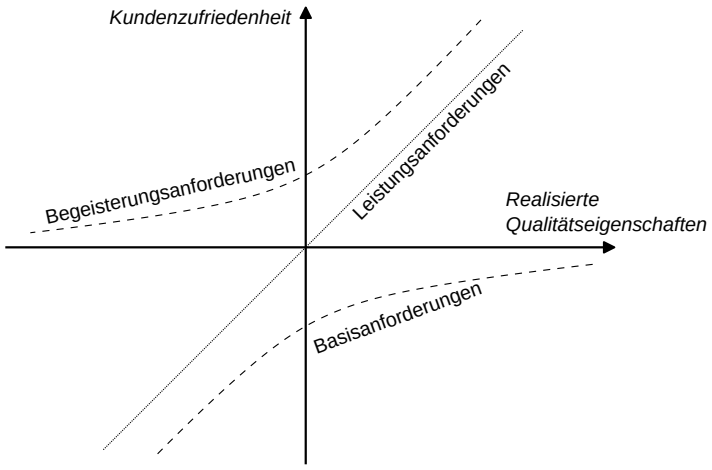


Abbildung 2.7: *Das Kano-Modell der Kundenzufriedenheit, nach [Kano 1984]*

2.4 Wissenschaftliche Fragestellungen

Aus den in der Praxis aufgetretenen Schwierigkeiten und Problemen bei global verteilter Entwicklung und den beschriebenen Ausprägungen des Offshoring ergeben sich interessante Fragestellungen für die Forschung. Westner und Strahringer haben Veröffentlichungen zum Offshoring von Informationssystemen in den Jahren 1996 bis 2006 untersucht [Westner u. Strahringer 2007]. Dabei haben sie festgestellt, dass es 2006 erst wenige Forschungsergebnisse auf diesem Gebiet gab. Seit 2003 sei aber die Zahl der Veröffentlichungen stark angestiegen. Anfang 2008 bilanzieren Hirschheim, Dibbern und Heinzl, dass sich die Forschung mittlerweile verstärkt bisher ungelösten Themen im Kontext des IT-Sourcings widmet und schon gewidmet hat [Hirschheim et al. 2008].

Im nächsten Abschnitt folgt ein Überblick über allgemeine Forschungsgebiete und Themen des Offshoring. Im darauf folgenden Abschnitt werden Forschungsgebiete und Themen für die Softwaretechnik bei global verteilter Entwicklung genannt.

2.4.1 Forschungsgebiete und Themen des Offshoring

Für die Volkswirtschaftslehre sind die Auswirkungen von Offshoring auf auslagernde Nationen und Anbieterländer interessant. Der World Investment Report der United Nations Conference on Trade and Development stellt die Entwicklung detailliert dar [UNCTAD 2004]: Die Zielländer erleben einen Anstieg des Exports, verbunden mit der Entstehung neuer Arbeitsplätze, höheren Löhnen und einer Verbesserung der beruflichen Qualifikationen. Davon profitieren viele Wirtschaftsbereiche, nicht nur die IT. Auf der anderen Seite werden gut qualifizierte Arbeitskräfte aus anderen Wirtschaftsbereichen abgezogen. Die prognostizierten Nebenwirkungen, z. B. Umweltverschmutzung und

Ausbeutung natürlicher Rohstoffe, werden als gering angesehen. Für Industrieländer ergibt sich die Chance, dass Unternehmen ihre Wettbewerbsfähigkeit durch Offshoring stärken und sich auf höherwertigere und produktivere Tätigkeiten konzentrieren können.

Shao und David haben eine Taxonomie von Arbeitsplätzen erstellt, die wahrscheinlich zukünftig ausgelagert werden bzw. im eigenen Land bleiben [B. M. Shao u. Smith David 2007]. Zur ersten Gruppe zählen sie Routinearbeiten (insbesondere Anwendungsentwicklung, Implementierung und Komponententests), Standarddienstleistungen mit hoher Austauschbarkeit („Commodity Services“) und die Auslagerung kompletter Geschäftsprozesse, zur zweiten spezialisierte IT-Qualifikationen, an einen bestimmten Ort gebundene Tätigkeiten und spezielle Unternehmensprozesse. Wie die Job Migration Task Force der ACM in ihrem Globalisierungsreport feststellt, ist es einfacher, Routinearbeiten in Entwicklungsländer auszulagern als höher qualifizierte Stellen. Während Offshoring anfangs auf diese Routinearbeiten ausgerichtet war, werden heutzutage vermehrt jedoch auch solche höherwertigen Arbeiten verlagert, z. B. im Forschungsbereich [Aspray et al. 2006]. Diese Entwicklung bestätigen zunehmende Berichte in den Medien über die Verlagerung solcher Aufgaben (vgl. z. B. [Lohr 2004]).

Offshoring hat damit Auswirkungen auf Ausbildungsinhalte in Industrieländern. Experten sehen eine Verschiebung weg von Programmierung hin zu Modellierung, Entwurf und Projektmanagement als nötig an. Für diese Entwicklung sind Wirtschaftsinformatiker besser aufgestellt als Kerninformatiker [Heinzl u. Autzen 2006].

Für die Wirtschaftsinformatik ist die Wirtschaftlichkeit von Offshoring-Projekten von Forschungsinteresse. Einer möglichen kurzfristigen Verringerung der Kosten werden langfristige Auswirkungen gegenübergestellt und die Gesamtkosten berechnet. Zur wissenschaftlichen Auswertung werden eine Reihe von Theorien herangezogen. Dazu gehören die Transaktionskostentheorie, bei der die Kosten der Koordination und Steuerung der Leistungserstellung berücksichtig werden, und die Ressourcentheorie, bei der die Auswirkungen des Offshoring auf Ressourcen, Strategie und zukünftige Entwicklung eines Unternehmens betrachtet werden. Eine Zusammenstellung weiterer angewendeter Theorien findet sich in [Dibbern et al. 2004]. Forschungsfelder sind auch der Auswahlprozess des Offshoring-Dienstleisters; das Vorgehen bei der Umstellung auf Offshoring; Wissensmanagement, Projektmanagement, Konfigurationsmanagement, Organisationsfragen und die Zusammenarbeit bei verteilten Teams. Ein Ziel dieser Bemühungen ist die Ermittlung von Risiko- und kritischen Erfolgsfaktoren (vgl. [Wiener 2006], [Jennex u. Adelakun 2003]). Eine weitere Absicht ist die Entwicklung geeigneter Werkzeuge zur Teamunterstützung im Bereich CSCW. Eine recht aktuelle Übersicht über diese und weitere Themen der Wirtschaftsinformatik findet sich auch bei King und Torkzadeh [King u. Torkzadeh 2008].

Einzelne Fragen des Offshoring spielen auch in anderen Disziplinen eine Rolle. Dies sind z. B. juristische Untersuchungen zu Verträgen, Gewährleistung und Haftung, Service Level Agreements, Arbeits- und Steuerrecht, Datenschutz und zur Einhaltung gewerblicher Schutz- und Urheberrechte (vgl. [Bräutigam u. Grabbe 2004]). Oder

soziologische Untersuchungen zur interkulturellen Zusammenarbeit von Menschen aus unterschiedlichen Ländern.

2.4.2 Rolle der Softwaretechnik bei verteilter Entwicklung

Welchen Beitrag kann die Softwaretechnik bei der globalen Entwicklung mit Offshore-Teams leisten? Bertrand Meyer stellte 2006 fest, dass diese Frage in der Softwaretechnik-Forschung noch nicht ausreichend diskutiert wird [Meyer 2006, S. 124]: "There's little analysis of offshore development in the technical computer science or software engineering literature, but this phenomenon is bound to have a profound effect on the field: not only the obvious human effect, but also a long-lasting influence on the technology itself." Meyer sieht Offshoring als eine große Herausforderung für die Softwaretechnik an. Während es in nicht-verteilten Projekten möglich ist, auch ohne die stringente Befolgung von Richtlinien der Softwaretechnik erfolgreich zu sein, ist das bei global verteilter Anwendungsentwicklung nicht mehr der Fall. Fast alle Fragen der Softwaretechnik sind für solche Projekte von großem Belang. Meyer nennt beispielhaft Anforderungsermittlung, Dokumentation, Konfigurationsmanagement, Qualitätssicherung, Verträge und Projektmanagement. Die vermehrten Veröffentlichungen der letzten Zeit machen deutlich, dass die Forschergemeinde die Herausforderung aufgegriffen hat. Es scheint aber noch ein weiter Weg zu sein, bis die Mechanismen und Kräfte von global verteilter Softwareentwicklung richtig verstanden werden.

Es bestehen schon seit einiger Zeit Bemühungen, eine internationale Forschungsgemeinschaft zu global verteilter Entwicklung aufzubauen und Kontakte zu benachbarten Disziplinen zu knüpfen [Hargreaves et al. 2004]. Andreas Braun unterscheidet in [Braun 2004] drei wesentliche Kategorien von wissenschaftlichen Fragestellungen für das GSD: Kommunikation, Organisation und Steuerung sowie Wissens- und Dokumentenmanagement. Bei der Kommunikation geht es um die Abstimmung von Einzelpersonen und Teams, die durch Entfernung, Zeit und Organisationszugehörigkeit getrennt sind, und ihre technische und nichttechnische Unterstützung. Bei der Organisation geht es um Koordination und Steuerung zur Unterstützung verteilt arbeitender Entwicklergruppen und um Strategien zum Konfliktmanagement. Wissens- und Dokumentenmanagement beschäftigt sich mit Artefakten und Dokumenten, die im Softwareentwicklungsprozess entstehen und Arbeitsergebnisse und Wissen darstellen: Quelltexte, Entwurfsdokumente, E-Mails u. Ä. Diese Ergebnisse müssen verwaltet und zwischen den Teams ausgetauscht werden. Problematisch ist dabei insbesondere implizites Wissen („Tacit Knowledge").

In dieser Arbeit wird die Forschung in diesen drei Kategorien in drei aufeinander aufbauenden softwaretechnischen Schwerpunkten beschrieben: global verteilte Entwicklung unter Anwendung von agilen Methoden und Techniken (Kapitel 3), geeignete Kooperationsmodelle für Agiles Offshoring (Kapitel 4) und der Einsatz von architekturzentrierter Entwicklung (Kapitel 5).

2.5 Zusammenfassung

In diesem Kapitel wurden grundlegende Konzepte und Begriffe der global verteilten Softwareentwicklung und des Offshoring eingeführt und Eigenschaften, Aussichten und Probleme von global verteilten Entwicklungsprojekten diskutiert.

Bei der global verteilten Softwareentwicklung arbeiten Entwickler an weit voneinander entfernten Standorten in einem Projekt zusammen. Eine häufig anzutreffende Organisationsform der global verteilten Softwareentwicklung ist das Offshoring. Offshoring ist ein Teilgebiet des Outsourcings, bei dem Aufträge an Unternehmen in Niedriglohnländern vergeben werden. Eine wichtige Unterscheidung besteht darin, ob die verschiedenen Teams zu einem globalen Unternehmen gehören (Offshore Insourcing) oder ob ein inländischer Dienstleister einen externen Offshore-Dienstleister beauftragt (Offshore Outsourcing).

Mit der global verteilten Anwendungsentwicklung verbinden sich viele Hoffnungen, insbesondere auf Kosteneinsparungen und flexiblen Zugriff auf Personalressourcen. Allerdings sind diese Entwicklungsprojekte auch besonders anspruchsvoll, da sie allgemeine Schwierigkeiten der Softwareentwicklung mit Anforderungen von verteilter Entwicklung und kulturellen Unterschieden verbinden. In der Praxis sind daher mehrere Projekte gescheitert. In diesem Kapitel wurden die Besonderheiten von global verteilter Entwicklung herausgearbeitet und Erklärungsmodelle für spezifische Herausforderungen vorgestellt.

Die global verteilte Entwicklung ist als Untersuchungsgegenstand für verschiedene Wissenschaften interessant. Es ergeben sich vielfältige Fragestellungen, u. a. in den Bereichen Wirtschaftsinformatik, Volks- und Betriebswirtschaftslehre sowie der Ausbildung in der Informatik. Besondere Herausforderungen stellen sich für die Softwaretechnik. In diesem Kapitel konnten Forschungsgebiete und Themen aufgezeigt und Forschungslücken identifiziert werden.

Die folgenden Kapitel bauen auf den in diesem Kapitel geschaffenen Grundlagen auf und untersuchen Lösungsansätze für die hier aufgeworfenen Fragestellungen. Im nächsten Kapitel werden zunächst Einsatzmöglichkeiten von agilen Methoden bei global verteilter Softwareentwicklung diskutiert.

3 Agile Methoden bei global verteilter Entwicklung

In diesem Kapitel wird der Einsatz von agilen Methoden bei der global verteilten Entwicklung untersucht, um neue Herangehensweisen für die im letzten Kapitel beschriebenen Probleme durch die Verteilung der Teams zu finden.

Das Kapitel beginnt mit einer Definition der wichtigsten Begriffe. Im darauf folgenden Abschnitt wird herausgearbeitet, welche generelle Sichtweise und welche Arbeitsvorgänge bei agilen Methoden im Mittelpunkt stehen. Anschließend diskutieren wir den scheinbaren konzeptionellen Widerspruch von verteilter Entwicklung und Agilität. Danach wird der praktische Einsatz agiler Methoden durch Auswertung veröffentlichter Erfahrungsberichte analysiert.

Die beobachteten spezifischen Probleme und Lösungsansätze dienen als Grundlage für eine kritische Reflexion, bei der die Beobachtungen und Schlüsse mit wissenschaftlichen Erkenntnissen abgeglichen werden. Daraus werden dann wichtige Bereiche für die weitere Analyse in dieser Arbeit abgeleitet.

3.1 Begriffsklärung

Eine Methode umfasst im Sprachgebrauch dieser Arbeit in Anlehnung an Lars Mathiassen [Mathiassen 1998] drei Dinge: Vorschriften zur Ausführung eines bestimmten Typs von Arbeitsvorgängen, ein Anwendungsgebiet und eine generelle Sichtweise auf die Art der Arbeitsvorgänge und ihr Umfeld. Die Vorschriften beinhalten Techniken (Wie wird ein Arbeitsvorgang ausgeführt?), Werkzeuge zur Unterstützung der Ausführung und organisatorische Regelungen. Eine Methode sollte so genau dokumentiert sein, dass sie weitergegeben und in unterschiedlichen Kontexten angewendet werden kann.

Beim Einsatz *agiler* Methoden bei der global verteilten Softwareentwicklung lassen sich allgemeine agile global verteilte Softwareentwicklung und die spezielle Variante Agiles Offshoring unterscheiden.

Definition Agile global verteilte Softwareentwicklung:

Agile global verteilte Softwareentwicklung (AGSD, vom engl. Agile Global Software Development) ist die Anwendung agiler Methoden und Techniken in global verteilten Projekten zur Softwareentwicklung.

Agile global verteilte Softwareentwicklung beinhaltet insbesondere die Arbeit in geografisch verteilten Teams von gleichem Status (vgl. die Begriffsklärung in Abschnitt 2.1).

Eine andere Form von AGSD ist Agiles Offshoring. Es lässt sich wie folgt definieren:

Definition Agiles Offshoring:

Beim Agilen Offshoring (AO) handelt es sich um agile global verteilte Softwareentwicklung, bei der die beteiligten Teams in einer Offshoring-Beziehung stehen.

In der Literatur finden sich hierfür auch die Begriffe Agile Offshore Software Development (AOSD, z. B. in [Massol 2004]), Agile Offshore Outsourcing (AOO) und Agile Outsourcing (trotz Verwendung des allgemeinen Outsourcing-Begriffs bezogen auf Offshore-Entwicklung, siehe [Braithwaite u. Joyce 2005a]).

3.2 Motivation für Agilität in verteilten Umgebungen

Agile Methoden wie Extreme Programming verbreiten sich immer mehr in der Praxis. Nach einer Untersuchung von Forrester Research setzten Ende 2005 14% aller nordamerikanischen und europäischen Unternehmen agile Softwareentwicklungsprozesse ein und weitere 19% erwogen oder planten dies. Während es anfangs meist kleine Hightech-Unternehmen waren, zogen immer mehr auch IT-Abteilungen in größeren Unternehmen nach [Schwaber 2005]. In einem neueren Report von Anfang 2010 gibt Forrester Research einen aktuellen Einsatzgrad von 35% an [West u. Grant 2010].

Bei verteilter Entwicklung spielen agile Methoden noch keine große Rolle. Dies liegt zu einem großen Teil daran, dass gerade indische Firmen bisher noch weniger auf Agilität setzen. Dies wird durch den hohen Anteil an CMM Level 5 und anderen Zertifizierungen bestätigt [Moore u. Barnett 2004]. Die Vereinbarkeit von CMM und Agilität wird in Abschnitt 3.2.4 diskutiert.

In diesem Abschnitt wird die prinzipielle Anwendbarkeit agiler Methoden bei verteilter Entwicklung betrachtet und begründet, warum eine Kombination beider Ansätze konzeptionell keinen Widerspruch darstellt. Dazu werden zunächst wichtige Merkmale

von agilen Methoden vorgestellt. Anschließend werden Chancen und konzeptionelle Schwierigkeiten von agiler verteilter Entwicklung diskutiert. Danach werden bekannte agile Methoden vorgestellt, die für die verteilte Entwicklung angepasst wurden, und es wird das Thema CMM-Zertifizierung wieder aufgegriffen.

3.2.1 Merkmale agiler Methoden

Agile Methoden verfolgen das Ziel, ein leichtgewichtiges und flexibles Vorgehen zu unterstützen und die beteiligten Personen in den Mittelpunkt zu rücken.

Dieses Ziel spiegelt sich im **Agilen Manifest** wider. Es wurde von einer Gruppe von IT-Fachleuten und -Beratern im Jahr 2001 veröffentlicht und gilt seitdem als gemeinsame Basis aller agilen Methoden. Cockburn beschreibt die Hintergründe des Treffens in [Cockburn 2002]: Es waren Vertreter vieler leichtgewichtiger Ansätze anwesend – auch zu den im Folgenden vorgestellten drei Methoden – um Gemeinsamkeiten dieser Ansätze zu diskutieren. Statt des Adjektivs „leichtgewichtig“ wurde „agil“ gewählt, um die flexible Anpassung an sich ändernde Anforderungen innerhalb des Zeitrahmens eines Projekts in den Mittelpunkt zu stellen. Das Agile Manifest im Wortlaut [Agile Alliance 2001]:

> “We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> - Individuals and interactions over processes and tools
> - Working software over comprehensive documentation
> - Customer collaboration over contract negotiation
> - Responding to change over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.”

In [Abrahamsson et al. 2002] werden weitere Merkmale von agilen Entwicklungsmethoden beschrieben:

- Die Entwicklung verläuft inkrementell mit häufiger Veröffentlichung neuer Versionen.
- Die Entwicklung ist kooperativ: Kunden und Entwickler arbeiten beständig zusammen und kommunizieren dabei eingehend.
- Die Entwicklung ist einfach in dem Sinne, dass die verwendete Methode schnell erlern- und veränderbar ist.
- Die Entwicklung ist leicht anpassbar: Die Beteiligten können schnell und flexibel auf Änderungen reagieren.

Gegenüber plangetriebenen Methoden (nach [Fowler 2006b]; in anderen Quellen auch: prozessgetrieben oder konventionell) lässt sich mit agilen Methoden eine größere Flexibilität erreichen. Dokumentation, Planung und Abstimmungstreffen werden auf das nötigste Maß beschränkt. Viele agile Methoden versuchen, ein großes Einsatzgebiet abzudecken, und zeigen dabei mehr oder wenige schwerwiegende Lücken. Experten empfehlen, diese breite Ausrichtung einzuschränken und an das jeweilige Gebiet angepasste Methoden zu entwickeln [Abrahamsson et al. 2003].

3.2.2 Agil und verteilt: konzeptionelle Gegensätze?

In diesem Abschnitt wird diskutiert, wie agile Methoden in global verteilten Entwicklungsprojekten eingesetzt werden können, obwohl Agilität bei oberflächlicher Betrachtung in einem Gegensatz zu verteilter Entwicklung steht.

Probleme plangetriebener verteilter Projekte

Am Beispiel des Offshoring wird deutlich, dass global verteilte Entwicklung einfach mit einem Softwarentwicklungsprozess nach dem Wasserfallmodell [Royce 1987] vereinbar zu sein scheint. Dabei legt der Auftraggeber seine Anforderungen und die Abnahmekriterien fest. Anhand der Anforderungen wählt er einen Offshore-Dienstleister aus, der die Anforderungen zu einem Festpreis in reiner Offshore-Arbeit umsetzt und dem Auftraggeber die fertige Anwendung übergibt. Dieser testet sie anhand der Abnahmekriterien und überführt die Anwendung bei Erfolg in den produktiven Betrieb. Ansonsten muss der Offshore-Dienstleister nachbessern.

Dieses Vorgehen besitzt jedoch eine ganze Reihe von Nachteilen, die sich in jedem Prozess nach dem Wasserfallmodell finden [Sommerville 2009]:

- Fehler werden möglicherweise erst spät erkannt. Wenn Fehler in frühen Phasen passieren, entsteht für ihre Behebung ein erheblicher Aufwand auf beiden Seiten.
- Fehler bei der Anforderungsermittlung sind besonders kritisch. Die Anwendung kann erst gegen Ende des Entwicklungsprozesses getestet werden. Fehlerhafte Anforderungen und nicht erfüllte Basisanforderungen (vgl. Abschnitt 2.3.3) treten dadurch erst sehr spät zutage. Das Feedback der Anwender fließt nicht in die Entwicklung ein.
- Der gesamte Prozess ist wenig flexibel. Sinnvolle Änderungen, die z. B. durch zwischenzeitlich veränderte betriebliche Abläufe und Rahmenbedingungen entstehen, können nicht so einfach umgesetzt werden.

Diese Probleme werden bei global verteilter Entwicklung durch interkulturelle Unterschiede verschärft. Dies führt mit dazu, dass viele Offshoring-Projekte scheitern (vgl. Abschnitt 2.3.3).

Chancen von agiler verteilter Entwicklung

Agile Methoden gehen die Probleme plangetriebener verteilter Projekte an. Durch das iterative Vorgehen können Fehler frühzeitig entdeckt werden. Die Anwender sind permanent beteiligt und liefern Feedback zur Anwendung. Diese Anregungen können genauso zeitnah berücksichtigt und umgesetzt werden wie durch externe Faktoren geänderte Anforderungen. Eine Kombination von agilen Methoden und Offshore-Entwicklung verspricht die Kombination der jeweiligen Vorteile, also insbesondere Flexibilität und Kostenersparnis.

Agile Methoden bieten erprobte Lösungsansätze für das Kommunikationsproblem an. Auch wenn diese Lösungsansätze nicht ohne Änderung auf verteilte Projekte übertragbar sind, wird das Problem damit explizit gemacht und aktiv angegangen [Paasivaara u. Lassenius 2006]. Ein gutes Beispiel für den Umgang mit möglichen Kommunikationsproblemen ist die Einbeziehung des Kunden in die Entwicklung, bei der er Teil des Teams wird. Dessen Einbindung kann dabei helfen, Vertrauen und Gemeinschaft aufzubauen [Korkala et al. 2009], [Danait 2005].

Entwicklungsteams haben bei agilen Methoden mehr Verantwortung und sind es gewohnt, selbstständig zu arbeiten. Diese Eigenschaft ist bei verteilter Entwicklung sehr nützlich: Die Umstellung auf verteilte Entwicklung kann agilen Teams leichter fallen als konventionell arbeitenden Teams. Die kontinuierliche Ermittlung und Abstimmung der Anforderungen führt nicht nur zu mehr Flexibilität im Entwicklungsprozess, sondern auch zum Verzicht auf eine vollständige Spezifikation zu Beginn des Projekts. Bei agilen Methoden ist die Möglichkeit zur Durchführung von Änderungen in den Entwicklungsprozess integriert. Dadurch sind Anpassungen viel leichter und schneller möglich als mit einem fromalen und schwergewichtigen Change Management, das in konventionellen Projekten installiert wird.

Mit den häufigen Auslieferungen und Feedbackmechanismen agiler Methoden erhält man eine frühe Rückkopplung, ob die verteilte Entwicklung funktioniert und ob die Anforderungen von allen richtig verstanden wurden. Dies ist ansonsten ein häufiges Risiko bei verteilten Entwicklungsprojekten [Paasivaara u. Lassenius 2006]. Mit einem intensiven Autor-Kritiker-Zyklus können ggf. schnell Gegenmaßnahmen getroffen werden. Ein weiterer Vorteil der häufigen Auslieferungen besteht darin, dass Schwierigkeiten bei der Integration von getrennt entwickelten Teilen über ein gemeinsames Versionsverwaltungssystem schon bei jedem Einchecken direkt durch die jeweiligen Entwickler behoben werden und nicht erst am Ende des Projekts, wenn sich die Probleme zu einem großen zusammenfügen [Simons 2002], [Eckstein 2009].

Agile Methoden versprechen auch Erfolg bei Projekten, die sonst nur sehr schwer mit verteilter Entwicklung zu realisieren wären. In Abbildung 3.1 werden Projekte nach ihren Erfordernissen in Bezug auf Anforderungsmanagement und Interaktion klassifiziert. Die Erfordernisse an das Anforderungsmanagement steigen mit zunehmender Instabilität der Anforderungen, größerer Anzahl von Schnittstellen und höherer

Komplexität von technischen Anforderungen. Die Interaktionsanforderungen steigen mit häufigerer Kommunikation mit dem Kunden und zunehmender Notwendigkeit, intern zu kommunizieren [Dumslaff 2007].

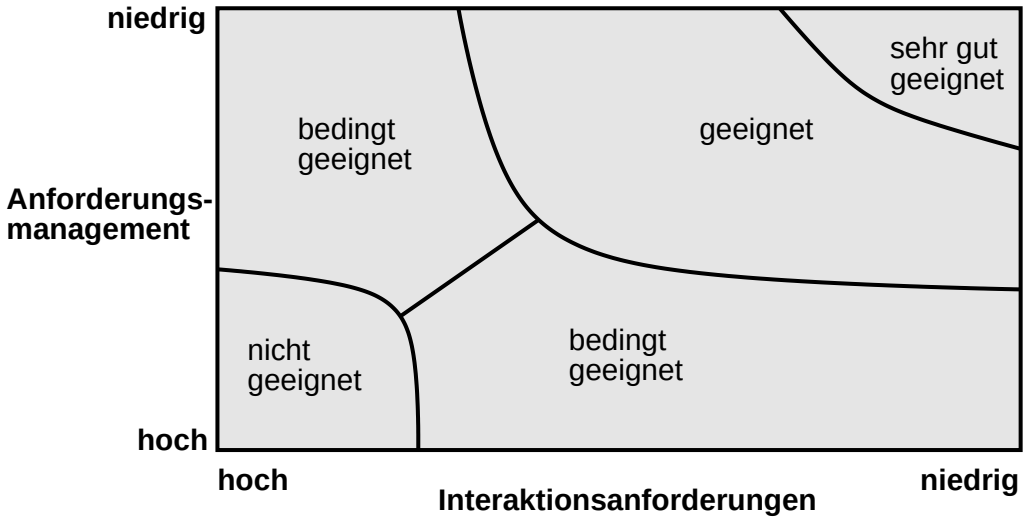


Abbildung 3.1: *Eignung von Projekten für verteilte Entwicklung, nach [Dumslaff 2007]*

Das Anforderungsmanagement und die Interaktion sind bei Offshoring-Projekten durch die räumliche, temporale und organisatorische Trennung besonders anspruchsvoll. Gut geeignet für das Offshoring sind daher Projekte mit niedrigen Ansprüchen an das Anforderungsmanagement und an Interaktion. Wenn die Interaktionsanforderungen hoch sind, die Ansprüche an das Anforderungsmanagement aber niedrig, lassen sich Projekte durch gute Kommunikationsfähigkeiten und ggf. Rückgriff auf formellere Interaktionsmethoden offshore bewältigen. Im umgekehrten Fall (niedrige Interaktionsanforderungen, hohe Ansprüche an das Anforderungsmanagement) sind etablierte Softwareentwicklungsprozesse, ein strukturiertes Anforderungs- und Changemanagement sowie gute Managementfähigkeiten gefordert. Agile Methoden zeigen Stärken in den erforderlichen Bereichen mit einem expliziten, flexiblen Anforderungsmanagement, das auch mit sich ändernden Anforderungen umgehen kann, und einer offenen Kommunikation aller Beteiligten mit intensivem Feedback und regelmäßiger Abstimmung.

Konzeptionelle Schwierigkeiten von agiler verteilter Entwicklung

Bei der Gegenüberstellung von ausgewählten Werten und Prinzipien agiler Methoden und den besonderen Eigenschaften konventioneller verteilter Entwicklung ergeben sich jedoch auch konzeptionelle Gegensätze (vgl. Tabelle 3.1).

Agile Methoden	**Global verteilte Projekte**
Die Entwickler arbeiten in einem Team eng zusammen und kommunizieren dabei eingehend.	Teams arbeiten auf verschiedenen Kontinenten.
Die Kunden geben den Entwicklern häufig persönlich Feedback und sind in den Prozess eingebunden.	Kunden und Entwickler sind durch Zeitzonen, unterschiedliche Sprachen und Kulturen (beim Offshoring zusätzlich Organisationsgrenzen) getrennt.
Auf die Etablierung einer gemeinsamen, vertrauensvollen Projektkultur wird viel Wert gelegt.	Die Teams arbeiten in zwei getrennten Welten, nicht nur geografisch, sondern beim Offshoring auch organisatorisch.
Die Anforderungen sind flexibel und können während der Entwicklung geändert werden.	Die Anforderungen und Abnahmekriterien werden vor der Entwicklung detailliert spezifiziert.
Die Prozesse sind informell, Menschen stehen im Vordergrund.	Die Outsourcing-Beziehung beruht auf formalisierten Prozessen und detaillierten Verträgen.

Tabelle 3.1: *Gegenüberstellung: agile Methoden und global verteilte Entwicklungsprojekte*

Turk et al. haben verschiedene agile Projekte ausgewertet und einen Katalog von Prinzipien und Grundannahmen erstellt [Turk et al. 2002]. Sie haben dabei herausgefunden, dass sich agile Methoden nicht für alle Arten von Projekten gleichermaßen eignen. Zwei Aspekte aus dem entstandenen Katalog sind besonders problematisch für die in dieser Arbeit betrachteten Entwicklungsprojekte. Der erste Aspekt trifft generell auf GSD zu. Der zweite Aspekt ist spezifisch für Agiles Offshoring:

1. **Begrenzte Unterstützung für verteilte Entwicklung:** Agile Methoden gehen davon aus, dass Kunden und Entwickler eng zusammenarbeiten, sich Entwickler jederzeit Feedback der Kunden einholen können und alle Entwickler nah zusammensitzen. Durch die Verteilung leidet die informelle Kommunikation. Es ist schwieriger, über die räumlichen, zeitlichen, kulturellen und organisatorischen Grenzen zusammenzuarbeiten und ein Teamgefühl zu entwickeln. Gute Dokumentation sollte die Wissensvermittlung unterstützen. Die entfernt arbeitenden Entwickler benötigen sie, um sich in den fachlichen Kontext einzuarbeiten, da sie die Arbeit der Anwender nicht direkt vor Ort beobachten können. Werkzeuge zur Kommunikationsunterstützung können ebenfalls helfen, die Entfernung zu überbrücken. Vgl. auch [Ramesh et al. 2006].

2. **Begrenzte Unterstützung für Auftragsvergabe an Externe:** Für den Offshoring-Dienstleister ist in der Regel weniger das Wohl des Kunden als die eigene Gewinnmaximierung wichtig. Dies beeinflusst die Zusammenarbeit. Dem Vertrag als schriftliche Arbeitsgrundlage mit explizit definierten Anforderungen und Abnahmekriterien kommt eine große Bedeutung zu. Änderungen an den Anforderungen sind bei festen Verträgen über Change Requests potenziell teuer und langwierig. Vgl. auch [Ambler 2003].

Fazit

Die Diskussion von konzeptionellen Vor- und Nachteilen des Einsatzes von agilen Methoden in verteilten Projekten zeigt, dass die Kombination von Agilität und Verteilung sinnvoll sein kann. Aufgrund ihrer teils gegensätzlichen Grundannahmen müssen agile Methoden aber sorgfältig in den verteilten Entwicklungsprozess integriert werden.

Um dafür Empfehlungen aussprechen zu können, sollen zunächst die bisherigen praktischen Erfahrungen mit Agilität in global verteilten Projekten untersucht und mögliche Probleme und Lösungsansätze identifiziert werden. Dazu werden in Abschnitt 3.3 Erfahrungsberichte aus der Praxis betrachtet.

Vorher werden in den nächsten beiden Abschnitten bekannte agile Methoden und das Verhältnis von agilen Methoden und CMM-Zertifizierung betrachtet, da diese Themen Grundlage für die darauf folgenden Abschnitte sind.

3.2.3 Bekannte agile Methoden mit Anpassungen für verteilte Entwicklung

Extreme Programming (XP), Scrum und die Dynamic Systems Development Method (DSDM) sind bekannte agile Methoden. Diese Methoden werden in Anhang A genauer vorgestellt. Aus einer Studie aus dem Jahr 2006 mit 722 Teilnehmern lässt sich die relative Verbreitung der Methoden ablesen. Unter den 84% der Befragten, deren Unternehmen Erfahrung mit agilen Methoden haben, kommt Scrum auf einen Anteil von fast 50%. 16,5% der Unternehmen setzen kombinierte oder zugeschnittene agile Methoden ein [VersionOne 2006], siehe Tabelle 3.2.

Methode	**Anteil [%]**
Scrum	47
Extreme Programming	27
kombiniert/zugeschnitten	16,5
DSDM	9,5

Tabelle 3.2: *Verbreitung agiler Methoden, nach [VersionOne 2006, S. 3]: „Welcher agilen Methode folgen Sie am engsten?“*

Weitere bekannte agile Entwicklungsmethoden sind u. a. Feature-Driven Development (FDD), eine agile Entwicklungsmethode, die sich insbesondere für große Projekte und heterogene Teams eignet [Palmer u. Felsing 2002], und Crystal, das aus einer Familie von agilen Methoden mit Varianten besteht, die je nach Projektgröße und -ausrichtung eingesetzt werden [Cockburn 2002].

Die Erfahrungen mit Agilität bei GSD sind noch begrenzt, sodass die Adaption agiler Methoden für die global verteilte Softwareentwicklung noch nicht weit fortgeschritten ist. Für XP, Scrum und DSDM existieren mit Distributed Extreme Programming (DXP), Distributed Scrum bzw. DSDM-O jedoch schon angepasste Varianten. Sie werden ebenfalls in Anhang A vorgestellt.

Allen drei angepassten Methoden ist gemein, dass sie die zugrunde liegende agile Methode ohne Veränderung ihrer jeweiligen Werte und Grundannahmen für verteilte Entwicklung adaptieren. Die Änderungen an allen drei Methoden sind recht gering. Im Mittelpunkt stehen jeweils Maßnahmen zur Verbesserung der Kommunikation zwischen räumlich getrennten Teams. Dafür werden Techniken, Vorgänge und Rollen erweitert und technische Systeme wie Videokonferenzen und Instant Messenger genutzt.

Grundlegende Unterschiede in der Art der Anpassungen ergeben sich aus der Ausrichtung der jeweils zugrunde liegenden Methode. Bei Distributed Extreme Programming konzentrieren sich die Änderungen auf konkrete Techniken. Techniken, die bei einem verteilten Team nicht in der ursprünglichen Form angewendet werden können, werden

angepasst (insbesondere Planungsspiel, Programmieren in Paaren, Fortlaufende Integration und Kunde vor Ort). Bei Distributed Scrum ändern sich vor allem Abstimmungs- und Planungsprozesse. Die Akteure sollen über wichtige Vorgänge in anderen Teams informiert bleiben, z. B. durch Telefonkonferenzen der Scrum Master. Bei DSDM-O ergeben sich zusätzliche Aktivitäten in einzelnen Prozessschritten. Zusätzlich werden bestehende Rollen an das Offshoring adaptiert und neue geschaffen.

3.2.4 Agile Methoden und CMM-Zertifizierung

In mehreren typischen Offshoring-Ländern, insbesondere Indien, setzen Unternehmen mehr auf disziplinierte, CMM-zertifizierte Prozesse als auf agile Methoden. CMM steht für Capability Maturity Model, auf Deutsch übersetzt mit Reifegradmodell. Mittels dieses Modells lässt sich die Reife eines Prozesses in einer Organisation messen, hier die Qualität des Softwareentwicklungsprozesses. 1991 wurde CMM in der ersten Version freigegeben. CMM wurde 2007 durch das modulare CMMI (Capability Maturity Model Integration) abgelöst. CMM kennt fünf Stufen, die bei CMMI Reifegraden entsprechen [Ahern et al. 2001]:

1. **Initial:** Ausgangszustand aller Organisationen.
2. **Repeatable (CMM)/Managed (CMMI):** Es existieren grundlegende, unter ähnlichen Bedingungen wiederholbare Prozesse. Erfahrungen aus durchgeführten Projekten werden bei neuen Projekten berücksichtigt.
3. **Defined:** Standardprozesse sind eingeführt und dokumentiert. Kosten und Zeiten werden überwacht.
4. **Managed (CMM)/Quantitatively Managed (CMMI):** Die Ziele werden quantitativ vorgegeben und überwacht. Kosten, Zeiten und die Produktqualität werden regelmäßig statistisch kontrolliert.
5. **Optimizing:** Der Prozess ist einer fortwährenden Optimierung unterworfen.

Diese Stufen machen deutlich, dass bei den Reifegradmodellen eine Standardisierung von Prozessen angestrebt wird. Viele Schritte werden dokumentiert und unterliegen einem beständigen Qualitätssicherungsprozess. Abweichungen sollen vermieden und möglichst alle Entscheidungen nachvollziehbar sein. Dies impliziert eine andere Unternehmenskultur als bei agilen Methoden, die – wie oben ausführlich diskutiert – mehr auf menschenzentriertes Handeln, Flexibilität und ständige Anpassung setzen. Dieser Unterschied kann in kooperativen Projekten leicht zu Problemen führen, beispielsweise wenn ein agil-geprägtes Team von einem CMM-geprägten Team flexibles Handeln erwartet und nicht bereit ist, die eigenen Arbeiten detailliert zu dokumentieren.

Um mit den unterschiedlichen Herangehensweisen umzugehen, kann ein geeigneter gemeinsamer Prozess definiert werden, der Anforderungen aus beiden Welten erfüllt. Dies kann recht aufwendige Verhandlungen erfordern, die Zugeständnisse und eine

Anpassung der Unternehmenskultur auf beiden Seiten erfordern. Zwei Erfolgsberichte zeigen, dass dies machbar ist. Bos und Vriens berichten in [Bos u. Vriens 2004], wie ein agiler, hauptsächlich auf XP basierender Prozess mit Scrum-Elementen die CMM Level 2 Zertifizierung erreicht hat. Dazu wurde eine Datenbank für User Storys eingeführt. Der Aufwand von Storys wird geschätzt. Sie können aus mehreren Tasks bestehen. Die Programmierer aktualisieren sie täglich mit den erbrachten Entwicklungszeiten und einer neuen Schätzung. Regelmäßig wird die Schätzung mit dem tatsächlichen Aufwand abgeglichen. In einem Projektmanagementplan werden die User Storys auf Iterationen aufgeteilt. Der Status von Iterationen und Storys ist von allen Beteiligten (Management, Programmierer, Kunden) jederzeit einsehbar. Quelltexte und andere Entwicklungsdokumente werden in einem gemeinsamen Projektarchiv aufbewahrt. Allen Dokumenten wird ein Status (Entwurf, Vorschlag, Gebilligt) zugeordnet. Durch einen Beauftragten aus einem anderen Projekt erfolgt eine unabhängige Qualitätskontrolle. Er prüft monatlich den Stand und erstellt einen Report. Ebenfalls monatlich schreibt der Projektleiter einen Projektstatusbericht. Bos und Vriens berichten von sehr guten Erfahrungen mit diesem angepassten Prozess. Der einmalige Umstellungs- und der regelmäßige Dokumentationsaufwand werden in Anbetracht der Vorteile als gering eingeschätzt.

Sutherland, Jakobsen und Johnson beschreiben in [Sutherland et al. 2007a], wie mit Scrum sogar CMMI Level 5 erreicht werden kann. Dabei wurde der umgekehrte Weg beschritten und Scrum in eine CMMI Level 5 zertifizierte Organisation eingeführt. Kernpunkte der Einführung waren das Training der Mitarbeiter in agilen Methoden, die klare Definition und Zuweisung von Rollen und Verantwortlichkeiten, ein einheitliches Konfigurationsmanagement, das Tracking des Projektfortschritts[1] und eine fortwährende Verbesserung des Einsatzes von agilen Methoden. Im Detail werden ähnliche Maßnahmen wie im von Bos und Vriens untersuchten Unternehmen eingesetzt: Die Auslieferungen werden genau geplant. Durch zweiwöchige Releases und die intensive Einbeziehung der Kunden können technische Probleme schon früh geklärt werden. Zu Beginn der Umsetzung einer neuen Story werden Abnahmekriterien definiert. Die gesamte Umsetzung wird von einem Prüfer begleitet, der an vorbestimmten Prüfpunkten zusammen mit dem Programmierer dessen Arbeit begutachtet. Diese Prüfungen sind leichtgewichtig und dauern typischerweise weniger als fünf Minuten. Auch in diesem Unternehmen wurde die Kombination von Agilität und CMMI als Erfolg gewertet. Die Produktivität konnte in einigen Projekten fast verdoppelt werden.

3.3 Analyse von veröffentlichten Erfahrungsberichten

Nach den prinzipiellen Betrachtungen zur agilen verteilten Entwicklung im vorangegangenen Abschnitt sollen jetzt Wege aufgezeigt werden, um Agilität erfolgreich in

[1] Ein Tracker, auf deutsch am ehesten mit „Verfolger“ übersetzbar, sammelt ausgewählte Informationen im Entwicklungsprozess und bereitet sie auf. Dazu gehören typischerweise Daten zur Budgetkontrolle, zum Projektfortschritt und zur Effektivität des Teams [Wolf et al. 2005].

global verteilten Entwicklungsprojekten einsetzen zu können. Dafür werden in diesem Abschnitt zunächst wichtige Themen, Probleme und Lösungsansätze aus der Praxis aufgegriffen und diskutiert.

Die Untersuchung baut auf einem Artikel aus dem Jahr 2006 auf [Sauer 2006b]. Sie bezieht weitere Erfahrungsberichte ein und diskutiert die gewonnenen Erkenntnisse. Die Ergebnisse werden detailliert klassifiziert und mit vergleichbaren Auswertungen und interviewbasierten Studien verglichen.

3.3.1 Methodik der Analyse

Ziele der Analyse

Mit dieser Analyse soll die erste Forschungsfrage zur global verteilten Entwicklung beantwortet werden: Welche Themen sind in der Praxis besonders problematisch? Die Untersuchung betrachtet die eingesetzten Methoden aus Sicht der Softwaretechnik und behandelt sowohl die spezifischen Probleme aus der Praxis als auch konstruktive Lösungsmöglichkeiten.

Die Erkenntnisse werden im Sinne einer Metastudie durch Auswertung mehrerer veröffentlichter Erfahrungsberichte gewonnen. Dies ist nötig, da der praktische Nutzen von einzelnen Erfahrungsberichten gering ist, wie auch Taylor et al. aufzeigen. Bei vielen Berichten wiederholen sich die beschriebenen Methoden, ohne dass wesentliche neue Ergebnisse dabei aufgezeigt würden [Taylor et al. 2006]. Die Argumentation in den meisten Erfahrungsberichten genügt wissenschaftlichen Kriterien nicht.

Die Erfahrungsberichte sind in ihrem Aufbau sehr heterogen und von unterschiedlicher Qualität und Ausführlichkeit. Die Erfahrungen sind meist episodisch beschrieben. Durch die Auswertung mehrerer Erfahrungsberichte wird vom Einzelfall abstrahiert; die Ergebnisse sind allgemeinerer Natur. Dies entspricht der Forderung von Westner und Strahringer, die aufgrund ihrer Analyse von veröffentlichten Forschungsberichten der letzten Jahre die Notwendigkeit empirischer Forschung auf Basis mehrerer Projekte herausstellen [Westner u. Strahringer 2007].

Auswahl der Erfahrungsberichte

Zur Auswahl der Erfahrungsberichte wurden die Tagungsbände und Webseiten relevanter Konferenzen in diesem Bereich genutzt. Dies sind im Wesentlichen die Konferenzreihen „Agile“, „Agile Business Conference“, „ICGSE“, „ICSE“, „XP“ und „XP/Agile Universe“. Zusätzlich wurden Literaturreferenzen in Fachartikeln nachverfolgt und im Internet veröffentlichte Artikel geprüft. Die Berichte wurden mit einbezogen, wenn sie über Projekte im professionellen Umfeld berichten, über ein gewisses wissenschaftliches Niveau verfügen und die aufgetretenen Probleme und die angewendeten Lösungsansätze detailliert genug beschreiben. Es stellte sich heraus, dass entsprechende Berichte zum

Recherchezeitpunkt noch selten waren, wie auch von anderen Forschern festgestellt wurde (vgl. z. B. [Paasivaara u. Lassenius 2006]). Es wurden Veröffentlichungen zu Offshoring-Projekten und zu allgemeinen GSD-Projekten bis zum Jahr 2006 berücksichtigt. Insgesamt wurden elf Erfahrungsberichte ausgewählt.

Bei der Auswahl wurde Wert darauf gelegt, dass unterschiedliche agile Methoden berücksichtigt werden, um die Untersuchungen nicht durch spezifische Unzulänglichkeiten einer Methode zu verfälschen. Zu jedem Bericht werden der Titel, die Autoren, das Jahr der Veröffentlichung, die Art der Projekte und der Kontext der Entwicklung angegeben. Bei der Art der Projekte wird zwischen GSD und AO unterschieden. Dabei wird die Einschätzung des Autors bzw. der Autoren des jeweiligen Berichts übernommen. Die vollständigen bibliografischen Daten finden sich jeweils im Literaturverzeichnis.

Erfahrungsberichte zur Kombination von Distributed Scrum und Distributed Extreme Programming bzw. Feature-Driven Development (vgl. Abschnitt 3.2.3):

- Agile Offshore Techniques, A. Danait, 2005, AO, Offshoring mit mehreren Teams in den USA und Indien mit Distributed Scrum und DXP, [Danait 2005]
- Cross-Continent Development Using Scrum and XP, B. Jensen und A. Zilmer, 2003, GSD, Verteilte Entwicklung mit einem dänischen Team und mehreren Teams aus den USA, [Jensen u. Zilmer 2003]
- Outsourcing and Offshoring with Agility: A Case Study, C. Kussmaul, R. Jack, B. Sponsler, 2004, AO, Offshoring der Entwicklung mit Teams in den USA und in Indien mit Distributed Scrum und FDD, [Kussmaul et al. 2004]

Erfahrungsberichte zu Distributed Extreme Programming:

- Distributed eXtreme Programming, M. Kircher, P. Jain, A. Corsaro, D. Levine, 2002, GSD, Verteilte Entwicklung von vier Einzelpersonen in Indien, Deutschland, den USA und Italien, [Kircher et al. 2002]
- Distributed Product Development Using Extreme Programming, C. Poole, 2004, GSD, Verteilte Entwicklung mit Teams aus den USA und aus Dublin, [Poole 2004]
- Follow the Sun: Distributed Extreme Programming Development, M. Yap, 2005, GSD, Verteilte Entwicklung mit Teams aus den USA, Großbritannien und Singapur, [Yap 2005]
- XP Expanded: Distributed Extreme Programming, K. Braithwaite, T. Joyce, 2005, GSD, Dasselbe Projekt wie bei [Yap 2005], [Braithwaite u. Joyce 2005a]

Erfahrungsbericht zu DSDM-O:

- Embracing Agility – The Change Evolution, A. Craddock and M. Singhal, 2006, AO, Offshoring mit Teams in Großbritannien und Indien, [Craddock u. Singhal 2006]

Erfahrungsberichte mit angepassten Methoden:

- AOSD: Agile Offshore, V. Massol, 2004, AO, Offshoring mit großen Entwicklungsteams in Frankreich und Indien, [Massol 2004]
- Internationally Agile, M. Simons, 2002, AO, Offshoring mit einem Entwicklungslabor in Indien mit XP-Techniken, [Simons 2002]
- Using an Agile Software Process with Offshore Development, M. Fowler, 2006, AO, Dasselbe Projekt wie bei [Simons 2002], [Fowler 2006b]

Analyse mit offenem Kodieren und Kategorienbildung

Aus den Berichten wurden relevante Themen der global verteilten Entwicklung extrahiert. Diese Themen wurden dann nach zwei Gesichtspunkten analysiert:

1. Welche Probleme treten bei verteilter Entwicklung mit agilen Methoden häufig auf (siehe Abschnitt 3.3.3)?
2. Welche Lösungsansätze wurden für konkrete Probleme entwickelt (siehe Abschnitt 3.3.4)?

In der Analyse in dieser Arbeit wurden die Erfahrungsberichte mit Verfahren aus der Grounded Theory ausgewertet: offenes Kodieren und das Bilden von Kategorien (zur Theorie der Methodik siehe Abschnitt 1.3.1). Dabei wurden in einem ersten Durchlauf Konzepte für relevante Themen der global verteilten Entwicklung gebildet. Diese wurden zu Kategorien aggregiert. Im zweiten Durchlauf wurden Probleme und beobachtete Lösungsansätze bei diesen Themen untersucht.

Die Ergebnisse des offenen Kodierens werden kritisch reflektiert und in einem ersten Schritt mit Ergebnissen aus anderen Publikationen über die Auswertung von Feldstudien verglichen. In einem zweiten Schritt werden durch eine Untersuchung der Diskrepanzen zwischen Problemen und vorhandenen Lösungsansätzen wichtige Felder für die weitere Forschung identifiziert. Diese werden zur Absicherung mit wissenschaftlichen Erkenntnissen abgeglichen. Aus der Untersuchung werden dann zwei Schlüsselbereiche für die weitere Betrachtung ausgewählt.

3.3.2 Relevante Themen der global verteilten Entwicklung

Durch das offene Kodieren wurden 20 relevante Themen als Konzepte identifiziert, die anschließend in sechs Kategorien eingeteilt wurden. Diese Kategorien sind „Kommunikation zwischen allen Beteiligten“, „Zusammenarbeit zwischen den Entwickler-Teams“, „Einbeziehung des Kunden“, „Ausbildung und Kenntnisse“, „Infrastruktur“ sowie „Projektmanagement und Qualitätssicherung“. Die Kategorien sind absteigend nach

Häufigkeit der Nennungen der darin enthaltenen Themen geordnet; analog ergibt sich die Reihenfolge der Themen.

Tabelle 3.3 fasst die gefundenen Kategorien und Themen zusammen.

Kategorie	**Themen**
Kommunikation zwischen allen Beteiligten	Teamgeist Gemeinsame Vision Kulturelle Unterschiede Sprachliche Verständigung Informelle Kommunikation
Zusammenarbeit zwischen den Entwickler-Teams	Kooperation und Koordination Wissenstransfer Gemeinsame Verantwortlichkeit Aufgabenverteilung
Einbeziehung des Kunden	Abstimmung mit dem Kunden Anforderungsermittlung und -spezifikation Aufbau von Vertrauen
Ausbildung und Kenntnisse	Training der Onshore- und Offshore-Entwickler Arbeitszufriedenheit Entwurfs- und Architekturkenntnisse
Infrastruktur	Kommunikationsinfrastruktur Hard- und Softwareausstattung
Projektmanagement und Qualitätssicherung	Management verteilter Teams und QS Aufwandsschätzung Fortschrittskontrolle

Tabelle 3.3: *Katalog von relevanten Themen bei agiler verteilter Entwicklung*

3.3.3 Beobachtete Probleme

Im Folgenden werden die Probleme beschrieben, die bei den Themen aufgetreten sind. Hinter jedem Problem ist angegeben, in welchen Erfahrungsberichten es beobachtet wurde.

Kommunikation zwischen allen Beteiligten

Die am häufigsten genannten Probleme betreffen allgemein die Kommunikation zwischen allen Beteiligten.

Teamgeist: Durch die Verteilung reduziert sich der persönliche Kontakt unter den Beteiligten. Die verteilten Teams handeln weitgehend getrennt voneinander und sehen das jeweils andere Team als eigenständige Gruppe an. Das behindert das Entstehen eines Teamgefühls und ein abgestimmtes Handeln. Kulturelle Unterschiede können diesen Effekt verstärken. *[Jensen u. Zilmer 2003], [Poole 2004], [Yap 2005], [Braithwaite u. Joyce 2005a], [Massol 2004], [Simons 2002], [Fowler 2006b]*

Gemeinsame Vision: Der mangelnde Teamgeist ist auch ein Grund, warum es schwierig ist, eine gemeinsame Vision zu etablieren: "In a distributed environment [...] it is that much harder to gain that shared vision due to cultural, ideological, and technical differences that seem to be so easily misunderstood and misrepresented and are amplified by the remoteness of a team." [Poole 2004, S. 65] Eine gemeinsame Vision umfasst eine Menge geteilter Werte und Grundsätze für den Entwicklungsprozess und die Zusammenarbeit. Wenn es keine gemeinsame Vision gibt, hinter der alle Beteiligten stehen, kann es leichter vorkommen, dass man sich nicht versteht oder sogar gegeneinander arbeitet. Dies kann sowohl die Zusammenarbeit unterschiedlicher Entwicklungsteams als auch die Beziehung zwischen Auftraggeber und Auftragnehmer betreffen. *[Jensen u. Zilmer 2003], [Poole 2004], [Braithwaite u. Joyce 2005a], [Massol 2004], [Fowler 2006b]*

Kulturelle Unterschiede: Kulturelle Differenzen können sich auf ganz verschiedenen Ebenen zeigen: bei Organisationsstrukturen und Arbeitsweisen, bei länderspezifischen Eigenheiten und beim Verhalten jedes Einzelnen. In den Projekten führten die Unterschiede manchmal zu Missverständnissen zwischen den Beteiligten und Frust. Viele Unterschiede zeigten sich überraschend und es dauerte einige Zeit, um sie bei den Beteiligten zu thematisieren und einen Umgang mit ihnen zu finden. *[Kussmaul et al. 2004], [Poole 2004], [Yap 2005], [Massol 2004], [Fowler 2006b]*

Sprachliche Verständigung: Eine weitere häufige Quelle für Missverständnisse ist die Sprache. Da die Beteiligten bei den ausgewählten Erfahrungsberichten Englisch als Muttersprache oder als Fremdsprache gut beherrschten, war eine Verständigung grundsätzlich möglich. Trotzdem kam es zu mehreren Problemen, wie in dem Projekt aus [Yap 2005]: "Even though members of all the regions speak English, which on the surface presents a minimal language barrier, there is much more to a language than how its words are pronounced. Many misunderstandings arose as the groups did not really share a higher level common language." [Yap 2005, S. 2] Die Probleme in diesem und in anderen Projekten erklären sich u. a. durch die Aussprache, das Fehlen der Körpersprache bei telefonischer Kommunikation und durch die unterschiedliche Interpretation einzelner Wörter oder Redewendungen. *[Yap 2005], [Massol 2004], [Simons 2002], [Fowler 2006b]*

Informelle Kommunikation: In nicht-verteilten Projekten werden viele Informationen außerhalb formeller Treffen oder Arbeitsschritten ausgetauscht, beispielsweise während Kaffeepausen, zufälligen Begegnungen auf dem Gang oder Treffen nach

der Arbeit. Diese Treffen schaffen auch eine angenehmere Atmosphäre und tragen zum Aufbau von gegenseitigem Vertrauen und zum Entstehen eines Teamgeists bei. Durch die Verteilung der Beteiligten konnte diese Art informeller Kommunikation in den betrachteten Projekten zwischen den Teams nicht oder nur schlecht realisiert werden. *[Yap 2005], [Braithwaite u. Joyce 2005a], [Simons 2002], [Fowler 2006b]*

Zusammenarbeit zwischen den Entwickler-Teams

In diese Kategorie fallen Probleme, die entstehen können, wenn die Entwicklung zwischen mehreren Teams aufgeteilt wird und sich diese abstimmen müssen.

Kooperation und Koordination: Alle untersuchten Erfahrungsberichte nennen die Kooperation und die Koordination der Aktivitäten mehr oder weniger explizit als das Hauptproblem bei der Zusammenarbeit. Persönliche Gespräche sind bedingt durch die geografische und zeitliche Distanz durch verschiedene Zeitzonen selten möglich. Dies fördert Missverständnisse. Abstimmungen, die schnell vorzunehmen wären, wenn sich das Team an einem Ort befinden würde, benötigen so sehr viel mehr Zeit als in nicht-verteilten Projekten. Zur Abstimmung müssen formale Prozesse etabliert werden. Informationen und Vereinbarungen, die in nicht-verteilten Projekten mündlich weitergegeben werden, müssen schriftlich festgehalten und verbreitet werden. *[Jensen u. Zilmer 2003], [Kussmaul et al. 2004], [Kircher et al. 2002], [Poole 2004], [Yap 2005], [Braithwaite u. Joyce 2005a], [Craddock u. Singhal 2006] [Massol 2004], [Simons 2002], [Fowler 2006b]*

Wissenstransfer: Wissen, das in einem Team entsteht und das für alle wichtig ist, muss weitergegeben werden. Durch die Verteilung ist dieser Prozess explizit zu planen und zeitaufwendig. Das gemeinsame Wissen der Teams muss kontinuierlich aufgebaut und erhalten werden, auch wenn die Teammitglieder im Laufe eines Projekts wechseln. Der Wissenstransfer kann vor allem bei der Einführung von Neuerungen kritisch sein [Yap 2005, S. 6]: “We found that the most difficult part of employing distributed teams is introducing new architecture changes or technologies that are needed to satisfy customer requirements. [...] Without a whole team concensus, the team polarized regionally making it hard to introduce the change to the other regions.” *[Jensen u. Zilmer 2003], [Yap 2005], [Braithwaite u. Joyce 2005a], [Massol 2004], [Simons 2002]*

Gemeinsame Verantwortlichkeit: Wenn an verschiedenen Orten entwickelt wird, kann es aufwendig und schwierig sein, ein Gefühl der gemeinsamen Verantwortlichkeit bei der Arbeit mit Quelltexten zu etablieren. Zum einen ist es psychologisch schwierig, den eigenen Quelltext mit fremden oder weniger bekannten Entwicklern zu teilen und für deren Fehler mit geradezustehen. Zum anderen gab es in mehreren der beschriebenen Projekte Probleme mit der gemeinsamen Versionsverwaltungssoftware, meist aufgrund einer mangelhaften Infrastruktur (siehe die

entsprechende Kategorie weiter unten). *[Kircher et al. 2002], [Poole 2004], [Yap 2005], [Braithwaite u. Joyce 2005a], [Simons 2002]*

Aufgabenverteilung: Bei der Kooperation zwischen mehreren Teams ist die Verteilung der Aufgaben von wesentlicher Bedeutung. Dabei können zwei aufeinander aufbauende Aspekte unterschieden werden. Zum einen die Frage, welches Team welche Aufgaben übernimmt. Eine gute Abstimmung ist wichtig, da es sonst schnell zur doppelten Erledigung derselben Aufgabe oder zu Wartezeiten auf die Ergebnisse des anderen Teams kommen kann: "Team members at one location are sometimes blocked while waiting for actions or decisions taken at another site. This creates resentment on both sides; the dependent site resents the productivity impact and the loss of decision making power; the depended site resents the interruption of thought and activity." [Braithwaite u. Joyce 2005a, S. 184] Der andere Aspekt betrifft die Übergabe der Arbeit. Hier kommt es zu Koordinationsproblemen, wenn die Arbeitsergebnisse und Schnittstellen nicht eindeutig definiert sind. Dies betrifft nicht nur, aber insbesondere, die Schnittstellen von Anwendungskomponenten [Fowler 2006b]: "It's very hard to properly specify the semantics of an interface, so even if you have all the signatures right, you still get tripped up on assumptions about what the implementation will actually do." *[Jensen u. Zilmer 2003], [Kircher et al. 2002], [Braithwaite u. Joyce 2005a], [Fowler 2006b]*

Einbeziehung des Kunden

Der Kunde hat bei agilen Methoden eine aktive Rolle im Projekt und wird intensiv bei der Festlegung und Priorisierung der Anforderungen und beim regelmäßigen Test neuer Versionen einbezogen.

Abstimmung mit dem Kunden: Wenn bei verteilter Entwicklung schon die Kommunikation zwischen den Teams beeinträchtigt ist, ist das die Kommunikation mit dem Kunden erst recht. Da der Kunde meist nicht gewillt ist, Mitarbeiter für die Arbeit an entfernten Standorten abzustellen, bleibt den dort arbeitenden Teams nur die Kommunikation mit technischen Hilfsmitteln und ggf. die Benennung eines Kunden-Stellvertreters unter den eigenen Mitarbeitern. Das ist nicht ideal und führt dazu, dass die Klärung von fachlichen Fragen mit dem Kunden in der Regel sehr viel länger dauert. *[Kussmaul et al. 2004], [Kircher et al. 2002], [Poole 2004], [Simons 2002]*

Anforderungsermittlung und -spezifikation: Es ist für offshore arbeitende Teams sehr schwierig, Anforderungen des Kunden aus der Entfernung zu ermitteln. Daher wird diese Rolle so gut wie immer Beratern aus dem Onshore-Team übertragen. Unabhängig davon, ob diese Personen zur Organisation des Kunden oder des Dienstleisters zählen, stellt sich die Frage, in welcher Form die Anforderungen den Offshore-Teams übermittelt werden. Da diese Teams die Anforderungen nicht

aus erster Hand kennen und Nachfragen beim Kunden schwierig sind, müssen die Anforderungen so weitergegeben werden, dass sie eindeutig interpretiert werden können. Die notwendigen Funktionen sollten möglichst vollständig beschrieben werden. Meist wird eine formalere Form als bei nicht-verteilten Projekten gewählt, was den Aufwand erhöht. *[Kircher et al. 2002], [Poole 2004], [Braithwaite u. Joyce 2005a], [Fowler 2006b]*

Aufbau von Vertrauen: Durch die Beeinträchtigung der Kommunikation und durch kulturelle Unterschiede ist es für die Entwickler schwieriger, beim Kunden Vertrauen in die eigene Arbeit und die Qualität der Ergebnisse sowie eine freundliche Arbeitsatmosphäre aufzubauen. Da die intensive Einbeziehung des Kunden eine wichtige Rolle bei agilen Methoden spielt, kann dies das gesamte Projekt beeinträchtigen. *[Jensen u. Zilmer 2003], [Kircher et al. 2002], [Braithwaite u. Joyce 2005a], [Craddock u. Singhal 2006]*

Ausbildung und Kenntnisse

Nicht alle Projektbeteiligten sind auf demselben Stand bezüglich technischem und fachlichem Wissen, insbesondere in einem Offshoring-Projekt.

Training der Onshore- und Offshore-Entwickler: Es kann sehr schwierig sein, einen einheitlichen Kenntnisstand aller Teammitglieder zu erreichen. Die Heterogenität des Personals ist ein Grund dafür. Weitere Gründe sind, dass Onshore- und Offshore-Teams oft nicht zu Beginn des Projekts zusammen auf die Arbeit vorbereitet werden können und dass sich die Kenntnisse im Verlauf des Projekts auseinander entwickeln. Mit einem regelmäßigen Tausch des Partners beim Programmieren in Paaren lässt sich in zusammensitzenden Teams ein einheitlicher Stil erreichen. Auch können so weniger erfahrene Entwickler ausgebildet werden. Diese Technik ist standortübergreifend jedoch nur eingeschränkt umsetzbar. Hinzu kommt die Schwierigkeit, agile Methoden in Firmen mit starken Hierarchien einzuführen. *[Danait 2005], [Yap 2005], [Craddock u. Singhal 2006]*

Arbeitszufriedenheit: Unzufriedenheit stellt sich bei einem offshore arbeitenden Team leicht ein, wenn die Entwickler ihnen langweilig erscheinende Tätigkeiten ausführen müssen. Dies kann der Fall sein, wenn die Anforderungen an die Anwendung zu strikt vorgegeben werden und kaum noch Gestaltungsspielräume bei der Entwicklung verbleiben. Die Projektleitung sollte auch keine Prozesse vorgeben, die sich nicht mit der lokalen Kultur in Einklang bringen lassen: “We found that each region should have a certain amount of flexibility to adapt the processes according to their local culture, but this doesn’t mean the region can do whatever it wants. There should be a balance between the global process and local adaptation.” [Yap 2005, S. 6] Die Arbeitszufriedenheit spielt wegen der in vielen Offshore-Ländern anzutreffenden hohen Fluktuationsrate eine wichtige Rolle. Entwickler verlassen schnell Unternehmen, die ihnen keine interessante Arbeit und Möglichkeiten zur

persönlichen Weiterentwicklung bieten können. *[Poole 2004], [Yap 2005], [Massol 2004]*

Entwurfs- und Architekturkenntnisse: In einigen Berichten wurden die Fähigkeiten und Kenntnisse der offshore arbeitenden Teams bemängelt, insbesondere bezüglich Systementwurf und -architektur. Dies kann zu Problemen bei der Qualität der erstellten Anwendung führen. *[Poole 2004], [Yap 2005]*

Infrastruktur

Eine funktionierende Infrastruktur ist eine wichtige Basis für eine gute Projektarbeit. Wenn Teams verteilt arbeiten, sind sie darauf noch mehr angewiesen.

Kommunikationsinfrastruktur: Damit die Kommunikation zwischen verteilten Teams reibungslos gelingt, ist vor allem eine gute Netzverbindung erforderlich, über die Telefonate, E-Mails, Quelltexte, Anwendungen, Videokonferenzen u. Ä. übertragen werden können. Da sich viele Offshoring-Dienstleister in Ländern mit weniger gut ausgebauter Infrastruktur befinden, kommt es gerade beim Offshoring regelmäßig zu Problemen mit mangelnder Bandbreite, zu geringer Übertragungsgeschwindigkeit und zusammenbrechenden Netzen. *[Kircher et al. 2002], [Yap 2005], [Braithwaite u. Joyce 2005a], [Fowler 2006b]*

Hard- und Softwareausstattung: Heterogene Hard- und Softwareausstattungen der Teams und abweichende Konfigurationen können zu Problemen führen, wenn sich dadurch die in Entwicklung befindliche Anwendung oder Entwicklungswerkzeuge je nach Rechner anders verhalten. Bei einem nicht-verteilten Team wird oft ein zentraler Integrationsrechner benutzt, auf dem die geänderte Anwendung getestet wird, bevor Änderungen in den gemeinsamen Quelltextbestand übernommen werden. Bei verteilten Teams lässt sich das mitunter nur umständlich realisieren, abhängig von der gemeinsamen Infrastruktur. *[Kircher et al. 2002], [Yap 2005], [Massol 2004], [Fowler 2006b]*

Projektmanagement und Qualitätssicherung

Projektmanagement und Qualitätssicherung (QS) gehören zu den wichtigsten Aufgaben in Entwicklungsprojekten. In den Erfahrungsberichten haben sich dabei eine Reihe von Problemen gezeigt.

Management verteilter Teams und QS: Die Abstimmung und Steuerung eines Projekts ist viel schwieriger, wenn die beteiligten Teams entfernt voneinander sind. Auch wenn onshore und offshore arbeitende Teams jeweils über lokale Projektleiter verfügen, müssen ihre Arbeiten koordiniert und zu einem Gesamtergebnis zusammengefügt werden. Die Qualität des Gesamtergebnisses muss genau untersucht werden. Das Management von Offshoring-Projekten wird durch kulturelle

Unterschiede erschwert, beispielsweise durch einen abweichenden Umgang mit Hierarchien und Kontrolle. Dadurch steigen auch das Risiko und die Kosten. Trotz des agilen Vorgehens können Probleme erst spät aufgedeckt werden, die bei einem lokal zusammenarbeitenden Team schon deutlich früher aufgefallen wären. *[Kircher et al. 2002], [Yap 2005]*

Aufwandsschätzung: Ein wichtiges Element des Projektmanagements ist die Aufwandsschätzung und -kontrolle. Durch die Verteilung wird es viel schwieriger, verlässliche Schätzungen zu erreichen: "Customers, managers, and developers all estimate projects differently, each using their own 'fudge factor', based on what the other groups tell them. In an offshore situation, where it is entirely likely that the development team will never meet the project manager or customers, the standard deviation of traditional project estimates can vary wildly." [Simons 2002] In einigen Projekten war es auch schwierig, entfernt arbeitende Teams überhaupt an den Schätzungen zu beteiligen. Das ist insbesondere dann kritisch, wenn ein Großteil der Entwicklungsarbeit offshore erledigt wird. *[Simons 2002]*

Fortschrittskontrolle: Für die Projektleitung ist es schwierig, den Status des Projekts festzustellen. Trotz der bei agilen Methoden regelmäßigen Releases werden manche aufgeschobenen Probleme erst spät bekannt. Hier spielen auch kulturelle Unterschiede eine Rolle, da es in verschiedenen Kulturen einen unterschiedlich offenen Umgang mit Problemen gibt. *[Simons 2002]*

3.3.4 Beobachtete Lösungsansätze

Die in den Erfahrungsberichten beschriebenen Lösungsansätze wurden ebenfalls in die sechs entwickelten Kategorien eingeteilt. In jeder Kategorie werden zuerst organisatorische Lösungen *(o)* beschrieben, d. h. besondere Änderungen in den Prozessen, Rollen, Konventionen, Aktivitäten oder Aufgaben. Anschließend Lösungen, die auf besonderen Dokumenten oder Artefakten beruhen *(d)* und zuletzt besondere Werkzeuge und Anwendungen *(w)*. Sofern Lösungen verschiedenen Kategorien oder Typen zugeordnet werden konnten, wurde die Einordnung gewählt, die am besten zu den jeweiligen Beschreibungen in den Erfahrungsberichten passt. Analog zum Vorgehen bei den Problemen werden auch bei den Lösungen jeweils die Berichte angegeben, in denen entsprechende Erfahrungen gesammelt wurden.

Kommunikation zwischen allen Beteiligten

(o) In den Berichten werden zwei gegensätzliche Strategien zum Umgang mit dem Kommunikationsproblem beschrieben. Zum einen ein Vorgehen, das eine intensive direkte Kommunikation zwischen allen Beteiligten fördert und erleichtert. Zum anderen eine Beschränkung und Formalisierung der Kommunikationswege und damit eine Reduzierung der Kommunikation. In jedem Fall sollte die Projektleitung Wert

darauf legen, dass Vertrauen zwischen den Beteiligten entsteht. Eine gemeinsame Vision der Anwendung mit einem klaren und einfach verständlichem Vokabular sollte entwickelt und allen vermittelt werden. Beim Aufbau von Vertrauen und als erster Schritt zum Entstehen eines Teamgeistes kann ein gemeinsames Kickoff-Treffen mit allen Beteiligten am Onshore-Standort helfen. Ein vorheriges kulturelles Training kann Vorurteile abbauen und gegenseitiges Verständnis schaffen. *[Jensen u. Zilmer 2003], [Kussmaul et al. 2004], [Kircher et al. 2002], [Poole 2004], [Yap 2005], [Braithwaite u. Joyce 2005a]*

(d) Um sicherzustellen, dass entfernt arbeitende Teams die Anforderungen verstanden haben, entwickeln diese Teams zuerst Komponententests auf Basis der Anforderungen. Diese Testfälle werden onshore geprüft, bevor offshore die eigentliche Funktionalität implementiert wird. *[Braithwaite u. Joyce 2005a], [Fowler 2006b]*

(w) Technische Maßnahmen sind sehr wichtig, um die Kommunikation zu unterstützen. Es sollten sowohl synchrone (Telefon, Videokonferenzsysteme) als auch asynchrone Kommunikationsmittel (E-Mail, Instant Messaging, Wiki) zur Verfügung stehen. *[Danait 2005], [Jensen u. Zilmer 2003], [Kussmaul et al. 2004], [Poole 2004], [Yap 2005], [Braithwaite u. Joyce 2005a], [Massol 2004], [Simons 2002], [Fowler 2006b]*

Zusammenarbeit zwischen den Entwickler-Teams

(o) Es werden eine ganze Reihe von Möglichkeiten beschrieben, um verteilte Teams in agilen Projekten zu unterstützen. Die Arbeitszeiten von Teams, die in unterschiedlichen Zeitzonen arbeiten, können angepasst werden, um eine größere Überlappung zu erreichen. Dadurch wächst der mögliche Zeitraum für synchrone Kommunikation. Zur Abstimmung werden tägliche Standup-Meetings abgehalten, telefonisch oder mittels Videokonferenzsystemen. Weitere speziellere Treffen werden wöchentlich oder monatlich angesetzt.

Damit sich die Teammitglieder besser kennenlernen, arbeiten einige Entwickler für mehrere Wochen im jeweils anderen Team an dessen Standort mit. Dadurch entstehen ein intensiverer persönlicher Kontakt und ein Verständnis für die jeweilige Arbeitssituation. Mitglieder, die neu ins Team kommen, werden allen anderen in einer Webkonferenz vorgestellt. Eine gute Gelegenheit für persönlichen Kontakt ergibt sich zu Beginn des Projekts, wenn im Anschluss an das Kickoff-Treffen ein gemeinsames Training für die Entwickler onshore stattfindet. „Botschafter“ und „Umherreisende Gurus“ rotieren als Experten zwischen den Teams, verbreiten fachliches und technisches Wissen und klären Missverständnisse. Die Aufgaben sollten so auf die Teams aufgeteilt werden, dass diese an möglichst kohärenten, voneinander unabhängigen Teilen arbeiten. Aufgaben, die viel gegenseitige Abstimmung benötigen, sollten innerhalb eines Teams bearbeitet werden. Dadurch wird die notwendige teamübergreifende Kommunikation verringert. Größere technische Änderungen, die viele Beteiligte betreffen, sollten nur in einem etablierten formalen Prozess bekannt gemacht werden. Aufgrund der verteilten Entwicklung und

des dadurch erforderlichen komplexeren Entwicklungsprozesses kann es nötig werden, die Iterationsdauer zu erhöhen oder flexibler zu gestalten. *[Danait 2005], [Jensen u. Zilmer 2003], [Kussmaul et al. 2004], [Kircher et al. 2002], [Poole 2004], [Yap 2005], [Braithwaite u. Joyce 2005a], [Massol 2004], [Simons 2002], [Fowler 2006b]*

(d) Ein gemeinsames Dokumentenarchiv ist eine wichtige Grundlage für die Entwicklung. In mehreren Projekten wurden sehr gute Erfahrungen mit Wikis gemacht. Dort werden u. a. Anwendungsfälle, Story-Cards, Aufgaben, Fragen und Antworten, Metriken und Trackingdaten, Kalender sowie Blogs der Teams und der einzelnen Mitglieder mit persönlicher Vorstellung abgelegt. Einheitliche Standards, Quelltextkonventionen, Prüflisten und Beispiele erleichtern eine teamübergreifende Zusammenarbeit.

In mehreren Projekten wird der Quelltext als hauptsächliches Kommunikationsmedium zwischen den Entwicklungsorten genutzt [Braithwaite u. Joyce 2005a, S. 187]: "Use the code base as a communications medium between sites. Converse with remote colleagues via the codebase. Express problems as failing tests in a suite outside the build, express design ideas as working code in a scratch area of the repository." *[Jensen u. Zilmer 2003], [Kussmaul et al. 2004], [Poole 2004], [Braithwaite u. Joyce 2005a], [Fowler 2006b]*

(w) Als unterstützende technische Systeme können neben dem Versionsverwaltungssystem mit gemeinsamem Archiv auch Application Sharing[2] und virtuelle Whiteboards genutzt werden. Damit wird entferntes Programmieren in Paaren, verteiltes Bearbeiten von Entwurfsskizzen u. Ä. möglich. *[Danait 2005], [Kircher et al. 2002], [Yap 2005], [Braithwaite u. Joyce 2005a], [Craddock u. Singhal 2006]*

Einbeziehung des Kunden

(o) Zusätzlich zu den fachlichen Analysten on-site werden auch in offshore arbeitenden Teams fachliche Experten ausgebildet. Alle Gruppen kooperieren intensiv. Anfangs kommunizieren nur die Onshore-Analysten mit dem Kunden und bereiten Anforderungen für die Offshore-Teams auf. Im Projektverlauf übernehmen die Offshore-Analysten eine wichtigere Rolle. Wenn sie sich in die Fachlichkeit eingearbeitet haben, können sie bei Bedarf auch direkt mit dem Kunden Kontakt aufnehmen und Fragen klären. Offshore-Personal kann auch die Rolle der Kunden-Stellvertreter für das Offshore-Team übernehmen. In einigen Projekten wurden damit recht gute Erfahrungen gesammelt.

Es ist für die Kommunikation mit entfernt arbeitenden Teams hilfreich, wenn die Anforderungen während einer Iteration unverändert bleiben und Anpassungen in die nächste Iteration verschoben werden [Kussmaul et al. 2004, S. 150]: "Minimizing requirement changes during a sprint is even more important with a distributed team, since it can be more difficult to agree on the scope, monitor the implementation, and

[2] Beim Application Sharing wird anderen Nutzern an entfernten Computern Zugriff auf Programme und Daten eines Host-Computers gewährt. Damit lassen sich z. B. Dokumente gemeinsam bearbeiten.

ensure that nothing is forgotten." *[Kussmaul et al. 2004], [Craddock u. Singhal 2006], [Simons 2002]*

(d) Mockups – mit einfachen Mitteln erstellte Prototypen der Benutzungsoberfläche – können genutzt werden, um die Anforderungen zu übermitteln. Sie werden unter Einbeziehung des Kunden on-site entwickelt und dienen dann entfernt arbeitenden Entwicklern als Vorlage. *[Danait 2005], [Kussmaul et al. 2004]*

(w) Bei der Kommunikation von entfernt arbeitenden Teams mit dem Kunden können Videokonferenzsysteme genutzt werden. Application Sharing kann auch bei der gemeinsamen Arbeit an Anforderungen und Aufgaben genutzt werden. *[Danait 2005], [Kircher et al. 2002], [Braithwaite u. Joyce 2005a], [Craddock u. Singhal 2006]*

Ausbildung und Kenntnisse

(o) Schon vor dem eigentlichen Projektstart sollte die Grundlage für erfolgreiches Arbeiten gelegt werden, indem Mitarbeiter nicht nur nach technischen, sondern auch nach sozialen Fähigkeiten ausgewählt werden. Durch die verteilte Arbeit müssen eventuell gewohnte Arbeitsweisen aufgegeben oder angepasst werden. So sind z. B. oft eine Angleichung der Arbeitszeiten und Geschäftsreisen oder sogar ein zeitlich begrenzter Standortwechsel erforderlich. Anfangs sollte mit kleinen Teams gearbeitet werden. Die Mitarbeiterzahl sollte erst erhöht werden, wenn die Zusammenarbeit gut funktioniert. Es ist hilfreich, wenn erfahrene Coaches zur Verfügung stehen, insbesondere mit Offshoring-Erfahrung.

Solange entfernt arbeitende Teams fachlich und technisch noch nicht so erfahren sind, sollten sie nicht zu lange unabhängig arbeiten. In der ersten Phase kann auch die gemeinsame Verantwortlichkeit eingeschränkt werden. Offshore-Teams erhalten dann auf bestimmte Module nur lesenden Zugriff. Im späteren Projektverlauf können den Offshore-Teams dann mehr Verantwortung übertragen werden. Das kommt der Arbeitszufriedenheit der Offshore-Teams zugute. Wenn Aufgaben zu Offshore-Standorten verlagert werden können, ergeben sich durch niedrigere Lohnkosten auch Kostenvorteile. *[Jensen u. Zilmer 2003], [Kussmaul et al. 2004], [Kircher et al. 2002], [Braithwaite u. Joyce 2005a], [Massol 2004]*

(d) Die Kernarchitektur der Anwendung sollte von einem kleinen, erfahrenen Team on-site während der ersten Iteration entwickelt werden. So wird eine gute Ausgangsbasis für alle Entwickler und ggf. für die verteilte Entwicklungsarbeit geschaffen. Ein gemeinsames Rahmenwerk[3] kann zur Unterstützung ebenfalls sinnvoll sein. *[Kussmaul et al. 2004], [Yap 2005]*

[3] „Ein Rahmenwerk (Framework) ist eine Architektur aus Klassenhierarchien, die eine allgemeine generische Lösung für ähnliche Probleme in einem bestimmten Kontext gibt. Wiederverwendet werden dabei nicht einzelne Klassen, sondern die gesamte Konstruktion aus zusammenspielenden Komponenten. Ein Rahmenwerk gibt so den Kontrollfluss für die Anwendung vor." [Züllighoven 1998, S. 118]

Infrastruktur

(w) Eine funktionierende Infrastruktur hängt von vielen Faktoren ab, die oft nicht durch einzelne Projekte zu beeinflussen sind, wie z. B. die Anbindung eines Landes ans Internet oder die Verfügbarkeit von zuverlässigen Telefonverbindungen. Angepasste Anwendungen erleichtern jedoch den Umgang mit begrenzter Bandbreite und häufigem Ausfall. So gibt es beispielsweise spezielle Versionsverwaltungssysteme, die mit lokalen Zwischenspeichern (engl. proxies) arbeiten, die bei der Überbrückung von Netzwerkproblemen helfen.

Eine häufige Fehlerquelle sind unterschiedliche Hard- und Softwarekonfigurationen. Die Abhilfe besteht darin, bei allen Teams die gleichen Systeme einzusetzen. Das erfordert eventuell eine gewisse Einarbeitungszeit, z. B. bei englisch- statt deutschsprachigen Betriebssystemen und Entwicklungsumgebungen. *[Kircher et al. 2002], [Poole 2004], [Yap 2005], [Massol 2004], [Simons 2002], [Fowler 2006b]*

Projektmanagement und Qualitätssicherung

(o) Durch die verteilte Arbeit können weitere Rollen im Projekt sinnvoll sein. So sollte es onshore und offshore Projektleiter geben, die sich direkt abstimmen. Risiken lassen sich verringern, wenn die entwickelte Anwendung häufig ausgeliefert wird. Dadurch lassen sich der Entwicklungsstand und die noch zu implementierenden Anforderungen besser einschätzen. Außerdem werden Integrationsprobleme vermieden: “The value of continuous integration is that it solves small integration headaches immediately instead of trying to solve huge ones at the end of the projects. Configuration management of offshore development becomes almost a non-issue.” [Simons 2002] Zusätzlich kann ein regelmäßiges (z. B. wöchentliches) Treffen zur Risikoeinschätzung mit Teilnehmern aus allen Teams durchgeführt werden. Die Qualitätssicherung sollte keine Aufgabe sein, die nur onshore durchgeführt wird. Auch Offshore-Teams sollten daran beteiligt sein. Dazu gehören auch regelmäßige formelle Überprüfungen des Quelltextes. *[Danait 2005], [Kussmaul et al. 2004], [Craddock u. Singhal 2006], [Massol 2004], [Simons 2002]*

(w) Komponententests sollten zum Standard gehören. Es hilft, wenn ein gemeinsames Versionsverwaltungssystem eingesetzt wird, das diese Tests beim Einchecken automatisch ausführt. An dieser Stelle können auch weitere automatische Überprüfungen durchgeführt und Metriken erstellt werden. Die Ergebnisse sollten für alle Beteiligten genauso abrufbar sein wie Fehler und Diskussionspunkte. Deren Analyse erlaubt eine recht gute Abschätzung des Fortschritts und der Entwicklungsqualität des Projekts. *[Danait 2005], [Kussmaul et al. 2004], [Braithwaite u. Joyce 2005a], [Massol 2004], [Simons 2002]*

3.3.5 Absicherung der Ergebnisse

Obwohl nicht jedes Problem und jeder Lösungsansatz in allen Erfahrungsberichten genannt wird, zeigen alle einen recht ähnlichen Umgang mit agiler verteilter Entwicklung. Größere Unterschiede gibt es einzig bei der Frage, ob das Kommunikationsproblem eher mit einer Formalisierung und Beschränkung der Kommunikationswege und des Kommunikationsverhaltens oder mit einer weitgehenden Freiheit und Bereitstellung verschiedenster Kommunikationsmedien angegangen wird.

Es gibt nur wenige vergleichbare Studien zu agiler verteilter Entwicklung, die mehrere Projekte umfassen. Die Ergebnisse von zwei dieser Studien werden im Folgenden mit den in dieser Arbeit erzielten Ergebnissen verglichen.

Ramesh et al. haben eine Studie von drei US-amerikanischen Firmen veröffentlicht, die Offshoring nach Indien betreiben [Ramesh et al. 2006]. Sie haben jeweils semistrukturierte Interviews mit Managern, Entwicklern und Kundenvertretern in beiden Standorten durchgeführt und nach Methoden der Grounded Theory ausgewertet. Die Untersuchung führte zu fünf wesentlichen Problemen und zugeordneten agilen Lösungsansätzen:

1. **Ausgleich zwischen formellen Kommunikationsmechanismen und flexibler, informeller Kommunikation:** Als agile Maßnahmen zur Verbesserung der Kommunikation bieten sich eine Angleichung der Arbeitszeiten, informelle Kommunikation – aber über formelle Kanäle, abgestimmte Koordination der Onshore- und Offshore-Teams und die Möglichkeit, jederzeit zu kommunizieren, an. Zusätzlich sollte der Wissenstransfer unterstützt werden, z. B. durch gemeinsame Archive für Produkte und Prozesse.

2. **Ausgleich zwischen festgelegten und sich entwickelnden Qualitätsanforderungen:** Um dem Offshore-Team Freiheit zu geben, es aber trotzdem kontrollieren zu können, sollte das Onshore-Team bei der Qualitätssicherung beteiligt werden. Die informelle Kommunikation über wichtige Artefakte und Prozesse sollte durch geeignete Dokumente ergänzt werden.

3. **Ausgleich zwischen formellen und informellen Prozessen:** Zusätzlich zu den Maßnahmen, die schon bei 2. beschrieben sind, sollte der Prozess kontinuierlich angepasst werden. So wurden in zwei der untersuchten Firmen anfangs Planungsiterationen durchgeführt, um kritische Anforderungen endgültig festzulegen und die Kernarchitektur auszuarbeiten. In anderen Fällen wurde die Dokumentation der Anforderung flexibel gestaltet und teils durch die Beschreibung von Anwendungsfällen, teils durch die Definition von Akzeptanztests realisiert.

4. **Ausgleich zwischen formellen und informellen Abmachungen** und

5. **Fehlen von Zusammenhalt im Team:** In beiden Fällen muss Vertrauen aufgebaut werden. Als Maßnahmen werden häufige gegenseitige Besuche durch Manager, Entwickler, Kunden und Entscheidungsträger auf beiden Seiten sowie

eine geeignete Zusammenstellung der Teams, insbesondere mit Kollegen, die sich schon aus anderen Projekten kannten, vorgeschlagen.

Ramesh et al. kommen zu dem Schluss, dass eine umsichtige Anwendung agiler Methoden unverzichtbar ist, um mit den Schwierigkeiten der verteilten Entwicklung im Offshoring umgehen zu können.

Paasivaara und Lassenius haben in fünf Projekten die Anwendung iterativer und inkrementeller Entwicklungsmethoden untersucht [Paasivaara u. Lassenius 2004]. Dazu haben sie 29 semistrukturierte Interviews durchgeführt. Sie haben sechs Praktiken identifiziert:

1. **Abstimmung der Auslieferungen:** Alle Projektpartner sollten ihre Iterationen aufeinander abstimmen.
2. **Überprüfung von Entwurfsdokumenten und Quelltexten:** Durch Reviews wird sichergestellt, dass entfernt arbeitende Teams die Anforderungen richtig verstanden haben. Die Review-Ergebnisse liefern den Teams auch schnelles Feedback zur Qualität der erstellten Artefakte.
3. **Kommunikation steht im Vordergrund:** Es finden regelmäßige persönliche oder virtuelle Treffen aller Beteiligter statt. Ad-hoc-Kommunikation über E-Mail und Chat wird gefördert.
4. **Aufteilung der Arbeit:** Durch Koordinierung der Entwicklungsaufgaben werden der spätere Test und die Integration von Teilmodulen erleichtert.
5. **Spezielle Verhaltensweisen:** Das Vorgehen wird durch spezielle Verhaltensweisen an das spezifische Projektumfeld angepasst. Beispielsweise wurde in einem Projekt festgelegt, dass jederzeit ein Projektleiter für Entwickler ansprechbar sein muss.
6. **Häufige Auslieferungen:** Kleine Aufgaben wurden genau spezifiziert und dann umgesetzt. Anschließend wurde eine neue Version erstellt und geprüft, bevor die nächsten Aufgaben vergeben wurden.

Durch diese Praktiken wurden in den untersuchten Projekten eine Reihe von Vorteilen erreicht: Der Entwicklungsprozess wurde transparenter, die Entwickler erhielten schneller Feedback, die Anforderungen und der Prozess konnten flexibel geändert werden, Missverständnisse durch falsch verstandene Anforderungen wurden reduziert und umfangreiche Integrationen wurden vermieden. Paasivaara und Lassenius folgern aus der Untersuchung, dass iterative und inkrementelle Techniken gut für verteilte Projekte geeignet sind.

Ein Vergleich dieser Ergebnisse mit Tabelle 3.3 zeigt eine weitgehende Übereinstimmung der beschriebenen Themen in allen Studien. Insbesondere wird die Bedeutung von

Kommunikation zwischen allen Projektbeteiligten in allen Untersuchungen hervorgehoben. Ramesh et al. betonen die Abstimmung zwischen formellen und informellen Kommunikationsmechanismen, Prozessen und Abmachungen. Paasivaara und Lassenius haben Praktiken identifiziert, welche die verteilte Zusammenarbeit erleichtern und die Kommunikation zwischen den Teams verbessern sollen. Bei der Analyse in dieser Arbeit haben sich weitere Bereiche gezeigt, in denen sich durch die Verteilung Schwierigkeiten ergeben können, die in den anderen Studien nicht thematisiert werden. Dies sind u. a. die Ausbildung, Wissensvermittlung, Infrastruktur und Projektmanagement.

3.4 Kritische Reflexion der Erfahrungen

In diesem Abschnitt werden die bisher gewonnenen Ergebnisse ausgewertet, mit weiteren wissenschaftlichen Untersuchungen verglichen und einer kritischen Reflexion unterzogen.

3.4.1 Eignung agiler Methoden für verteilte Entwicklung

Durch Vergleich mit plangetriebenen verteilten Projekten kann herausgearbeitet werden, ob agile Methoden die Probleme bei verteilter Entwicklung und Offshoring mindern.

Ein erstes Anzeichen für den Nutzen von Agilität bei verteilter Entwicklung sind die positiven Erfahrungen in den analysierten Berichten. Von den beschriebenen neun unterschiedlichen Projekten wurden sechs von den Autoren als erfolgreich eingestuft. Eines wurde als Misserfolg angesehen, aus dem die Beteiligten trotzdem viel gelernt hätten. In zwei Berichten wurde der Erfolg der jeweiligen Projekte nicht beurteilt. In sieben Projekten wurden agile Methoden als hilfreich oder sehr hilfreich für verteilte Entwicklung bewertet. Zwei Berichte geben keine explizite Bewertung ab. Obwohl die Ergebnisse der Analyse der Berichte keinen objektiven Beleg für die Eignung agiler Methoden für verteilte Entwicklung darstellen, konnten positive Wirkweisen agiler Methoden aufgezeigt und konkrete Nutzen ermittelt werden.

Zur weiteren Auswertung werden die ermittelten Probleme und Lösungsansätze mit denen aus plangetriebenen verteilten Projekten verglichen. In Abschnitt 2.3.3 haben wir Herausforderungen der global verteilten Anwendungsentwicklung betrachtet. Im Modell von Kornstädt und Sauer sind die Herausforderungen in drei Ebenen angeordnet worden, siehe Abschnitt 2.6. Die grundlegenden Herausforderungen der ersten Ebene zeigen sich auch in den Erfahrungsberichten: „Geringe Motivation am Offshore-Standort“ findet sich beim Thema „Arbeitszufriedenheit“, „Teilen von Zielen und Wissen“ findet sich bei den Themen „Gemeinsame Vision“ und „Wissenstransfer“, „kulturelle Unterschiede“ findet sich beim gleichnamigen Thema, „Infrastrukturkonflikte“ findet sich beim Thema „Kommunikationsinfrastruktur“ und „Hard- und Softwareausstattung“ und die „hohe Fluktuation der Offshore-Belegschaft“ findet sich als ein Aspekt des Themas „Arbeitszufriedenheit“. Die dagegen ergriffenen Maßnahmen entsprechen im

Modell den Herausforderungen der zweiten Ebene. In der vorgenommenen Analyse gelten sie als Problemlösungsansätze.

Zu allen Herausforderungen der zweiten Ebene finden sich in den Analyseergebnissen konkrete agile Lösungsansätze. Der hohen Reisetätigkeit lassen sich gegenseitige Arbeitsbesuche und herumreisende Botschafter zuordnen. Kulturelles Training findet sich als ein Lösungsansatz, besonders wirkungsvoll in Verbindung mit einem gemeinsamen Kickoff-Treffen. Zusätzliche Planung kann sich z. B. in einer Synchronisierung von Arbeitszeiten, gemeinsamen Standup-Meetings und weiteren Treffen, Onshore- und Offshore-Projektleitern und umfangreichen Qualitätssicherungsmaßnahmen ausdrücken. Eine Harmonisierung der Infrastruktur ist auch für agile verteilte Entwicklung sehr sinnvoll.

Ähnlich verhält es sich mit den von Carmel ermittelten zentrifugalen Kräften bei verteilten Teams (siehe Abbildung 2.5). Interessant ist eine Betrachtung der sechs zentripetalen (d. h. zum Zentrum führenden) Kräfte, die Carmel diesen als Lösungen entgegensetzt [Carmel 1999], siehe auch Abbildung 3.2:

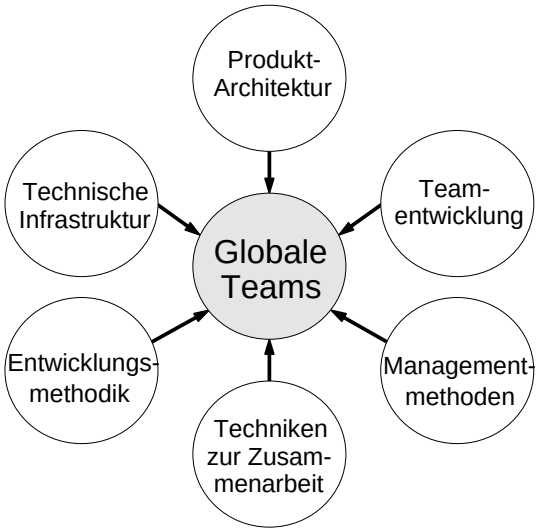


Abbildung 3.2: *Zentripetale Kräfte bei globalen Teams, nach [Carmel 1999, S. 81 ff.]*

Technische Infrastruktur: Eine verlässliche Netzverbindung mit ausreichend Bandbreite sowie eine einheitliche Hard- und Software ist eine unerlässliche Voraussetzung. Dies findet sich in der Auswertung der Erfahrungsberichte in den Lösungsansätzen Kommunikationsinfrastruktur und heterogene Hard- und Software direkt wieder.

Techniken zur Zusammenarbeit: Allgemeine Kollaborationstechniken wie Groupware-Systeme, E-Mail und Videokonferenzen und spezielle Werkzeuge für die Softwareentwicklung für das Konfigurations- und Projektmanagement u. Ä. sollten standardisiert sein und allen Beteiligten in ausreichender Anzahl zur Verfügung

stehen. Der Umgang mit diesen Systemen und Techniken sollte den Beteiligten vermittelt werden. In den in dieser Arbeit untersuchten agilen Projekten hat sich gezeigt, dass vor allem Wikis geeignet sind, da sie flexibel anpassbar und gut nutzbar sind. Versionsverwaltungssysteme sind für parallele Arbeiten am Quelltext unerlässlich.

Entwicklungsmethodik: Es sollte eine einheitliche Entwicklungsmethodik verwendet und allen Beteiligten vermittelt werden. Sie sollte im Laufe des Projekts an veränderte Bedingungen angepasst werden. Es ist dabei wichtig, dass alle Beteiligten dieselben Begriffe nutzen. In dieser Arbeit wurden mit Distributed Extreme Programming, Distributed Scrum und DSDSM-O drei agile Entwicklungsmethodiken vorgestellt und untersucht. Bei allen wird ein Vokabular für Rollen, Artefakte und Arbeitsschritte definiert. Darüber hinaus lassen sich fachspezifische Konzepte der Anwendungswelt über Metaphern für alle verständlich machen.

Produkt-Architektur: Carmel versteht darunter die Aufteilung von Aufgaben auf die Teams. Er unterscheidet dabei die Aufteilung nach Modulen, die Aufteilung nach Phasen im Entwicklungsprozess und integrierte Entwicklung mit täglicher Übergabe („Follow the Sun“), vgl. die Abschnitte 2.2.3 und 2.3.2. Die Möglichkeiten der Aufteilung von Aufgaben bei agiler global verteilter Entwicklung werden in Kapitel 5 detailliert diskutiert.

Teamentwicklung: Es werden eine Reihe von Möglichkeiten beschrieben, um persönliche Beziehungen und Vertrauen zwischen den Teams aufzubauen: gemeinsames Kickoff-Treffen, gemeinsames Feiern von Erfolgen, Vermittlung einer gemeinsamen Vision, formale Protokolle für die Kommunikation, Rotation der Beteiligten zwischen den Teams, Kulturtrainings u. Ä. Die Auswertung der Erfahrungsberichte hat ähnliche Lösungsansätze gezeigt.

Managementmethoden: Zu den Hauptaufgaben des Managements gehört es, die Organisation zu planen, Konflikte zu lösen und den Fortschritt zu messen und zu überwachen. Durch häufige Auslieferungen und ein gemeinsames Quelltext- und Dokumentenarchiv wird die Messung des Fortschritts vereinfacht. Gegenüber einem plangetriebenen Vorgehen wird das Management entlastet.

Die Gegenüberstellung der zentrifugalen und zentripetalen Kräfte zeigt, dass mit dem Einsatz agiler Methoden im Vergleich zu plangetrieben durchgeführten Projekten in der Regel die Flexibilität verbessert werden kann. Mit agilen Methoden konnten auch einige innovative Problemlösungsansätze entwickelt werden.

Mit diesen Ergebnissen lässt sich ein überwiegend positives Fazit zu verteilter Entwicklung mit agilen Methoden ziehen. Die Analyse von veröffentlichten Erfahrungsberichten im letzten Abschnitt hat gezeigt, dass es auch praktisch keinen Widerspruch zwischen Agilität und verteilter Entwicklung gibt. Die in Abschnitt 3.2.2 formulierten Erwartungen an agile verteilte Entwicklung konnten in der Mehrzahl der betrachteten Projekte recht gut umgesetzt werden. Die Analyse hat aber auch gezeigt, dass Agilität keine

Silver Bullet[4] für verteilte Entwicklungsprojekte darstellt. Die Techniken funktionieren nicht so optimal wie bei kleinen, nicht-verteilten Projekten.

Das recht positive Fazit deckt sich mit Einschätzungen, die in mehreren der analysierten Erfahrungsberichte geäußert werden. Martin Fowler fasst seine Erfahrungen der Überlegenheit von agilen gegenüber plangetriebenen Ansätzen bei verteilter Entwicklung treffend zusammen: "Our experience is that even if an agile approach suffers from the communication difficulties of offshore, it's still better than a documentation-driven approach." [Fowler 2006b]

Welche der agilen Methoden eignet sich besonders gut für verteilte Entwicklung? Es ist auf der Grundlage der untersuchten Erfahrungsberichte nicht möglich, die Vor- und Nachteile einzelner Methoden abschließend zu bewerten. Es lässt sich aber festhalten, dass Distributed Extreme Programming (siehe Anhang A.1.2) oder zumindest einzelne Techniken daraus in der Mehrzahl der betrachteten Projekte eingesetzt wurde. Einige DXP-Techniken wie verteiltes Programmieren in Paaren und Kunden-Stellvertreter wurden in mehreren Berichten als wertvolle Lösungen hervorgehoben. Da XP auch in nicht-verteilten Projekten weitverbreitet und damit vielen Entwicklern vertraut ist (vgl. Tabelle 3.2) und DXP nicht sehr davon abweicht, scheint es für viele Projektverhältnisse geeignet zu sein.

In den nächsten Abschnitten werden zwei relevante Problembereiche beschrieben, in denen sich Schwierigkeiten der global verteilten Entwicklung besonders bemerkbar machen und für die bisher noch keine befriedigenden Lösungen gefunden wurden: der Themenkomplex Kommunikation, Koordination und Kooperation (Abschnitt 3.4.4) sowie Möglichkeiten des Ausgleichs zwischen enger und flexibler Steuerung der Zusammenarbeit (Abschnitt 3.4.5). Zuvor wird untersucht, welche Probleme allgemein bei global verteilter Entwicklung auftreten und welche Probleme beim Offshoring verstärkt werden oder sich zusätzlich stellen. Daran anschließend werden auffällige Unterschiede zwischen kleinen und großen Projekten und Unternehmen (Abschnitt 3.4.3) thematisiert.

3.4.2 Verteilte Entwicklung im Vergleich zu Offshoring

Offshoring wurde in Abschnitt 2.2 als eine mögliche, häufig vorkommende Organisationsstruktur für global verteilte Entwicklung eingeführt, bei der mehrere rechtlich unabhängige Unternehmen beteiligt sind.

Aufgrund der Analyse der Erfahrungsberichte können jetzt Aussagen über charakteristische Probleme von Agilem Offshoring gegenüber agiler verteilter Entwicklung im Allgemeinen getroffen werden. In Abschnitt 2.2.4 wurden bereits charakterisierende

[4] Die Verwendung des Ausdrucks „Silver Bullet" in der Softwaretechnik geht auf einen weitverbreiteten Artikel von Brooks zurück, in dem er die inhärenten Schwierigkeiten der Softwareentwicklung beschreibt [Brooks 1987].

Eigenschaften der Entwicklung von Anwendungen im Offshoring herausgearbeitet: Die Punkte „Anwendungsentwicklung“, „Global verteilte Teams“ und „Kulturelle Unterschiede“ sind beiden Varianten gemein. Eine Outsourcing-Beziehung besteht beim Agilen Offshoring immer, bei agiler verteilter Entwicklung nur in speziellen Fällen.

Zur genaueren Untersuchung werden zunächst Dimensionen der Verteilung und ihre Auswirkungen auf die Zusammenarbeit der Teams aus der Literatur abgeleitet. Anschließend werden die Ergebnisse mit denen aus der Analyse der Erfahrungsberichte abgeglichen und Schlüsse zum Verhältnis von Agilem Offshoring zu agiler global verteilter Entwicklung gezogen.

Dimensionen der Verteilung

Die Verteilung der Beteiligten bei global verteilter Entwicklung lässt sich auf verschiedene Art und Weise untersuchen. In dieser Arbeit wird der Versuch unternommen, für die Analyse der Verteilung geeignete Dimensionen aus mehreren Quellen zu konsolidieren und voneinander abzugrenzen. Dabei werden folgende Dimensionen mit einbezogen:

- Mittels einer Literaturauswertung wurden in [Gumm 2006] vier Dimensionen gefunden: physical (or geographical) distribution (in dieser Arbeit: *geografische Verteilung*), organizational distribution (*organisatorische Verteilung*), temporal distribution (*zeitliche Verteilung*) und distribution among stakeholder groups (*Verteilung nach Funktionsbereichen*).
- Fünf Dimensionen, als Boundaries bezeichnet, wurden in einer Feldforschung in mehreren Firmen identifiziert [Espinosa et al. 2003]: geographical boundary (*geografische Verteilung*), functional boundary (*Verteilung nach Funktionsbereichen*), temporal boundary (*zeitliche Verteilung)*, identity boundary (*persönliche Verteilung*) und organizational boundary (*organisatorische Verteilung*).
- In [Altmann u. Pomberger 1999] werden drei Dimensionen aus Sicht der kooperativen Softwareentwicklung beschrieben: aufgabenbezogene Verteilung (*Verteilung nach Funktionsbereichen*), zeitliche Verteilung im Sinne nebenläufiger Teilprojekte (*Verteilung nach Teilprojekten*) und räumliche Verteilung (*geografische Verteilung*).

Bei einer Analyse der beschriebenen Dimensionen zeigt sich, dass einige Dimensionen das Offshoring und global verteilte Softwareentwicklung charakterisieren, während andere auch bei nicht-verteilten Softwareentwicklungsprojekten eine Rolle spielen können.

Typische Dimensionen für global verteilte Entwicklung: Für global verteilte Entwicklung typisch sind die geografische und zeitliche, für Offshoring zusätzlich die organisatorische Verteilung.

Die **geografische Verteilung** ist die bei verteilter Entwicklung am meisten diskutierte Dimension. Wie schon in Kapitel 2 hervorgehoben wurde, stellt sie einen Hauptunterschied zu anderen Formen der kooperativen Softwareentwicklung dar.

Zeitliche Verteilung wird bei global verteilter Entwicklung typischerweise durch unterschiedliche Zeitzonen hervorgerufen. Sie kann jedoch auch durch unterschiedliche Arbeitsrhythmen (z. B. den Arbeitsbeginn oder die Mittagspause) oder Teilzeitarbeit bedingt sein.

Beim Offshoring entsteht eine starke **organisatorische Verteilung** durch die Trennung in Klient und Dienstleister. Darüber hinaus können Beteiligte auch innerhalb dieser Gruppen in verschiedenen Unterorganisationen arbeiten. Dies geschieht häufig, wenn für ein Softwareentwicklungsprojekt eine von der Linienorganisation abweichende Organisationsform gewählt wird.

Allgemeinere Dimensionen: Die Verteilung nach Funktionsbereichen und nach Teilprojekten sowie die persönliche Verteilung sind allgemeinere Dimensionen.

Eine **Verteilung nach Funktionsbereichen** liegt eigentlich immer vor, wenn sich ein Team aus Akteuren mit unterschiedlichen Aufgaben und Funktionen zusammensetzt. Das Team besteht dann aus mehreren Gruppen von Interessenvertretern (engl. stakeholder) [Gumm 2006] mit unterschiedlichen Vorstellungen und Herangehensweisen, Ausbildung und Wissen. Bei einigen agilen Methoden wird hingegen eine homogene Verteilung von Aufgaben und Funktionen angestrebt.

Wenn innerhalb eines Softwareentwicklungsprojekts mehrere Unterprojekte mit unterschiedlichen Teilzielen nebenläufig bearbeitet werden, spricht man von einer **Verteilung nach Teilprojekten**. Problematisch kann es werden, wenn zwischen den Teilprojekten Abhängigkeiten bestehen, die eine Synchronisierung und Koordination der Teilprojekte erforderlich machen.

Unter einer **persönlichen Verteilung** versteht man, dass ein Projektbeteiligter gleichzeitig auch an anderen Projekten arbeitet und somit seine Zeit und Aufmerksamkeit nicht vollständig einem Projekt widmen kann.

Auswirkungen auf die Zusammenarbeit: Es gibt mehrere Untersuchungen und Erfahrungen aus Fallstudien, wie die einzelnen Dimensionen der Verteilung die Zusammenarbeit beeinflussen. Die genauen Auswirkungen der Verteilung und insbesondere das Zusammenspiel verschiedener Dimensionen sind jedoch noch nicht im Detail verstanden. Sie sind Gegenstand von laufenden Forschungsarbeiten. Im Folgenden werden wichtige Forschungsergebnisse (Problemanalysen und Lösungsvorschläge) für die für globale Softwareentwicklung typischen Dimensionen genannt.

Die Kommunikation innerhalb von Teams nimmt mit zunehmender geografischer Entfernung zwischen den Teammitgliedern stark ab. Die Folgen können abgemildert werden,

wenn die Projektleitung mögliche Kommunikationswege früh im Projekt etabliert und durch Regelungen für häufige und regelmäßige Kommunikation sorgt [Cummings 2007]. In einer Befragung von zehn Managern wurde festgestellt, dass die Produktintegration am meisten unter der geografischen Verteilung leidet [Herbsleb u. Grinter 1999a]. In Abschnitt 3.3.4 haben wir gesehen, dass die Auswirkungen der Entfernung durch Reisen und temporären Einsatz im jeweils anderen Team reduziert werden können.

Die Auswirkungen von zeitlicher Distanz auf die Koordination sind schlechter verstanden. Neben der Arbeit in unterschiedlichen Zeitzonen spielen auch länderspezifische und regionale Eigenheiten eine Rolle. So unterscheiden sich z. B. die Arbeitszeiten in Spanien durch die dort übliche lange Mittagspause von denen in anderen Ländern Europas [Espinosa u. Carmel 2003]. Auch innerhalb Deutschlands gibt es Unterschiede, beispielsweise bei den Feiertagsregelungen und bei Urlaubszeiten. Dies erschwert eine Abstimmung der Arbeitszeiten. Wie wir in Abschnitt 2.3.2 schon gesehen haben, können mit einer „Follow the Sun“-Entwicklung durch unterschiedliche Zeitzonen jedoch auch Vorteile entstehen.

Gerade die Betrachtung unterschiedlicher Zeitzonen zeigt auch, dass die Untersuchung einzelner Dimensionen wegen der Abhängigkeiten zu anderen Dimensionen schwierig sein kann. Mit der geografischen Trennung geht im Allgemeinen auch eine Arbeit in verschiedenen Zeitzonen einher. Espinosa, Nan und Carmel haben in [Espinosa et al. 2007] versucht, die geografische und zeitliche Dimension voneinander zu trennen, und Unterschiede in West-Ost-verteilter, zeitzonenübergreifender und Nord-Süd-verteilter Arbeit in derselben oder einer benachbarten Zeitzone verglichen. Das überraschende Ergebnis: Die Geschwindigkeit der Aufgabenerledigung ist nicht proportional zu der Länge der Überlappung der Arbeitszeiten. Dies lässt sich vielleicht damit begründen, dass es sich um eine Laborstudie und keine empirische Untersuchung handelte. Es zeigt sich, dass hier noch erheblicher Forschungsbedarf besteht, insbesondere in realen Projekten.

Für ein Verständnis der organisatorischen Verteilung wird nicht nur im IT-Bereich häufig auf Conway's Law zurückgegriffen. Dieses Gesetz wurde 1968 erstmals veröffentlicht [Conway 1968]. Der Autor hat es 2001 wie folgt zusammengefasst: “Any organization which designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure.” [Conway 1968, Vorwort] Coplien hat dieses Gesetz für agile Entwicklung aufgegriffen und im Organisationsmuster „Organization Follows Location“ beschrieben [Coplien u. Harrison 2004]. Dieses Prinzip geht von zwei wesentlichen Eigenschaften verteilter Entwicklung aus: zum einen, dass Mitarbeiter naturgemäß lokalen Managern gegenüber loyaler sind als entfernten. Zum anderen, dass bei einer Arbeitsteilung zwischen zwei Standorten, bei der die Aufgaben nicht klar getrennt werden können, eine Partei die Leitung innehat, was naturgemäß zu Verstimmungen bei der anderen Partei führen kann. Coplien zieht daraus den Schluss, dass eine vernünftige Zuweisung von Aufgaben und Rollen bei verteilten Teams von großer Wichtigkeit ist.

Folgerungen aus der Analyse der Erfahrungsberichte

Ein Unterschied zwischen agiler global verteilter Entwicklung und Agilem Offshoring besteht in der Praxis in den Aufgaben und der Anzahl der Teams. In den Erfahrungsberichten, die eher Agilem Offshoring zuzuordnen sind, gab es in der Regel zwei Teams mit spezifischen Aufgabenbereichen. Das Onshore-Team war eher für die Erhebung der Anforderungen und die Kommunikation mit dem Kunden zuständig, das Offshore-Team eher für die Umsetzung. Bei agiler verteilter Entwicklung allgemein gab es dagegen mehrere Teams mit ähnlichen Aufgaben. Mehr Teams könnten erhöhten Abstimmungsbedarf und größere Unterschiede im Vorgehen der Teams, z. B. der Umsetzung agiler Praktiken, ergeben. Den Berichten ist jedoch kein Hinweis zu entnehmen, dass sich durch mehr als zwei Teams weitere Arten von Problemen ergeben haben.

Auch wenn die verschiedenen Ausprägungen nicht prinzipiell mit einem Ansatz verbunden sind (so findet man z. B. auch Offshoring mit mehr als zwei Teams oder global verteilte Entwicklung innerhalb einer Organisation mit Teams, die unterschiedliche Aufgaben haben), so scheinen sie in der Praxis doch sehr häufig so aufzutreten.

Durch die organisatorische Verteilung beim Agilen Offshoring können Schwierigkeiten der verteilten Entwicklung verstärkt auftreten. Diese Schwierigkeiten lassen sich in zwei Gruppen einteilen. Zum einen sind es mögliche Probleme, die bei der Kommunikation über Organisationsgrenzen auftreten können, beispielsweise die Etablierung von Teamgeist und einer gemeinsamen Vision und den Umgang mit kulturellen Unterschieden. Zum anderen Probleme durch den fehlenden gemeinsamen organisatorischen Hintergrund: Die Einstellung zu Agilität und die Umsetzung agiler Praktiken, die Entwurfs- und Architekturkenntnisse und das allgemeine Ausbildungsniveau können größere Unterschiede aufweisen, wenn die Teams aus unterschiedlichen Organisationen stammen.

Mit Zeit und Geld sind viele dieser Probleme behebbar, aber die meisten Projekte – insbesondere kleine – haben diese Möglichkeiten nicht. Die Auswirkungen unterschiedlich großer Projekte werden im nächsten Abschnitt diskutiert.

3.4.3 Unterschiede zwischen kleinen und großen Projekten

Bei der Analyse der Erfahrungsberichte zeigte sich, dass Unterschiede in der Größe der Projekte deutliche Auswirkungen auf die nutzbaren Lösungsalternativen haben. In größeren, länger laufenden Projekten können andere Mittel eingesetzt werden als in Projekten kleineren und mittleren Umfangs. Diese Unterschiede sollen an dieser Stelle aufgegriffen und mit weiteren wissenschaftlichen Erkenntnissen abgeglichen werden.

Es ist schwierig, die Projektgröße in Kategorien einzuteilen, weil viele Faktoren eine Rolle spielen. Als Anhaltspunkt soll die im V-Modell 97 [IABG 1997] verwendete Einteilung dienen. Dort werden die Faktoren „Aufwand in Personenjahren“ und „Anzahl der Mitarbeiter“ berücksichtigt, siehe Tabelle 3.4.

Projektgröße	Personenjahre		Mitarbeiterzahl
klein	$\leq 0,5$	oder	≤ 2
mittel	≤ 5	oder	≤ 5
groß	> 5	oder	> 5

Tabelle 3.4: *Projektgrößeneinstufung im V-Modell 97, nach [IABG 1997, T.3.1.1]. Bei konfligierender Einordnung aufgrund von Personenjahren und Mitarbeiterzahl ist die größere Projektgröße zu wählen.*

Bei großen, lang laufenden Projekten sind umfangreichere Anfangsinvestitionen möglich. Ein langwierigerer Beginn kann im späteren Projektverlauf ausgeglichen werden. Wenn viele Prozesse ausgelagert werden, ist auch der Aufbau eines eigenen Offshore-Zentrums möglich. Dies lohnt sich für kleine und mittlere Projekte nicht. Kleinere Projekte müssen früher produktive Arbeitsweisen erreichen und wirtschaftlich sein. Die Kommunikation und Koordination von verteilten Teams muss effizient funktionieren, sonst können die niedrigeren Personalkosten in Offshore-Ländern die erhöhten Kommunikationsaufwände nicht ausgleichen. Die BITKOM schreibt in ihrem „Leitfaden Offshoring“ dazu: „Ein Offshore-Projekt muss eine gewisse Größe und Dauer haben, um die erhöhten Kommunikations- und Projektinitiierungskosten durch die niedrigeren Lohnkosten auszugleichen. Ein durchschnittliches Offshore-Projekt – mit den Projektphasen Setup, Design, Build, Deploy, Run, Maintenance – erreicht den Break-Even in der Regel erst nach Abschluss der Build-Phase. Dafür verantwortlich sind vor allem die Overheadkosten, die bei einer Auslagerung direkt zu Beginn entstehen (u. a. Kosten für Reisen, Kommunikation, Management etc.).“ [Boos et al. 2005, S. 25]

In der Mehrzahl der untersuchten Erfahrungsberichte finden sich keine genauen Zahlen über die Projekt- und Unternehmensgröße. In einigen werden jedoch spezifische Probleme kleinerer und mittlerer Projekte genannt. Kussmaul et al. kommen zu einer leicht nachvollziehbaren Einschätzung [Kussmaul et al. 2004, S. 150]: “Avoid projects that are too small to amortize the overhead required for a effective distributed team. Very small projects are best done by local teams, unless an offshore team already has direct expertise.” Die Projektgröße hat direkte Auswirkungen auf die Problemlösungsansätze. Matt Simons stellte schon 2002 fest [Simons 2002]: “Smaller projects might not be able to recover the expenses associated with rotating resources through cost savings on offshore development and might have to figure out different ways of collaborating and communicating.” Neben den teuren gegenseitigen Besuchen und persönlichen Treffen sollten auch Kosten für den Aufbau einer geeigneten Infrastruktur, für die Substitution von direkter Kommunikation durch Videokonferenzsysteme u. Ä., die technische und fachliche Ausbildung der Mitarbeiter sowie Projektmanagement und Qualitätssicherung berücksichtigt werden. Zusätzlich zu den direkten Kosten ist die benötigte Zeit für den Aufbau und den Betrieb der technischen Infrastruktur zu berücksichtigen.

Die Größe von verteilten Entwicklungsprojekten hängt mit der Größe des auslagernden Unternehmens zusammen. Die Unternehmensgröße sagt jedoch nicht unbedingt etwas über die Komplexität von Projekten aus. Bei kleinen Unternehmen, in denen jeder Beteiligte viel Verantwortung trägt, kann Offshoring komplexer sein als bei Arbeitsteilung in großen Unternehmen. Analog zur Projektgröße sei hier als Anhaltspunkt eine Definition der Größe von Unternehmen angegeben. Wir verwenden die etablierte Klassifikation von kleinen und mittleren Unternehmen (KMU) der Europäischen Kommission, siehe Tabelle 3.5.

Größe	**Mitarbeiterzahl**	**Jahresumsatz**		**Jahresbilanzsumme**
mittel	≤ 250	≤ 50 Mio. €	oder	≤ 43 Mio. €
klein	≤ 50	≤ 10 Mio. €	oder	≤ 10 Mio. €
kleinst	≤ 10	≤ 2 Mio. €	oder	≤ 2 Mio. €

Tabelle 3.5: *Unternehmensgrößeneinstufung der Europäischen Kommission, nach [Europäische Kommission 2006, S. 14]. Bei konfligierender Einordnung aufgrund von Jahresumsatz und -bilanzsumme ist die kleinere Unternehmensgröße zu wählen.*

Eric Olsson hat in [Olsson 2004] besondere Probleme für europäische KMU im Zusammenhang mit Offshoring zusammengestellt. Hohe Kosten und ungewisse Herausforderungen stellen ein beträchtliches Risiko dar. Für KMU gibt es oft Probleme bei der Finanzierung von Offshoring-Projekten sowie der Anwerbung qualifizierter Mitarbeiter, die geeignet sind, Offshoring-Projekte mit ihren besonderen Anforderungen zu managen und das Offshore-Team zu steuern. Informelle Abstimmungsprozesse sind für KMU sehr wichtig. Komplexere Groupware-Werkzeuge spielen für sie nur eine untergeordnete Rolle [Boden et al. 2009]. Sprach- und Kulturbarrieren sind für KMU besonders problematisch, da für sie Flexibilität und Agilität bei der Entwicklung besonders wichtig sind. Sie sind daher von Problemen kulturell-sprachlicher Divergenzen besonders betroffen [Ågerfalk u. Fitzgerald 2006]. Dies ist ein Grund dafür, dass KMU seltener als große Unternehmen Near- und Offshoring betreiben. Eine Studie aus dem Jahr 2007 hat ergeben, dass von den Unternehmen mit einem Jahresumsatz von bis zu 10 Mio. €, die Outsourcing planen, 80% einen Dienstleister suchen, der on-site oder onshore arbeitet, 15% einen Nearshoring- und nur 5% einen Offshoring-Dienstleister [Steria Mummert Consulting AG 2007]. Dies hängt auch damit zusammen, dass KMU die Größe und damit meist auch die politische Durchsetzungskraft fehlt, um gute Investitionsbedingungen und Schutz vor bürokratischen Schikanen in geografisch entfernten Ländern zu erreichen. Instabile rechtliche Rahmenbedingungen führen zu Zusatzkosten für Beratung, Rechtsstreitigkeiten, Korruptionsschutz u. Ä., die für KMU besonders ins Gewicht fallen [Krick u. Voß 2005].

Albayrak, Gadatsch und Olufs schätzen, dass sich aus diesen und weiteren Gründen Offshoring-Projekte erst ab einer Gesamtbelegschaft von etwa 300 Mitarbeitern wirtschaftlich lohnen [Albayrak et al. 2007]. Diese Einschätzung scheint sehr pessimistisch

zu sein, da sich für KMU durch Offshoring auch neue Möglichkeiten ergeben. Die Nutzung von Offshoring-Ressourcen kann ihnen eine Skalierung ohne langwierige Personalakquisition und langfristige Bindung an Mitarbeiter erlauben, sodass sie unter Umständen auch mit wesentlich größeren Unternehmen konkurrieren können [Olsson 2004]. Für KMU stellt eine mögliche Kostenreduzierung einen großen Anreiz für Offshoring-Projekte dar. Außerdem können sie besonders von den Erfahrungen externer Dienstleister mit konkreten Werkzeugen und Projekten profitieren. Sie verfolgen beim Offshoring andere Strategien als größere Firmen. Charakteristisch ist, dass sie die Offshore-Arbeit möglichst genau steuern und kontrollieren möchten. Es ist ihnen sehr wichtig, vertrauensvolle soziale Beziehungen zwischen den beteiligten Personen herzustellen [Sieber 2006].

Carmel und Nicholson stellen fest, dass die Forschung bisher hauptsächlich große Unternehmen im Fokus hatte: "The prior research in onshore and offshore software sourcing has improved our understanding of the management of software outsourcing and the additional complexities presented in offshore relationships. However, most of this research has centered on large organizations that have the internal resources to address the problems of managing across time and space." [Carmel u. Nicholson 2005, S. 34] Hinzu kommt, dass kleine Unternehmen in der Regel keine formalisierten Entwicklungsprozesse einsetzen, sondern individuell und flexibel auf den Kunden eingehen. Informelle Kommunikation hat daher eine große Bedeutung. Von den Mitarbeitern wird hohe Selbstständigkeit erwartet [Nett et al. 2004]. Formale Spezifikationsmethoden haben wenig Aussicht darauf, akzeptiert zu werden; agile Methoden bieten sich an.

Die Forschungslücke und die besondere Eignung agiler Methoden für KMU machen Offshoring-Projekte von KMU für diese Arbeit interessant. In der weiteren Diskussion werden daher vor allem Lösungsansätze für kleine und mittlere Projekte von KMU behandelt.

3.4.4 Kommunikation, Koordination und Kooperation

Wie schon festgestellt wurde, sind die notwendigen Einschränkungen bei der Kommunikation und damit einhergehend die Koordinations- und Kooperationsschwierigkeiten die Hauptschwierigkeiten bei verteilten Entwicklungsprojekten. Die hier durchgeführte Analyse von Erfahrungsberichten hat dies auch für den Einsatz von agilen Methoden bestätigt. Das ist besonders wichtig, da offene Kommunikation und ständiges Feedback eine wichtige Basis von Agilität darstellen.

Christiansen hat den Umgang mit dem Kommunikationsproblem in 22 unterschiedlich großen, plangetriebenen oder agilen Projekten durch Interviews mit Managern und Entwicklern untersucht. Er kommt ebenso zu dem Schluss, dass es sich lohnt, auf gute Kommunikation Wert zu legen: "The research material show that companies with emphasized communication in their software processes ran into fewer problems,

whether technical or organizational." [Christiansen 2007] Die Studie empfiehlt folgende Grundsätze zur Unterstützung der Kommunikation:

1. **Legen Sie Wert auf synchrone Kommunikation!** Mit synchroner Kommunikation erhält man schneller Feedback. Das gegenseitige Verständnis wird erhöht. Problematisch könnte sein, dass sich viele Leute scheuen, über Videokonferenzen und Telefon direkt zu kommunizieren, auch aufgrund von Sprachproblemen.

2. **Versuchen Sie, andere Kulturen zu verstehen!** Kulturelle Unterschiede sollten verstanden und akzeptiert werden. Durch die beschränkte Kommunikationsbandbreite bei verteilter Arbeit kann es jedoch schnell zu Missverständnissen kommen.

3. **Fördern Sie Sprachkenntnisse!** Zumindest die wichtigsten Informationsverteiler sollten gute mündliche Sprachkenntnisse haben, um Irrtümer zu vermeiden.

4. **Rotieren Sie Teammitglieder!** Onshore-Mitarbeiter sollten für einige Zeit auch offshore arbeiten und umgekehrt.

5. **Setzen Sie Artefakte richtig ein!** Der richtige Umgang mit Entwicklungsartefakten wie Quelltexten, Prototypen, Testdokumenten u. Ä. hilft bei Lernprozessen und bei der Kommunikation.

6. **Vereinheitlichen Sie die IT-Infrastruktur!** Abweichende Hard- und Softwareausstattungen führen zu Problemen, genauso wie nicht funktionierende Telefonleitungen.

7. **Spezifizieren Sie Anforderungen sorgfältig!** Anforderungsspezifikationen enthalten viel implizites Wissen. Dieses Wissen muss Offshore-Entwicklern durch ausführliche Dokumente oder Meetings vermittelt werden.

Diese Empfehlungen decken sich mit den ermittelten Lösungsansätzen aus der in dieser Arbeit durchgeführten Studie. Die Punkte 1, 2, 4, 6 und 7 finden sich fast identisch. 3 und 5 ergeben sich schlüssig aus den beobachteten Problemen. Die Frage, wie sich ausführliche Dokumentation mit einem agilen Vorgehen verbinden lässt, wird im nächsten Abschnitt wieder aufgegriffen.

Eine bei [Christiansen 2007] und in weiteren wissenschaftlichen Untersuchungen (vgl. z. B. [Balaji u. Ahuja 2005] und [Carmel u. Tjia 2006]) wichtige und nicht befriedigend behandelte Frage ist die des Wissenstransfers zwischen den verteilten Teams. Dabei ist von großer Bedeutung, welche Art von Informationen ausgetauscht werden müssen. Es scheint in der Praxis keine gute Lösung zu sein, Anforderungen onshore zu spezifizieren und dann offshore nur umsetzen zu lassen (siehe „Probleme plangetriebener verteilter Projekte“ auf Seite 44, die Themen „Zusammenarbeit zwischen den Entwickler-Teams“ und „Einbeziehung des Kunden“ in Abschnitt 3.3.3 und Punkt 7 bei [Christiansen 2007]). Wenn jedoch onshore und offshore entwickelt wird, stellen sich weitere Probleme bei der Zusammenarbeit, wie zuvor beschrieben. Neben den Kundenanforderungen

muss zusätzliches Wissen ausgetauscht werden, beispielsweise über die Anwendungsarchitektur, Komponenten und Schnittstellen, verwendete Technologien, Bibliotheken und Quelltextkonventionen.

Die Probleme bei der Kommunikation und bei der Zusammenarbeit zwischen den Entwickler-Teams legen eine genauere Untersuchung der Aufgaben der verteilten Teams, ihrer Schnittstellen untereinander und mit dem Kunden und der Koordinierung der Zusammenarbeit der Teams aus softwaretechnischer Sicht nahe. Diese Untersuchung ist daher Thema des nächsten Kapitels.

3.4.5 Flexible oder enge Steuerung?

Eine wichtige Entscheidung bei agiler verteilter Entwicklung besteht darin, ob die Teams, insbesondere die Offshore-Teams, flexibel und eigenbestimmt arbeiten dürfen, wie es in herkömmlichen agilen Entwicklungsprojekten angestrebt wird, oder ob sie enger gesteuert werden sollten, z. B. durch formale Kommunikationswege. Nach einer Untersuchung von Hinds und McGrath bestehen wahrnehmbare Unterschiede in der Effektivität von Organisationsstrukturen zwischen nicht-verteilten und global verteilt arbeitenden Teams [Hinds u. McGrath 2006].

Bei einer **engen Steuerung** sind die Prozesse und die erwünschten Kommunikations- und Berichtswege genau festgelegt. Die Teams haben nur wenige Freiheiten bei ihrer Arbeit. Die Vorgabe der Anforderungen, die Abnahme der Ergebnisse und der gesamte Informationsaustausch erfolgen strukturiert und geregelt. Dadurch lässt sich die Arbeit besser nachverfolgen. Das Projektmanagement kann den Status des Projekts genauer ermitteln.

Zu Beginn des Abschnitts 3.3.5 wurde schon festgestellt, dass es große Unterschiede in den berichteten Erfahrungen zu der Frage gibt, inwieweit eine Formalisierung der Kommunikation und der ausgetauschten Nachrichten und Dokumente für agile verteilte Entwicklung generell notwendig ist. Die Arbeit ist klar zwischen den Teams aufgeteilt. Um den Aufwand für die Kommunikation zu beschränken, muss analysiert werden, welches Wissen für jede Aufgabe nötig ist. Dadurch kann der Wissenstransfer besser geplant werden. Die Kommunikation mit dem Kunden ist klar geregelt und die Projektleitung hat feste Ansprechpartner.

Viele Punkte sprechen für eine **Flexibilisierung und weitgehende Freiheit** bei der Steuerung. Die Teams organisieren ihre Arbeit weitgehend in eigener Verantwortung. Damit sind sie in der Lage, flexibel auf Anforderungen zu reagieren. Eine interessante, eigenständige Arbeit kann dabei helfen, die Arbeitszufriedenheit der Offshore-Entwickler zu erhöhen. Die Kündigung guter Entwickler und der damit einhergehende Wissensverlust können Projekte bedeutend schwächen, wie schon in Abschnitt 2.3.1 diskutiert wurde (vgl. auch [Kalakota u. Robinson 2005]). Wenn die Offshore-Entwickler interessante Aufgaben übernehmen, kann die Fluktuation verringert werden [Olsson 2004].

Da die Beteiligten sich auch bei unvorhergesehenen Problemen direkt miteinander abstimmen können, ohne dass vorbestimmte Kommunikationswege eingehalten werden müssen, werden die indirekte Weitergabe von Informationen und damit einhergehende Bürokratie vermieden. Fragen können direkt an den technischen oder fachlichen Experten gerichtet und so schneller und leichter geklärt werden. Da auch Raum für informelle Kommunikation gegeben ist, werden die Beteiligten dabei unterstützt, Vertrauen und Teamgeist aufzubauen. Es fällt leichter, eine gemeinsame Verantwortlichkeit zu realisieren und zusammen an Aufgaben zu arbeiten. Das entlastet auch das Management.

Zu einem agilen Vorgehen passt sicherlich eine flexible Steuerung deutlich besser. Die untersuchten Erfahrungen legen jedoch nahe, dass auch bei agiler verteilter Entwicklung Vorteile einer Formalisierung genutzt werden sollten. Es stellt sich die Frage, wie dies abgestimmt werden kann. Welche Aufgaben im Entwicklungsprozess müssen formal dokumentiert werden, bei welchen können Offshore-Entwickler freier und eigenständig, ohne permanente Rücksprache agieren? Daran knüpft die Frage an, welche spezifizierenden Dokumente nötig sind, um den agilen Prozess auszurichten. Mit welchen Dokumentarten können z. B. die Anforderungsspezifikation und das Projektmanagement unterstützt werden?

In Kapitel 5 werden diese Fragen wieder aufgegriffen. Dort wird untersucht, wie mit architekturzentrierter Entwicklung ein guter Kompromiss zwischen einer engen Steuerung und einer flexiblen, eigenständigen Arbeit gefunden werden kann und wie die Architekturbeschreibung als zentrales Dokument die Abstimmung der Teams unterstützt.

3.5 Zusammenfassung

In diesem Kapitel wurden Möglichkeiten und Schwierigkeiten der Anwendung agiler Methoden und Techniken bei der verteilten Softwareentwicklung betrachtet. Ein Vergleich der Annahmen und Eigenschaften von agilen Methoden und verteilten Projekten ergab, dass es durchaus Gegensätze gibt, insbesondere bei der Intensität der Zusammenarbeit zwischen Entwicklern und Kunden sowie bezüglich des Umfangs der Dokumentation und Formalisierung des Entwicklungsprozesses. Es gibt jedoch auch viele positive Eigenschaften agiler Methoden, die bei verteilter Entwicklung genutzt werden können, wie z. B. die höhere Flexibilität, die größere Eigenständigkeit der Entwicklungsteams, die Änderbarkeit von Anforderungen und die Bereitstellung von Artefakten und Prozessen zur Unterstützung der Kommunikation zwischen allen Beteiligten.

In einer Analyse von veröffentlichten Erfahrungsberichten der agilen Methoden Distributed Extreme Programming, Distributed Scrum, DSDM-O u. a. wurde die Eignung dieser Methoden für global verteilte Entwicklung untersucht. Als Ergebnis der Analyse wurde ein Katalog von Themen, Problemen und Lösungsansätzen entwickelt. Es zeigte sich, dass die Flexibilität und lokale Entscheidungsfreiheit durch den Einsatz von agilen Methoden einen Vorteil gegenüber einem plangetriebenen Vorgehen darstellen. Dieser

Vorteil kann gerade in Projekten und Projekten mit innovativem Charakter – und damit zwangsweise unvollständiger und nur vorläufiger Spezifikation – genutzt werden. In der Analyse konnten mehrere Maßnahmen ermittelt werden, die dabei helfen, agile verteilte Entwicklung erfolgreich umzusetzen.

Der Katalog zeigt jedoch auch, dass sich bei der Anwendung von agilen Methoden in verteilten Entwicklungsprojekten zusätzliche Schwierigkeiten ergeben. Besondere Probleme stellen sich in kleinen Projekten, die nicht über die nötige Vorlaufzeit und den Finanzrahmen verfügen, um einige technische und organisatorische Lösungen umzusetzen. Hier besteht weiterer Forschungsbedarf. Genauso wie in zwei weiteren wichtigen Bereichen: dem Themenkomplex Kommunikation, Kooperation und Koordination sowie dem Ausgleich zwischen enger Steuerung und Freiheit der Entwickler. Sich daraus ergebende Fragen werden in den nächsten Kapiteln aufgegriffen und weiteren Analysen unterzogen.

Teil II

Erarbeitung von Lösungskonzepten

4 Kommunikation beim Agilen Offshoring

In diesem Kapitel wird die größte Schwierigkeit bei agiler verteilter Entwicklung aufgegriffen: die Kommunikation zwischen allen Beteiligten. Durch die geografische Verteilung wird vor allem die Koordination und Kooperation zwischen den Teams schwieriger als bei nicht-verteilten Projekten. Beim Offshoring kommt die organisatorische Verteilung hinzu, welche die Zusammenarbeit zusätzlich erschweren kann. Daher steht in diesem Kapitel Agiles Offshoring im Mittelpunkt, das einen häufig anzutreffenden Fall der agilen verteilten Entwicklung darstellt.

Verschiedene Kooperationsmodelle, d. h. Organisationsformen der Verteilung der Beteiligten auf die Projektstandorte und ihre Zusammenarbeit, werden durch die verteilte Arbeit auf unterschiedliche Art und Weise beeinflusst. Wir untersuchen in diesem Kapitel, welche Kooperationsmodelle für Agiles Offshoring besonders geeignet sind und wie sie jeweils beim Umgang mit den im letzten Kapitel identifizierten Problemen helfen.

Zur systematischen Aufarbeitung des Themas werden im ersten Abschnitt zuerst grundlegende Modelle und Begriffe aus dem Bereich Kommunikation vorgestellt. Anschließend werden Technologien zur Unterstützung von Kommunikation, Koordination und Kooperation diskutiert. In Abschnitt 4.2 werden wichtige Auswirkungen kultureller Unterschiede auf die Kommunikation betrachtet. Diese Ausführungen dienen als Analyserahmen für die Untersuchung der bedeutendsten Kooperationsmodelle für Agiles Offshoring in Abschnitt 4.3 und 4.4.

4.1 Kommunikation, Koordination und Kooperation

4.1.1 Grundlegende Begriffe: das 3K-Modell

In der bisherigen Diskussion wurden die Begriffe Kommunikation, Koordination und Kooperation ohne exakte Abgrenzung verwendet. In diesem Abschnitt wird die Einführung von Begriffsdefinitionen, Modellen und Abgrenzungen im Hinblick auf die Konsequenzen für die Zusammenarbeit beim Agilen Offshoring vorgenommen.

Die nachfolgenden drei Definitionen sind wörtlich aus [Teufel et al. 1995, S. 12] übernommen.

Definition Kommunikation:

Kommunikation ist die Verständigung mehrerer Personen untereinander.

Kommunikation kann somit als umfassendster Begriff gebraucht werden. In diesem Sinne wurde der Begriff auch schon beim Thema „Kommunikation zwischen allen Beteiligten“ verwendet, vgl. Abschnitt 3.3. Nach [Kupka et al. 1982] beruht Kommunikation auf Vorstellungen über den Partner und über sich selbst. Sie wird beeinflusst vom Kommunikationsgegenstand, von der Situation und vom Medium, über das die Kommunikation stattfindet. Kommunikation profitiert von vergleichbaren Verstehensgrundlagen, insbesondere Konventionen und Wissen. Eine gemeinsame Sprache ist eine der wichtigsten Grundlagen und erleichtert die Kommunikation erheblich. Wenn mehrere alternative Kommunikationsprozesse zur Verfügung stehen, wird in der Regel der mit dem geringsten Aufwand und dem kleinsten Risiko des Missverstehens bevorzugt (Ökonomieaxiom).

Definition Koordination:

Koordination bezeichnet jene Kommunikation, welche zur Abstimmung aufgabenbezogener Tätigkeiten, die im Rahmen von Gruppenarbeit ausgeführt werden, notwendig ist.

Die Koordination kann entweder auf wechselseitigen Konventionen oder auf ausdrücklichen Regeln beruhen [Züllighoven 1998, S. 428]. Bei Koordinationsprozessen wird nicht vorausgesetzt, dass die Beteiligten auf ein gemeinsames Arbeitsziel hinarbeiten.

Definition Kooperation:

Kooperation bezeichnet jene Kommunikation, die zur Koordination und zur Vereinbarung gemeinsamer Ziele notwendig ist.

Oft wird der Kooperationsbegriff ausgeweitet und umfasst auch **kooperative Arbeit**. Bei dieser „arbeiten verschiedene Personen geplant und koordiniert zusammen, um ein gemeinsames Ergebnis zu erreichen.“ [Züllighoven 1998, S. 428] Züllighoven beschreibt vier wichtige Eigenschaften von kooperativer Arbeit. Wir beziehen hier gleich die Ergebnisse zum Agilen Offshoring der Anwendungsentwicklung aus Kapitel 3 mit ein:

- „Die Beteiligten wollen ein *gemeinsames Produkt* herstellen oder eine *gemeinsame Dienstleistung* erbringen.“

  Im diskutierten Kontext ist das Produkt die gemeinsam zu entwickelnde Anwendung.

- „Sie [die Beteiligten] müssen sich *begrenzte Ressourcen* teilen. Im einfachsten Fall müssen sie den Arbeitsgegenstand austauschen."

  Moderne Versionsverwaltungssysteme und Dokumentenarchive können den Umgang mit gemeinsamen elektronischen Ressourcen sehr erleichtern. Nützlich dabei ist, dass Software im Gegensatz zu vielen anderen Produkten ohne Qualitätsverlust duplizierbar und – bei ausreichender Infrastruktur – leicht übertragbar ist. Dies gilt insbesondere für die Quelltexte als wesentliche Arbeitsgegenstände.

- „Sie [die Beteiligten] müssen ihre *Arbeitshandlungen* so weit notwendig *aufeinander abstimmen*, damit zeitliche oder sachlogische Reihenfolgen eingehalten werden."

  Die Abstimmung zwischen den Teams ist ein großes Problem beim Agilen Offshoring. Die Reihenfolge von Arbeitsschritten wird bei agilen Methoden meist auf Basis kleiner Einheiten – Storys, Tasks oder Features – abgestimmt. Dadurch können sich leicht unvorhergesehene Abhängigkeiten ergeben. Wenn eine gemeinsame Verantwortlichkeit etabliert werden kann, kann die Reihenfolge flexibler gestaltet werden, da auch von anderen Entwicklern erstellte Artefakte geändert werden dürfen.

- „Sie [die Beteiligten] müssen sich in irgendeiner Weise darüber *verständigen*, was von wem wie getan wird."

  Für Agiles Offshoring geeignete Strategien zur Arbeitsteilung und -abstimmung werden in diesem Kapitel in Abschnitt 4.3 behandelt.

Die Kooperation kann implizit oder explizit erfolgen. Bei einer impliziten Kooperation können mehrere Benutzer auf gemeinsame Ressourcen zugreifen, ohne dass die Kooperation oder Koordination im Benutzungsmodell sichtbar wird. Ein Benutzer bemerkt z. B. einen konkurrierenden Zugriff eines anderen Benutzers auf das von ihm benötigte Arbeitsmaterial nur dadurch, dass dieses für ihn gesperrt ist. Eine Kooperation wird im Benutzungsmodell explizit, wenn sie durch geeignete Kooperationsmittel und -medien unterstützt wird. Beispiele für Kooperationsmittel sind Laufzettel und Vorgangsmappen. Beispiele für Kooperationsmedien sind elektronische Gruppenpostfächer und Notizbretter [Züllighoven 1998, S. 429 ff.]. Bei einer disjunktiven Kooperation reicht es, wenn ein Beteiligter das Ziel erreicht (Beispiel: Lösung eines algorithmischen Problems). Bei einer konjunktiven Kooperation müssen alle Beteiligten ihre jeweiligen Teilziele erreichen (Beispiel: Komponentenorientierte Entwicklung einer Anwendung) [Oberquelle 1999]. Bei der Mehrzahl der Aufgaben im Rahmen der Softwareentwicklung ist eine konjunktive Kooperation anzutreffen.

Kalakota und Robinson unterscheiden im Zusammenhang mit Offshoring zwischen strategischer und operativer Kommunikation [Robinson u. Kalakota 2004, S. 226 ff.]. Die strategische Kommunikation betrifft die formalen Rahmenbedingungen, die grundsätzlichen Absprachen und die Strategien, die beim Offshoring verfolgt werden. Im Rahmen einer strategischen Kommunikation sollten Mitarbeiter im Unternehmen des

Auftraggebers über die Gründe und Ziele des Offshoring aufgeklärt werden. Die operative Kommunikation betrifft hingegen die tagtägliche Verständigung zwischen Onshore- und Offshore-Teams sowie Auftragnehmer und Auftraggeber. Beide Arten der Kommunikation sind für erfolgreiche Offshoring-Projekte wichtig und sollten gut geplant werden.

Die Unterscheidung von Kommunikation, Koordination und Kooperation wird auch als 3K-Modell bezeichnet. Abbildung 4.1 aus [Teufel et al. 1995, S. 11] zeigt den Zusammenhang der definierten Begriffe. Die Hierarchie ergibt sich dadurch, dass Kooperationsprozesse für ihre Durchführung Koordinationsprozesse erfordern. Koordinationsprozesse wiederum erfordern Kommunikationsprozesse.

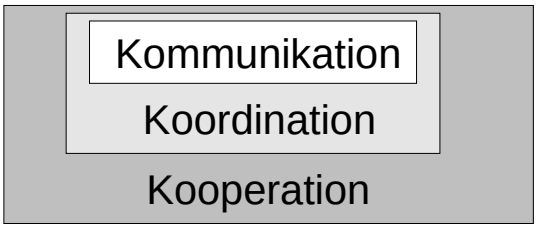


Abbildung 4.1: *Hierarchische Darstellung des 3K-Modells, nach [Teufel et al. 1995, S. 11]*

Boden et al. haben den Begriff **Artikulationsarbeit** des US-amerikanischen Soziologen Anselm Strauss [Strauss 1993] wieder aufgegriffen und auf verteilte Arbeit bei der Softwareentwicklung bezogen. Artikulationsarbeit in diesem Sinne umfasst alle Aktivitäten, die nötig sind, um kooperativ zusammenarbeiten zu können. Insbesondere beschäftigt sie sich mit der Verteilung der Arbeit und allen Handlungen, individuell und gemeinschaftlich, die für die Zusammenarbeit nötig sind [Boden et al. 2009]. Wenn man dieser Begriffsverwendung folgt, können die folgenden Ausführungen zur Zusammenarbeit beim Agilen Offshoring auch als ein spezieller Fall von Artikulationsarbeit gesehen werden.

4.1.2 Technologische Unterstützung der Zusammenarbeit

Eine Verbindung zwischen dem 3K-Modell und den im letzten Kapitel identifizierten Problemlösungsansätzen für Agiles Offshoring lässt sich über das „Klassifikationsschema nach Unterstützungsfunktionen“ [Teufel et al. 1995] herstellen (vgl. Abbildung 4.2). Dadurch können für spezifische Problemfälle geeignete technologische Mittel zur Unterstützung der Zusammenarbeit ausgewählt werden.

Im Klassifikationsschema finden sich verschiedene Applikationstypen zur Unterstützung verteilter Arbeit in ein Dreieck mit den 3Ks angeordnet. Benachbarte Applikationstypen wurden zu vier, nicht überschneidungsfreien Systemklassen zusammengefasst. In der nachfolgenden Aufzählung dieser Klassen wird zuerst die jeweilige Charakterisierung

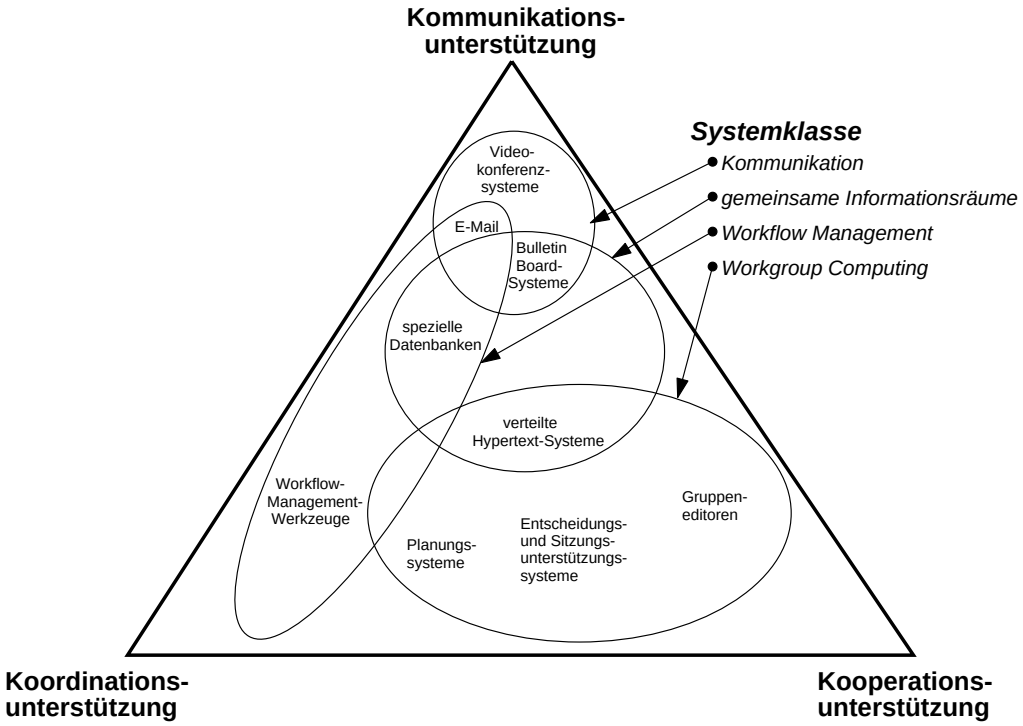


Abbildung 4.2: *Klassifikationsschema nach Unterstützungsfunktionen, nach [Teufel et al. 1995, S. 27]*

aus [Teufel et al. 1995] kurz dargestellt. Anschließend werden die Untersuchungsergebnisse zum Agilen Offshoring mit einbezogen. Dabei wird der Nutzen der gefundenen Lösungsansätze erörtert.

Systemklasse Kommunikation: Diese Klasse beinhaltet Systeme, mit denen räumliche und zeitliche Distanzen überbrückt und Informationen ausgetauscht werden können. Beispiele sind Videokonferenzsysteme, E-Mail und Bulletin Board-Systeme.

Beim Agilen Offshoring werden – wie bei agiler verteilter Entwicklung allgemein – erfolgreich weitere Systeme wie Instant Messaging und Wikis eingesetzt. Spezifische Probleme gibt es bei mangelhafter Infrastruktur und Schwierigkeiten mit Fremdsprachen. In Abschnitt 3.4.3 haben wir gesehen, dass der Aufbau einer geeigneten Kommunikationsinfrastruktur für kleine Projekte problematisch sein kann.

Systemklasse gemeinsame Informationsräume: In gemeinsamen Informationsräumen können Informationen für alle Beteiligte über längere Zeit zugänglich gemacht werden.

Technisch werden gemeinsame Informationsräume beim Agilen Offshoring meist als zentrale Dokumentenarchive mit Versionsverwaltung und Wikis realisiert. Zur Erhöhung der Flexibilität dienen die Prinzipien Selbstorganisation und gemeinsame Verantwortlichkeit.

Systemklasse Workflow-Management: Mit Workflow-Management-Werkzeugen können Workflows – arbeitsteilige Prozesse mit einer endlichen Folge von Aktivitäten – modelliert, ausgeführt und gesteuert werden.

Die Abarbeitung formaler Workflows ist mit dem Grundgedanken agiler Methoden schlecht vereinbar. Workflow-Prozesse sind gerade nicht leicht anpassbar, erlauben den Beteiligten keine schnelle und flexible Reaktion auf Änderungen und werden daher als zu einschränkend gesehen (vgl. die Merkmale agiler Methoden in Abschnitt 3.2.3). Die Diskussion zu enger oder flexibler Steuerung der Zusammenarbeit in Abschnitt 3.4.5 zeigt jedoch, dass eine stärker reglementierte Arbeit von einigen Autoren als vorteilhaft für Agiles Offshoring betrachtet wird. In den in Kapitel 3 untersuchten Erfahrungsberichten wurden dafür jedoch Konventionen und keine formalen Workflows eingesetzt. Prozesse, die nicht direkt mit der Softwareentwicklung zu tun haben, z. B. im Management und im Berichtswesen, bieten sich noch am ehesten für eine Unterstützung durch Workflow-Management-Werkzeuge an.

Systemklasse Workgroup-Computing: Zu dieser Klasse gehören Systeme, die kooperative Aufgaben einer Gruppe oder eines Teams unterstützen. Allgemeine Beispiele dafür sind Terminverwaltungssysteme, Gruppeneditoren und Entscheidungs- und Sitzungsunterstützungssysteme.

> Diese Werkzeuge scheinen auch für Agiles Offshoring nützlich zu sein, so sie denn flexibel gehandhabt werden können. Darüber hinaus wurde eine Reihe spezieller Werkzeuge entwickelt. Dazu gehören Editoren für verteiltes Programmieren in Paaren und auf agile Methoden zugeschnittene Projektplanungs- und Trackingwerkzeuge.[1]

Es lässt sich festhalten, dass die 3Ks auch bei agiler verteilter Entwicklung mit geeigneten Mitteln technisch unterstützt werden können. Neben allgemeinen Werkzeugen zur Zusammenarbeit wurden auch Lösungen entwickelt, die Besonderheiten der agilen Entwicklung berücksichtigen und speziell unterstützen.

Geeignete Werkzeuge sind jedoch nur *eine* Voraussetzung für eine gute Kommunikation. Mindestens genauso wichtig sind organisatorische Regelungen und Konventionen. Dies ist vor allem deshalb der Fall, weil sich mehrere Aspekte verteilter Arbeit auch mit immer ausgereifteren technischen Lösungen auf absehbare Zeit nicht zufriedenstellend unterstützen lassen. Nach Olson und Olson betrifft dies insbesondere diese vier Bereiche [Olson u. Olson 2000]:

1. **Gemeinsamkeiten, Kontext und Vertrauen:** Personen, die im selben Land aufwachsen, teilen gewisse Erfahrungen und Einstellungen. Die Zusammenarbeit mit Fremden erfordert erst einen Aufbau dieser gemeinsamen Basis und ein Kennenlernen des fremden Kontextes, um gegenseitiges Vertrauen zu schaffen.

2. **Unterschiedliche Zeitzonen:** Auch mit hoch entwickelten technischen Systemen werden sich Zeitunterschiede und andere Arbeitsrhythmen nicht aufheben lassen. Insbesondere die synchrone Kommunikation leidet.

3. **Kultur:** Es gibt keine überzeugenden Anzeichen, dass die fortschreitende Globalisierung der Arbeitswelt zu einem Abbau von kulturellen Unterschieden führt. Diese werden auch weiterhin zu Überraschungen und Missverständnissen führen können.

4. **Wechselbeziehungen:** Wenn lokale Kontexte, unterschiedliche Zeitzonen, kulturelle Unterschiede und Technik zusammenwirken, können unerwartete Probleme entstehen.

Zur genaueren Untersuchung der kommunikativen Besonderheiten des Agilen Offshoring werden im nächsten Abschnitt zunächst kulturelle Unterschiede und ihre Auswirkungen auf die Kommunikation genauer betrachtet.

[1] Ein weitverbreitetes Werkzeug dieser Art für Extreme Programming ist der XPlanner, siehe `http://xplanner.org/`.

4.2 Beeinflussung der Kommunikation durch kulturelle Unterschiede

Kulturelle Fragen und ihre Auswirkungen auf die Softwareentwicklung sind ein umfassendes, komplexes Thema. In dieser Arbeit wird es nur insoweit behandelt, wie es für die nachfolgende Betrachtung von Kooperationsmodellen wesentlich ist. Für eine weitergehende Darstellung sei auf spezifische Literatur verwiesen, siehe z. B. [Krishna et al. 2004], [Carmel 1999], [Narayanaswamy u. Henry 2005], [MacGregor et al. 2005] und [Nicklisch et al. 2008].

Mit unterschiedlicher Kultur ist mehr gemeint als nur länderspezifische Kultur. Im weiter gefassten Kulturbegriff werden unterschiedliche Ebenen berücksichtigt, die sich auf nationale, berufliche, religiöse und andere Aspekte beziehen. Diese Ebenen beeinflussen sich gegenseitig, sodass sie zusammen betrachtet werden müssen [Gallivan u. Srite 2005]. So sollten z. B. auch unterschiedliche Ausbildungen und Weltbilder, Rollen und Aufgaben berücksichtigt werden.

Kulturelle Unterschiede zeigen sich auch innerhalb eines Landes und an einem Standort zwischen Auftraggeber und Auftragnehmer; Fachabteilung und IT; Kunden, Entwicklern und Managern sowie verschiedenen Entwicklerteams. Bei dieser Sichtweise kommen mit dem Offshoring nur weitere Kulturdimensionen hinzu, die aber große Auswirkungen auf Entwicklungsprojekte haben und klassische Probleme noch verstärken [Kornstädt u. Sauer 2007a].

Bei der Diskussion kultureller Fragen sollten drei grundsätzliche Dinge nicht aus den Augen verloren werden. Zum einen ist es vor allem bei einer wirtschaftlich geprägten Sichtweise wichtig, auch die Würde und die Bedürfnisse des einzelnen Menschen zu berücksichtigen. Toni Steimle merkt dazu an [Steimle 2007, S. 87]: „Offshore-Mitarbeiter sind nicht einfach Ressourcen. Es sind Mitarbeiter, die unter Umständen für mehrere Jahre Teil des Entwicklungsteams sind. Sie identifizieren sich mit dem Projekt und mit dem Auftraggeber. Unter Umständen sind sie nach einer gewissen Zeit sogar enger mit dem Team des Auftraggebers verbunden als mit der Firma, bei der sie angestellt sind.“

Zweitens sollte man sich stets bewusst sein, dass durch die Untersuchung kultureller Fragen nur generelle Aussagen über das voraussichtliche Handeln der Beteiligten getroffen werden können. Zu einem großen Teil wird das Verhalten des Einzelnen durch seine Individualität bestimmt und ist daher nicht genau vorhersagbar. Drittens handelt es sich bei Kultur um etwas gar nicht oder nur schwer und langfristig Änderbares, was entsprechende Aufklärung und Toleranz nötig macht [Hofstede 2006].

In den nächsten Abschnitten werden zunächst die Auswirkungen von unterschiedlicher nationaler Kultur und anschließend die unterschiedlichen Rollen und Wertvorstellungen in agilen Projekten diskutiert, um eine Grundlage für die weiteren Betrachtungen von kulturellen Schwierigkeiten in dieser Arbeit zu bilden.

4.2.1 Nationale Kultur

Die wissenschaftliche Beschäftigung mit unterschiedlichen nationalen Kulturen hat eine große Bandbreite an unterschiedlichen Modellen zur Beschreibung und Analyse von Kulturen hervorgebracht. Viele der Modelle bauen auf das Beschreibungsmittel der Kulturdimension auf, um Verhalten in Gruppen einzuteilen und vergleichen zu können [Lytle et al. 1995]. Weitverbreitet ist das fünfdimensionale Modell von Hofstede [Hofstede 2004].

Die Unterschiede in den Kulturen verschiedener Länder äußern sich auf verschiedene Art und Weise. Um das Handeln unterschiedlicher Menschen besser verstehen zu können, sollte man gewisse Eigenheiten verschiedener Kulturen kennen. Es gibt eine Reihe von Ratgebern und wissenschaftlichen Veröffentlichungen, die Hintergründe, Modelle und praktische Ratschläge zum Umgang mit fremden Kulturen in Gesellschaft und Berufsleben liefern, siehe z. B. [Lewis 2006], [Martin u. Chaney 2008].

Darüber hinaus gibt es Publikationen mit mehr oder weniger konkreten Hinweisen und möglichen Maßnahmen für IT-Projekte im Offshoring. So nennt Böhm vier mögliche negative Folgen kultureller Unterschiede [Böhm 2003, S. 7]:

- Der Wissenstransfer zwischen den Teams wird gehemmt.
- Vorgeschriebene Arbeitsabläufe werden anders ausgelegt. Die Abstimmung kann behindert werden.
- In der individuellen Zusammenarbeit kann es zu Missverständnissen und anderen Problemen kommen.
- Fremdartiges Verhalten wird nicht akzeptiert. Das Vertrauen in das andere Team sinkt.

Kulturelle Unterschiede zwischen Kunden und ihren Dienstleistern können dadurch den Erfolg von Offshoring-Projekten gefährden [Winkler et al. 2008]. Als Strategien zur Risikominimierung schlägt Böhm vor, interkulturelle Trainings durchzuführen, die Leitung von Offshoring-Projekten interkulturell erfahrenen Managern zu übertragen und sicherzustellen, dass die Unternehmensleitung hinter diesen beiden Maßnahmen steht. Des Weiteren sollten die Aufgaben von Offshore- und Onshore-Teams entkoppelt werden und die Entwicklung an einem formalen und rigiden Prozessrahmenwerk (z. B. SEI-CMM) ausgerichtet werden [Böhm 2003]. Vgl. auch die Diskussion in Abschnitt 3.2.4.

Ergänzt werden diese generellen Strategien um konkrete Hinweise zum Verständnis von Mitarbeitern aus anderen Kulturen, die auf Erfahrungen aus durchgeführten Projekten beruhen. Wichtig ist neben einem Verständnis für andere Kulturen auch eine Reflexion des eigenen Verhaltens, auch um Fallstricke zu vermeiden [Böhm 2003]. So wird z. B. das Auftreten der Deutschen im Ausland häufig als geradeheraus und zu direkt angesehen, während Inder um ein gutes Klima bemüht sind und Probleme nicht direkt zugeben [Olsson 2004]. Diese stereotypen Vorstellungen prägen die Erwartungen, spiegeln aber

nicht unbedingt individuelle Erfahrungen wider. Letztendlich hängt die Wahrnehmung einer anderen Kultur vom eigenen kulturellen Hintergrund ab.

Bei der Diskussion von kulturellen Unterschieden sollte man nicht außer Acht lassen, dass kulturelle Gruppen nicht homogen sind und durchaus divergierende Subkulturen aufweisen können und dass Akteure auch mehreren Kulturen zugeordnet werden können [Krick u. Voß 2007]. Im Folgenden werden daher zusätzlich zur nationalen Kultur auch weitergehende kulturelle Rollen und Wertvorstellungen betrachtet.

4.2.2 Unterschiedliche Rollen und Wertvorstellungen

Akteure in agilen Offshoring-Projekten werden durch ihre Rollen im Projekt maßgeblich geprägt. Ein Verständnis dieser Rollen kann das Verhalten von Akteuren in den später in dieser Arbeit beschriebenen Projekten erklären.

Im Folgenden werden typische Rollen in einem XP-Projekt verdeutlicht (vgl. [Wolf et al. 2005]):

Kunde: Der Kunde ist für die fachlichen Anforderungen an die Anwendung zuständig. Er gibt vor, was mit welcher Priorität entwickelt werden soll. In XP ist der Kunde bei der Erstellung der Story-Cards aktiv beteiligt bzw. er schreibt sie sogar selbst. Damit kommt ihm eine wichtige Rolle im Entwicklungsprozess zu. Es ist sinnvoll, die Kundenrolle in zwei Rollen aufzuteilen: die des Auftraggebers und die des Anwenders [Wolf et al. 2005]. Der **Auftraggeber** gibt die (politischen) Projektziele vor und stellt Finanzmittel zur Verfügung. Von ihm hängt es maßgeblich ab, ob ein Projekt als erfolgreich gewertet wird. Der **Anwender** gibt anwendungsfachliches Wissen bezüglich der Anwendungslogik und der konkreten fachlichen Aufgaben am Arbeitsplatz an die Programmierer weiter.

In größeren Firmen kommt meist eine Unterteilung in Fach- und IT-Abteilung hinzu. Die **Fachabteilung** arbeitet an Computerarbeitsplätzen mit mehr oder weniger spezieller Hard- und Software. Die Mitarbeiter haben ein großes Wissen über die fachlichen Prozesse, verstehen aber in der Regel wenig von IT und Anwendungsentwicklung. Die **IT-Abteilung** dient als Dienstleister für die Fachabteilungen. Sie legt fest, welche Soft- und Hardwareausstattung benötigt wird, beschafft Standardsoftware, entwickelt Individualanwendungen bzw. gibt sie bei externen Dienstleistern in Auftrag und administriert die Systeme.

Programmierer: Bei agilen Methoden sind die Programmierer nicht nur für die Implementierung zuständig, sondern auch für Entwurf, Dokumentation und Test. Sie sind somit die technischen Experten. Fachliche Anforderungen und Fragen klären sie häufig direkt mit den Kunden.

Tester: Da Komponententests schon von den Programmierern geschrieben werden, sollen explizite Tester bei agilen Methoden in der Regel nur Akzeptanztests

durchführen. Sie müssen die fachlichen Anforderungen gut verstehen, um mit den Kunden die Tests formulieren und die Ergebnisse interpretieren zu können.

Projektleiter: Diese Rolle war in der ersten Fassung von XP nicht vorgesehen, da das Team eigenverantwortlich handeln sollte. In der zweiten Fassung ist sie hinzugekommen. Die Aufgabe des Projektleiters ist es, jegliche Störung von den Programmierern fernzuhalten, damit sie konzentriert und produktiv arbeiten können. Des Weiteren übernimmt er organisatorische Aufgaben wie die Personal- und Finanzplanung und klärt nicht-fachliche Fragen mit dem Kunden. Oft arbeitet er auch als Tracker und erhebt und kontrolliert Daten zum Projektfortschritt.

Diese Beschreibungen zeigen, dass verschiedene Akteure ein Projekt unterschiedlich wahrnehmen können. Dies macht die Kommunikationsschnittstellen zwischen den Rollen besonders interessant. Die meist vorhandenen unterschiedlichen Orientierungen (fachlich, technisch oder wirtschaftlich) können vor allem in großen Projekten problematisch werden.

4.3 Das Kooperationsmodell als konzeptioneller Rahmen

Es stellt sich die Frage, wie die beschriebenen Probleme im Bereich der Verteilung und der unterschiedlichen Kulturen für die Praxis und für die Forschung verwertbar gemacht werden können. In diesem Abschnitt wird das Konzept des Kooperationsmodells als Rahmen zur Darstellung und Untersuchung von Offshoring-Projekten eingeführt. Zur anschaulichen Darstellung werden grafische Darstellungen verwendet, deren Elemente zunächst vorgestellt werden.

4.3.1 Der Begriff „Kooperationsmodell“

Zu Beginn wird der zentrale Begriff Kooperationsmodell für diese Arbeit definiert und von den benachbarten Begriffen Geschäftsmodell, Vorgehensmodell und Prozessmodell abgegrenzt.

In dieser Arbeit verwenden wir den Begriff Kooperationsmodell in Anlehnung an [Züllighoven 1998, S. 427]:

Definition Kooperationsmodell:

Ein Kooperationsmodell ist ein Modell, das entweder explizit oder implizit die Kooperation beschreibt und regelt.

Amberg und Wiener sprechen in [Amberg u. Wiener 2006] ebenfalls von Kooperationsmodellen als Modellen für die Form der Zusammenarbeit zwischen Offshoring-Partnern. Im Gegensatz zu Geschäftsmodellen beschränken sie sich auf die Verteilung der Projektbeteiligten auf die verschiedenen Projektstandorte und lassen Organisationsformen und Besitzstrukturen außer Acht. Zu einem **Geschäftsmodell** gehört neben der Organisationsform die geografische Verteilung [Robinson et al. 2005]. Damit kombiniert es die Dimensionen „Finanzielle Abhängigkeit“ und „Standort der Leistungserstellung“ der in Abschnitt 2.2 diskutierten IT-Sourcing Übersichtskarte.[2]

Während die Begriffe Vorgehens- und Prozessmodell in der Regel synonym verwendet werden, grenzen Ludewig und Lichter sie klar voneinander ab [Ludewig u. Lichter 2010]. Dem schließen wir uns für diese Arbeit an. Von beiden Modellen lässt sich für konkrete Softwareentwicklungsprojekte nur sprechen, wenn der jeweilige Prozess bewusst gewählt wurde und nicht undefiniert und zufällig ist.

Mit einem **Vorgehensmodell** ist eine Vorstellung von der Vorgehensweise bei der Softwareentwicklung verbunden. Man unterscheidet mittlerweile veraltete, lineare Vorgehensmodelle (z. B. das Wasserfallmodell) von nichtlinearen Vorgehensmodellen (z. B. Prototyping, iterative und inkrementelle Softwareentwicklung, Spiralmodell). Bei letzteren sind Zyklen und bestimme Rückgriffe vorgesehen [Sommerville 2009].

Ein **Prozessmodell** ist weiter gefasst. Im Kern enthält es ein Vorgehensmodell und zusätzlich Organisationsstrukturen, Vorgaben für das Projektmanagement, für Qualitätssicherung, Dokumentation und Konfigurationsverwaltung. Beispiele sind der Unified Process [Scott 2002] und XP. Prozessmodelle sind wesentlich geprägt von Kommunikation, Koordination und Kooperation. Sie spielen eine wichtige Rolle bei der Softwareentwicklung.

Ein Beispiel soll diese Terminologie verdeutlichen. Ein mögliches Prozessmodell für agile Offshoring-Projekte ist Distributed Extreme Programming. Es ist verbunden mit dem Vorgehensmodell iterative und inkrementelle Entwicklung. Als Geschäftsmodell ist die Kombination eines Dual-Shore-Kooperationsmodells (siehe Abschnitt 4.4.1) mit einer Global Sourcing-Strategie (siehe Abschnitt 2.2.1) möglich.

Kooperationsmodelle eignen sich gut als konzeptioneller Rahmen für die systematische Betrachtung der Herausforderungen und Probleme beim Agilen Offshoring. Mit ihnen lassen sich die wichtigsten Akteure, ihre Verteilung auf unterschiedliche Organisationen und Standorte sowie ihre Kommunikationsschnittstellen und -kanäle grafisch klar darstellen. Anhand dieser Grafiken können die aufgeworfenen Fragestellungen systematisch und anschaulich untersucht werden.

[2] Diese Verwendung des Begriffs „Geschäftsmodell“ ist auf Offshoring zugeschnitten. In [Scheer et al. 2003] finden sich eine Bestimmung und eine Abgrenzung des allgemeinen Begriffs.

4.3.2 Eine grafische Notation für Kooperationsmodelle

In der Literatur ist keine einheitliche grafische Darstellung von Kooperationsmodellen zu finden. In dieser Arbeit verwenden wir eine eigens entwickelte Notation, die sich an der Notation für Kooperationsszenarios orientiert. Kooperationsszenarios werden im Rahmen der exemplarischen Geschäftsprozessmodellierung (eGPM) eingesetzt. Mit ihnen lässt sich die Kooperation mehrerer Akteure mit bearbeiteten und über bestimmte Übertragungsmedien verschickten Gegenständen und Informationen darstellen. Im Gegensatz zu den hier verwendeten Kooperationsmodellen sind Kooperationsszenarios in Szenarioform dargestellt und eignen sich mehr für Prozesse als für einen Überblick über alle Kooperationsbeziehungen. Details zu eGPM und Kooperationsszenarios finden sich bei Breitling et al. [Breitling et al. 2006].

Für die hier erstellten Modelle wurden eine Reihe von Annahmen getroffen: In der Organisation des Kunden gibt es eine IT-Abteilung und eine von der Softwareentwicklung profitierende Fachabteilung, wie in der Diskussion agiler Rollen in Abschnitt 4.2.2 beschrieben. Der formale Auftraggeber ist in der IT-Abteilung angesiedelt. Sowohl der Kunde als auch der Offshore-Dienstleister haben Projektleiter bzw. Manager zur Steuerung und Kontrolle des Projekts eingesetzt. Beim Kunden gibt es darüber hinaus die Rolle des Testers, der die Abnahme neuer Releases mit Akzeptanztests koordiniert. Diese Rollen können von mehreren Personen wahrgenommen werden. Eine Person kann auch mehrere Rollen innehaben. Alle Akteure sind in Abbildung 4.3 dargestellt.[3]

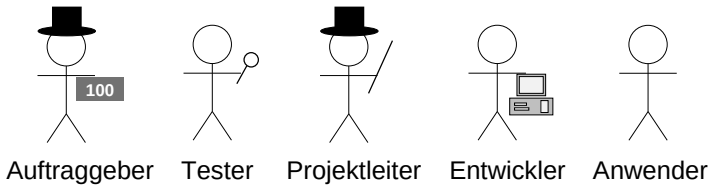


Abbildung 4.3: *Darstellung von Akteuren in Kooperationsmodellen*

Die Akteure sind auf zwei Teams, onshore und offshore, verteilt. Beim Onshore-Team wird nicht zwischen onshore und on-site unterschieden, da dies projektabhängig unterschiedlich gelöst sein kann und für die Diskussion nicht wesentlich ist. Die Zuordnung der Akteure zu den Teams und zu den beteiligten Organisationen wird durch verschiedene, teilweise ineinander geschachtelte Kästen sowie durch unterschiedliche Grauschattierungen der Köpfe verdeutlicht. Pfeile zwischen den Organisationen bedeuten, dass ein Kanal für eine Kommunikationsbeziehung besteht. Wenn in einem Modell bestimmte Akteure einer Organisation kommunizieren, geht der Pfeil direkt von diesen aus.

[3]Die Symbole wurden wie folgt gewählt: Auftraggeber und Projektleiter haben als Manager einen Hut auf. Der Auftraggeber wird mit einem Geldschein dargestellt, da er das Projekt finanziert. Der Projektleiter hat einen Dirigentenstab. Tester werden mit einer Lupe abgebildet, mit der sie Fehler finden. Entwickler tippen an einem Computer. Anwender werden als einfache Strichmännchen dargestellt.

4.3.3 Organisationsübergreifende Kommunikation beim Offshoring

Die generellen Kommunikationsbedarfe zwischen den am Offshoring-Projekt beteiligten Organisationen sind in Abbildung 4.4 illustriert. Hier wird beschrieben, welche Informationen und Artefakte ausgetauscht werden und welche grundsätzlichen Schwierigkeiten bei der Kooperation bestehen. Die Reihenfolge der nachfolgenden Erläuterungen entspricht der Nummerierung der Pfeile in der Abbildung.

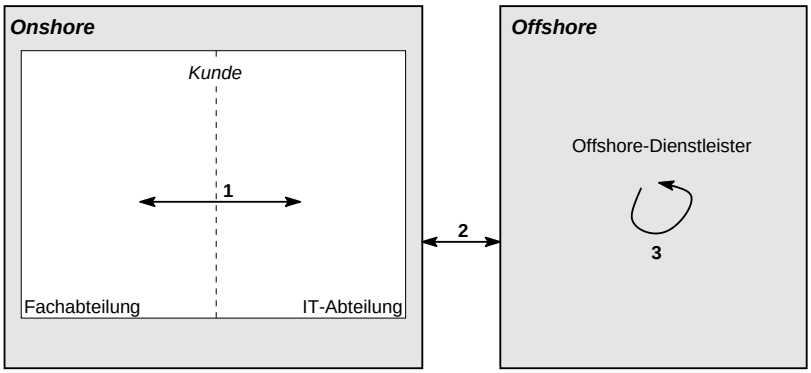


Abbildung 4.4: *Kommunikationsbedarfe bei Offshoring-Projekten*

1. Zwischen IT- und Fachabteilung werden der Bedarf einer Neu- oder Weiterentwicklung und mögliche Umsetzungen geklärt. Gemeinsam werden die Anforderungen dokumentiert. Während der Entwicklungsphase stimmen sich beide Abteilungen ab. Neue Releases werden von der IT-Abteilung für die Fachabteilung installiert und betrieben.

   Die Schwierigkeiten dieses Kommunikationskanals liegen hauptsächlich in der organisatorischen Trennung und in unterschiedlichen Funktionsbereichen. Beide Abteilungen müssen permanent versuchen, fachliche Anforderungen und technische Machbarkeit in Einklang zu bringen. Dies wird erschwert durch unterschiedliche Ausbildungen, Wissen und Herangehensweisen, aufgrund derer manchmal Missverständnisse zwischen Fach- und IT-Abteilung entstehen können.

2. Der Kommunikationskanal zwischen Onshore- und Offshore-Teams ist kritisch für den Erfolg. Zu Projektbeginn geht es um die Aushandlung der Vertragsbedingungen und die Abstimmung der Funktionalität und Qualitätskriterien, z. B. in Form eines Pflichtenhefts. Anschließend folgt die operative Durchführung des Projekts. Je nach gewähltem Prozess gibt es eine oder mehrere Iterationen, in denen das Offshore-Team jeweils eine neue Version erstellt und an den Kunden übergibt. Dieser testet sie und teilt Fehler und Änderungswünsche mit.

   Problematisch an der Kommunikation zwischen den beiden Teams ist, dass bei diesem wichtigsten Kommunikationskanal große Klüfte der Verteilung zu

überbrücken sind. Offensichtlich sind die geografische und zeitliche Distanz sowie die organisatorische Trennung. Je nach Länderkombination kommen meist noch unterschiedliche Sprachen und Kulturen hinzu.

3. Die interne Kommunikation beim Offshoring-Dienstleister während der Softwareentwicklung ist durch die Verteilung vor dem Kunden mehr oder weniger stark verborgen. Der Offshore-Projektleiter gibt seinem Entwicklerteam die Anforderungen und weitere Informationen des Kunden weiter und nimmt umgekehrt Rückfragen und neue Releases des Teams entgegen.

   Bei dieser Kommunikation sollte es keine besonderen Probleme durch Verteilung oder unterschiedliche Kulturen geben.

Im Folgenden werden wir sehen, wie unterschiedliche Kooperationsmodelle mit diesen Kommunikationsbedarfen umgehen.

4.4 Kooperationsmodelle für Agiles Offshoring

In diesem Abschnitt werden ausgewählte Kooperationsmodelle für Agiles Offshoring untersucht. Ziel dabei ist zum einen, ein für Agiles Offshoring geeignetes Modell der Zusammenarbeit zu finden und gesammelte Erfahrungen nutzbar zu machen. Zum anderen sollen grundsätzliche Probleme und Beschränkungen der Kooperationsmodelle und damit weiterer Forschungsbedarf aufgedeckt werden. Dieser Abschnitt erweitert die bereits in [Sauer 2008] publizierten Ergebnisse substanziell.

Um die Darstellung zu vereinfachen, gehen wir davon aus, dass das Projekt von zwei Teams durchgeführt wird: einem Onshore-Team und einem Offshore-Team (vgl. die Begriffsklärung in Abschnitt 2.2.1). Wie die Untersuchung in Abschnitt 3.4.2 gezeigt hat, ergeben sich durch weitere Teams keine anderen Arten von Schwierigkeiten.

Zuerst werden drei Kooperationsmodelle kurz vorgestellt und ihre generellen Kommunikationsschnittstellen und -kanäle bezüglich der Dimensionen der Verteilung (vgl. Abschnitt 3.4.2) und möglicher kultureller Unterschiede untersucht. Anschließend werden die Modelle mit dem im letzten Kapitel entwickelten Katalog von Problemen und Lösungsansätzen beim Agilen Offshoring abgeglichen. Die Analyse wird mit der Beschreibung von Realisierungsalternativen und -empfehlungen abgerundet. Die Untersuchung endet mit einer Gegenüberstellung und Bewertung der drei Kooperationsmodelle.

4.4.1 Kurzbeschreibung der Modelle

In der Literatur und in der Praxis des Offshoring finden sich eine Reihe von Kooperationsmodellen. Die Spanne reicht vom On-Site Delivery, bei dem der Offshoring-Dienstleister dem Kunden Personalressourcen für die Arbeit on-site zur Verfügung stellt [Amberg u. Wiener 2006], bis zu verschiedenen Multisourcing-Modellen (vgl. die

Begriffserläuterungen in Abschnitt 2.2.1). Die in dieser Arbeit untersuchten Modelle sind Direktes Offshoring, Indirektes Offshoring und Dual-Shore-Offshoring mit Onshore- und Offshore-Entwicklung. Sie wurden ausgewählt, da es sich um wesentliche in der Praxis verbreitete Modelle handelt, die sich deutlich voneinander unterscheiden. In der Praxis treten diese Modelle auch in Mischformen auf. Hier werden teilweise idealisierte Ausprägungen beschrieben, die für Agiles Offshoring von KMU typisch sind.

Zur Einführung werden für jedes Modell eine knappe Beschreibung sowie weitere Erläuterungen und häufig anzutreffende Synonyme angegeben.

1. Direktes Offshoring

Kurzbeschreibung: Beim Direkten Offshoring beauftragt der Kunde direkt einen Offshoring-Dienstleister, der die Entwicklung überwiegend an seinem Offshore-Standort durchführt. Typischerweise ist das Modell mit einer klaren Aufgabenteilung verbunden: Der Kunde erhebt on-site die Anforderungen, der Dienstleister setzt sie offshore um.

Begriffsklärung, Synonyme: Die Bezeichnung dieses Kooperationsmodells hebt die unmittelbare, d. h. nicht durch Vermittler unterstützte Beziehung des Kunden zum Offshoring-Dienstleister hervor. Robinson, Kalakota und Sharma bezeichnen dieses Modell auch als „First-Generation Offshore Outsourcing“ und grenzen es damit von später entwickelten, ausgeklügelteren Modellen ab [Robinson et al. 2005]. Da dieses Kooperationsmodell als erstes in größerem Umfang verwendet wurde, wird es vielfach auch als klassisches oder konventionelles Offshoring bezeichnet.

Wenn betont werden soll, dass sämtliche Umsetzungsarbeiten beim Offshoring-Dienstleister durchgeführt werden, bieten sich die Bezeichnungen „Reine Offshore-Softwareentwicklung“ [Gadatsch 2006] oder „Offshore Delivery“ [Amberg u. Wiener 2006] an.

2. Indirektes Offshoring

Kurzbeschreibung: Beim indirekten Offshoring vermittelt ein erfahrener externer Onshore-Dienstleister zwischen dem Kunden und dem Offshoring-Dienstleister.

Begriffsklärung, Synonyme: Neben der Bezeichnung Indirektes Offshoring wird oft auch der Begriff Brücke oder Brückenkopf verwendet. Bei Gadatsch findet sich neben Brückenkopf auch die englische Bezeichnung Facilitator, auf Deutsch übersetzbar mit Vermittler, Unterstützer [Gadatsch 2006]. Ein Spezialfall des Indirekten Offshoring ist das Global Delivery Model. Bei diesem beauftragt der Kunde einen international tätigen Dienstleister, der je nach Bedarf On-site-, Onshore- und Offshore-Ressourcen einsetzt [Robinson et al. 2005].

3. Dual-Shore-Offshoring mit Onshore- und Offshore-Entwicklung

Kurzbeschreibung: Dual-Shore-Offshoring mit Onshore- und Offshore-Entwicklung ist eine Weiterentwicklung des Indirekten Offshoring. Zusätzlich zum Personal des externen Dienstleisters arbeiten auch Entwickler des Offshoring-Dienstleisters für eine gewisse Zeit onshore. Optional arbeitet auch Onshore-Personal temporär offshore.

Begriffsklärung, Synonyme: Viele Autoren sprechen auch von Dual-Shore-Offshoring, wenn der Offshoring-Dienstleister nicht mit Entwicklern, sondern nur mit anderem Personal, z. B. Analysten und Testern, onshore vertreten ist. Bei diesem Modell findet keine verteilte Entwicklung statt. Um diesen wichtigen Unterschied zu betonen, nennen wir das hier diskutierte Modell explizit Dual-Shore-Offshoring mit Onshore- und Offshore-Entwicklung. In der Kurzbeschreibung wurde die Anwesenheit von Onshore-Personal beim Offshoring-Dienstleister als optional beschrieben. Wenn auf den Offshore-Brückenkopf verzichtet wird, ergibt sich eine Mischform aus Indirektem Offshoring und Dual-Shore-Offshoring.

Bei Robinson, Kalakota und Sharma heißt das Modell auch Hybrid Delivery Model (On-site and Offshore) [Robinson et al. 2005]. Gadatsch spricht von Onsite-Offshore Softwareentwicklung [Gadatsch 2006]. Seit einigen Jahren wird das Dual-Shore-Modell auch von mehreren Dienstleistern vermarktet. So findet sich der Begriff Dual-Shore bei Zensar Technologies [Winkler 2005] und NIIT Technologies [Stephan 2005]. Bei Valtech heißt es Duoshore [Schumacher u. Olsson 2005].

4.4.2 Analyse der Kommunikationsschnittstellen und -kanäle

Die Abbildungen 4.5, 4.6 und 4.7 zeigen die drei Kooperationsmodelle in grafischer Darstellung. Anhand der Abbildungen werden im Folgenden die Kommunikationsbeziehungen untersucht.

1. Direktes Offshoring

Beim Direkten Offshoring führt der Kunde das Projekt direkt mit dem Offshore-Dienstleister durch. Er setzt einen Projektleiter ein, der aus der IT-Abteilung stammt, genauso wie der Tester. Mit dieser Konstellation lassen sich die identifizierten Kommunikationsbedarfe der Organisationen unkompliziert decken. Die Nummerierung entspricht der in Abbildung 4.5:

1. Der Projektleiter klärt mit der Fachabteilung fachliche Themen. Er leitet Fragen des Offshoring-Dienstleisters weiter und übermittelt diesem die Antworten der Fachabteilung. Der Tester koordiniert mit der Fachabteilung die fachlichen Tests

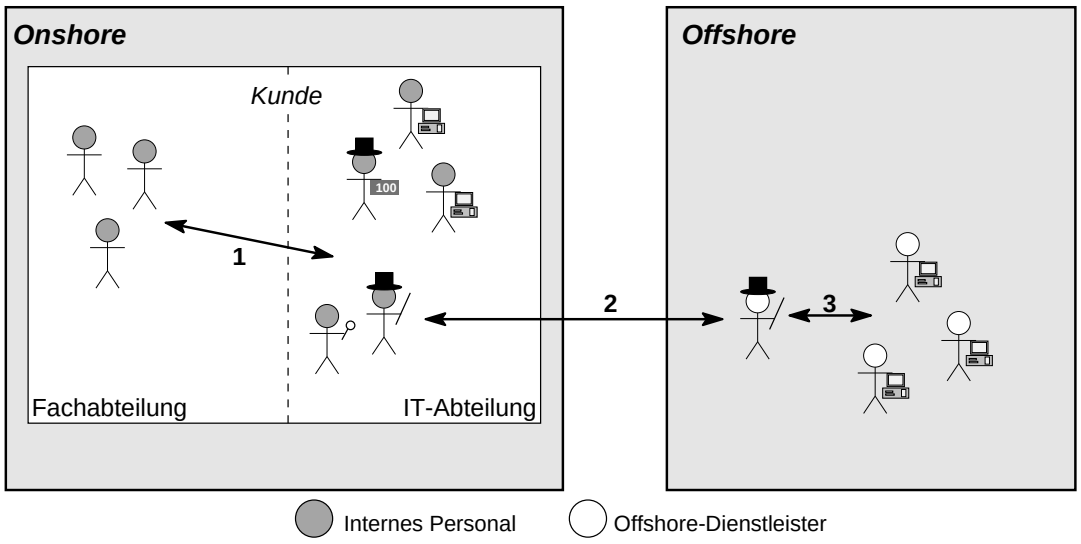


Abbildung 4.5: *Kooperationsmodell Direktes Offshoring*

von Releases. Zwischen dem Projektleiter und der IT-Abteilung werden organisatorische und technische Fragen besprochen. Der Tester bezieht die IT-Abteilung für die technische Abnahme von Releases mit ein.

Fach- und IT-Abteilung müssen die Zusammenarbeit ohne externe Hilfe bewältigen.

2. Der Dialog zwischen den beiden Projektleitern ist beim Direkten Offshoring in der Regel der einzige Kommunikationskanal zwischen Onshore- und Offshore-Teams. Sie sind dafür verantwortlich, die geografische, zeitliche und organisatorische Trennung sowie unterschiedliche Sprachen und Kulturen zu überbrücken.

3. Direktes Offshoring macht keine Annahmen über die Kommunikation innerhalb des Offshore-Dienstleisters.

2. Indirektes Offshoring

Beim Indirekten Offshoring kommen Akteure eines externen Dienstleisters hinzu, die onshore arbeiten. Die Nummerierung entspricht der in Abbildung 4.6:

1. Auch beim Indirekten Offshoring stimmen sich IT- und Fachabteilung des Kunden ab. Insbesondere legen sie die Anforderungen fest. Es kann hilfreich sein, wenn der externe Dienstleister bei der Erstellung des Pflichtenhefts und der Auswahl des Offshoring-Dienstleisters unterstützt und zwischen beiden Abteilungen vermittelt. Zum einen verfügt er über praktische Erfahrung mit Offshoring und kann

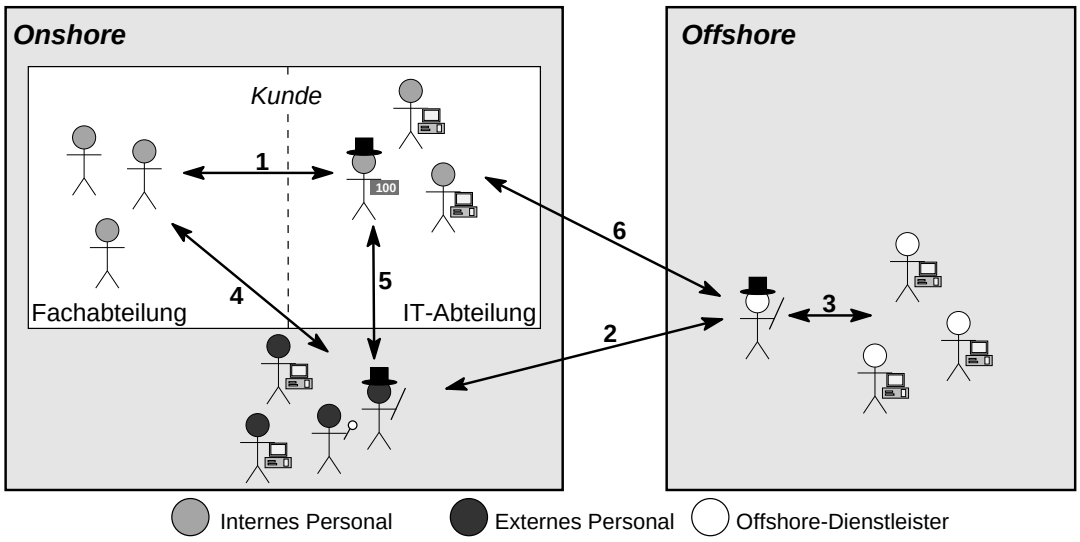


Abbildung 4.6: *Kooperationsmodell Indirektes Offshoring*

daher besser einschätzen, durch welchen Offshore-Dienstleister das Projekt gut abgewickelt werden kann. Zum anderen kann eine Vermittlung durch Externe mit entsprechender Ausbildung die Kommunikation zwischen IT- und Fachabteilung ggf. verbessern.

2. Die wesentliche Kommunikation zur operativen Durchführung des Projekts findet bei diesem Modell zwischen dem externen Projektleiter und dem Projektleiter des Offshoring-Dienstleisters statt.

   Auch hier können die bekannten Erschwernisse der Kommunikation zwischen Onshore- und Offshore-Team auftreten: geografische, zeitliche und organisatorische Trennung sowie unterschiedliche Muttersprachen und Kulturen. Ein Vermittler ist im Umgang mit diesen Schwierigkeiten in der Regel erfahrener.

3. Indirektes Offshoring macht keine Annahmen über die Kommunikation innerhalb des Offshoring-Dienstleisters.

4. Zwischen Fachabteilung und externem Dienstleister werden fachliche Fragen besprochen.

   Bei diesem Kanal ist zu beachten, dass eine organisatorische Trennung besteht und sich der externe Dienstleister erst in die Fachlichkeit einarbeiten muss.

5. Zwischen IT-Abteilung und externem Dienstleister werden organisatorische und technische Rahmenbedingungen besprochen: Wer ist wofür zuständig? Wer übernimmt wofür Verantwortung?

Besonders anfangs wird sich auch hier die organisatorische Verteilung bemerkbar machen, wenn sich der externe Dienstleister erst in die bestehende technische Infrastruktur und in die fachlichen Anforderungen einarbeiten muss.

6. Obwohl der Großteil der Kommunikation mit dem Offshoring-Dienstleister über den externen Dienstleister erfolgt, sind je nach rechtlicher und organisatorischer Situation einige Punkte direkt zwischen dem Kunden und dem Offshoring-Dienstleister zu regeln, hauptsächlich vertragliche und organisatorische Absprachen zu Beginn und am Ende des Projekts.

   Die bei 2. beschriebenen Erschwernisse können auch hier auftreten.

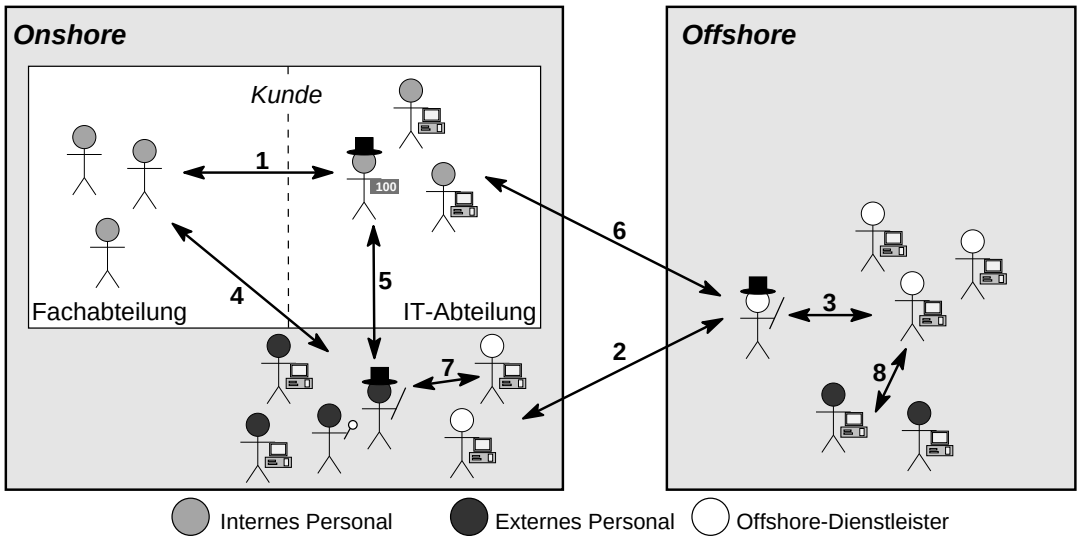


Abbildung 4.7: *Kooperationsmodell Dual-Shore-Offshoring*

3. Dual-Shore-Offshoring

Beim Dual-Shore-Offshoring kommen keine neuen Akteure hinzu. Da Mitarbeiter des externen Dienstleisters auch offshore und Mitarbeiter des Offshore-Dienstleisters auch onshore arbeiten können, ändert sich ihre Verteilung. Die Nummerierung entspricht der in Abbildung 4.7:

1. Wie beim Indirekten Offshoring kann der externe Dienstleister bei der Kommunikation zwischen IT- und Fachabteilung vermitteln.

2. Für die Kommunikation zwischen Onshore- und Offshore-Teams gibt es beim Dual-Shore-Modell viele Möglichkeiten. Kommunizieren nur die Teams der jeweiligen Dienstleister direkt miteinander oder gibt es auch teamübergreifende Kommunikation? Welche Rolle spielen die Projektleiter? Wie wird die gemeinsame

Entwicklung geregelt? Hier ist eine Klärung im Detail nötig. Eine entsprechende Diskussion von Realisierungsalternativen folgt in Abschnitt 4.4.7.

Generell lässt sich festhalten, dass durch die beiden Brückenköpfe viele Probleme der Verteilung und kultureller Unterschiede abgemildert werden können. Die Verteilung nach Funktionsbereichen entfällt weitgehend: Bei den anderen Kooperationsmodellen befinden sich auf der einen Seite Analysten und auf der anderen Seite Entwickler. In die eine Richtung werden hauptsächlich fachliche Anforderungen und in die andere Richtung Implementierungen in neuen Anwendungsversionen übermittelt. Beim Dual-Shore-Offshoring kommunizieren dagegen Entwickler mit Entwicklern. Auch wenn diese verschiedene Entwicklerkulturen pflegen können (vgl. z. B. die Diskussion von agilen gegenüber plangetriebenen Ansätzen in Kapitel 3), bietet das eine geeignetere Grundlage für die Kommunikation. Bestehen bleiben bei diesem Kommunikationskanal jedoch die geografische und die zeitliche Verteilung.

3. Wenn die Teams direkt miteinander kommunizieren können, hat der Offshore-Projektleiter beim Dual-Shore-Offshoring eine weniger exponierte Stellung. Während er bei den anderen Modellen für die gesamte Entwicklung verantwortlich war, liegt jetzt ein Teil der Verantwortung auch onshore.

4. Je nach Bedarf kann onshore arbeitendes Personal des Offshoring-Dienstleisters zur Abstimmung mit der Fachabteilung hinzugezogen werden. Durch den persönlichen Kontakt und den Einblick in die Arbeit vor Ort können fachliche Anforderungen leichter vermittelt werden. Die Offshore-Mitarbeiter können die Anforderungen dann in der Regel für ihre Offshore-Kollegen vollständiger und verständlicher aufbereiten.

   Auch wenn durch die Vor-Ort-Präsenz von Offshore-Personal die Dimensionen geografische und zeitliche Verteilung wegfallen, bleiben die organisatorische Verteilung sowie kulturelle Unterschiede und Sprachprobleme bestehen. Durch den persönlichen Umgang können jedoch das gegenseitige Verständnis erleichtert und Missverständnisse vermieden werden.

   Eine zusätzliche Schwierigkeit liegt in der Verteilung nach Funktionsbereichen. Dadurch muss erst eine gemeinsame Fachsprache gefunden werden. Je nach Ausbildungsniveau und Tätigkeitsbereich der Mitarbeiter der Fachabteilung kann es schwierig für sie sein, sich direkt mit Offshore-Mitarbeitern in einer Fremdsprache auseinanderzusetzen.

5. Onshore arbeitendes Personal des Offshoring-Dienstleisters kann auch direkt mit der IT-Abteilung kommunizieren. Offshore-Entwickler können eventuell die technische Umsetzbarkeit von Anforderungen besser als der externe Dienstleister beurteilen. Sie können auch die Genauigkeit der Aufwandsschätzung von offshore zu entwickelnden Teilen verbessern. Durch die direkte Beteiligung von Offshore-Entwicklern kann technisches und organisatorisches Wissen auch schneller an das Offshore-Team übermittelt werden.

Bis auf die Verteilung nach Funktionsbereichen können alle bei 4. beschriebenen Probleme auftreten.

6. Durch die Brückenköpfe besteht weniger Notwendigkeit, zwischen Kunden und dem ausländischen Standort des Offshoring-Dienstleisters direkt zu kommunizieren. Die meisten Abstimmungen können direkt vor Ort erfolgen.

7. Für die Zusammenarbeit von externen Onshore- und Offshore-Entwicklern bieten sich verschiedene Modelle an. Im Regelfall arbeitet man in gemischten Teams.

   Die Onshore-Entwickler können direkt fachliches Wissen der Anwendungsdomäne vermitteln, sodass Rückfragen beim Onshore-Team vermieden werden können. Im direkten Kontakt lassen sich auch Sprachprobleme mindern.

8. Offshore bieten sich ähnliche Möglichkeiten wie onshore. Ein Ziel sollte sein, die Offshore-Entwickler gut in die verwendete Technik einzuarbeiten und nebenbei ihr fachliches Wissen zu erweitern.

4.4.3 Bewertung der Kooperationsmodelle

Auf Grundlage der Analyse der Kommunikationsbeziehungen lassen sich jetzt die entscheidenden Faktoren sowie Vor- und Nachteile der Kooperationsmodelle diskutieren. Dabei fließen weitere Forschungsergebnisse und praktische Erfahrungen des Autors dieser Arbeit ein. Die Diskussion berücksichtigt noch nicht explizit die Besonderheiten, welche durch die Anwendung von agilen Methoden entstehen. Dies erfolgt erst im nächsten Abschnitt.

1. Direktes Offshoring

Entscheidende Faktoren: Beim Direkten Offshoring ist es vor allem schwierig, den Offshore-Entwicklern fachliches Wissen zu vermitteln und ihnen die Anforderungen verständlich zu machen. Das kann so aufwendig sein, dass sich das Offshoring nicht mehr lohnt. Cockburn berichtet von einem Extremfall [Cockburn 2002, S. 181]: “One designer told me that his team had to specify the program to the level of writing the code itself and then had to write tests to make sure that the programmers had correctly implemented every line they had written. [...] In the time they spent specifying and testing, they could have written the code themselves, and they would have been able to discover their design mistakes much faster.”

Beim Direkten Offshoring sind die Anforderungen an die Mitarbeiter des Kunden hoch. Insbesondere für KMU gilt, dass die leitenden Mitarbeiter zwar in der Regel über einige Erfahrung in Softwareentwicklungsprojekten, aber häufig über sehr wenige oder überhaupt keine im Management von Offshoring-Projekten besitzen. Im Kundenunternehmen muss eine Steuerorganisation eingerichtet werden, welche die

Offshoring-Aktivitäten leitet und koordiniert. Im einfachsten Fall besteht diese Organisation nur aus dem Projektleiter. Dies reicht in der Regel aber nicht aus, da die Aufgaben vielfältig und nicht einfach sind, insbesondere bei der Entwicklung von Anwendungssoftware. Albayrak et al. beschreiben eine Steuerorganisation mit drei Teams: Das Service-Management-Team kommuniziert mit den betroffenen Fachabteilungen. Das Service-Delivery-Team prüft die Leistungen, die der Dienstleister erbringt. Das Service-Financial-Team übernimmt das Controlling und die Budgetierung des Projekts. Mitunter muss die Steuerorganisation über kaufmännische, technische und fachliche Kenntnisse verfügen [Albayrak et al. 2007]. Sparrow betont in [Sparrow 2003], dass die geänderten Anforderungen durch geeignete Weiterbildungsmaßnahmen zu unterstützen sind. Vor allem sind eine erhöhte Sozialkompetenz und mehr Kenntnisse in Mitarbeiterführung vonnöten.

Vorteile: Projekte mit Direktem Offshoring sind kostengünstig realisierbar, da die Arbeit hauptsächlich von Offshore-Personal mit zumeist niedrigeren Lohnkosten erledigt wird.

Onshore- und Offshore-Teams können weitgehend entkoppelt voneinander arbeiten. Die Verteilung der Aufgaben zwischen dem On-site-Team des Kunden und dem Offshore-Team des Dienstleisters folgt in der Regel der Unterteilungsvariante der phasenweisen Aufteilung (vgl. Abschnitt 2.2.3). Die Anforderungen und technischen Vorgaben werden on-site aufgestellt. Die Entwicklung erfolgt durch das Offshore-Team. Integration und Akzeptanztests werden on-site vom Kunden durchgeführt. Bei einem agilen Vorgehen mit Iterationen werden mehrere Versionen entwickelt, integriert und getestet. Nach der Übernahme der entwickelten Anwendung in den produktiven Betrieb schließt die Wartungs- und Gewährleistungsphase an, die der Dienstleister offshore erbringt. Persönliche Treffen finden typischerweise – wenn überhaupt – nur am Anfang und am Ende des Projekts statt, ansonsten arbeiten Kunden- und Dienstleister-Teams getrennt. Dies führt dazu, dass die Organisation des Kunden intern agil und der Offshoring-Dienstleister intern plangetrieben arbeiten können, solange die Abhängigkeiten durch abgestimmte Anforderungsdefinitionen und Releasezeitpunkte berücksichtigt werden.

Nachteile: Beim Direkten Offshoring ist der Kunde allen Schwierigkeiten des Offshoring direkt ausgesetzt. Bei Problemen sind nur Ansprechpartner des Offshoring-Dienstleisters verfügbar. Die Betrachtung der Kommunikationsschnittstellen macht deutlich, dass die direkte Verbindung zwischen Onshore- und Offshore-Team (siehe 3. Kanal), die in der Regel nur aus den beiden Projektleitern besteht, stark belastet ist. Zum einen werden über sie viele wichtige Informationen ausgetauscht. Zum anderen können die in Abschnitt 4.2 diskutierten kulturellen Probleme auftreten. Dies kann die Effektivität der Kommunikation zwischen Onshore- und Offshore-Teams empfindlich stören.

2. Indirektes Offshoring

Entscheidende Faktoren: Beim Indirekten Offshoring hängt viel von der Erfahrung und vom Können des externen Dienstleisters ab. Er sollte schon mehrere Offshoring-Projekte durchgeführt haben und über Erfahrung im Umgang mit gemischten Teams und kulturellen Unterschieden verfügen. Im Regelfall kommt er aus dem Land des Kunden oder spricht zumindest gut dessen Sprache. Somit kann er zwischen dem Kunden und dem Offshoring-Dienstleister vermitteln und viele Probleme eigenständig, ohne Einbeziehung des Kunden lösen.

Durch geeignete Auswahl des Dienstleisters, z. B. nach Branchen- und technischem Wissen, kann der Kunde weiteres Know-how ins Projekt einbringen. Der externe Dienstleister ist in der Regel ein lokaler oder überregionaler IT-Dienstleister. Diese Rolle bietet auch kleineren deutschen Unternehmen gute Aussichten: „Die Chance der lokalen IT-Dienstleister besteht in der Mittler-Rolle zwischen Unternehmen und Offshoring-Anbietern, die es zurzeit noch schwer haben Fuß zu fassen." [Buchta et al. 2004] Nach einer Umfrage punkten lokale Anbieter bei deutschen Unternehmen mit Wettbewerbsvorteilen durch Sprache, Vor-Ort-Präsenz und Kenntnisse von Gesetzen und anderen lokalen Gegebenheiten [Prieß 2006].

Vorteile: Der Hauptvorteil des Indirekten Offshoring besteht darin, dass der Kunde nicht direkt mit einem entfernten Offshoring-Dienstleister kommunizieren muss, sondern einen einheimischen Partner vor Ort hat, der on-site beim Kunden oder zumindest onshore arbeitet. Die Kommunikation zwischen Onshore- und Offshore-Teams wird erleichtert, da der externe Dienstleister mehr Erfahrung mit Offshoring mitbringt und im Idealfall deutlich besser mit der Sprache und Kultur des Offshoring-Dienstleisters vertraut ist als der Kunde.

Auch die Arbeit onshore kann verbessert werden: Carmel und Nicholson bezeichnen die Rolle des externen Dienstleisters als Expertise Intermediaries, auf Deutsch übersetzbar als sachkundige Mittelsleute [Carmel u. Nicholson 2005]. Die Hinzunahme eines kompetenten externen Dienstleisters ist gerade für kleine Firmen sinnvoll. Eine wichtige Funktion des externen Dienstleisters kann darin bestehen, bei der formalen Abwicklung und der Steuerung des Offshoring-Dienstleisters zu unterstützen. Der externe Dienstleister kann auch zwischen Fach- und IT-Abteilung vermitteln und die Projektsituation so ggf. entspannen.

Nachteile: Die Nachteile des Indirekten Offshoring im Vergleich mit Direktem Offshoring liegen in der höheren Anzahl der Beteiligten, im in der Regel teureren einheimischen Personal beim Vermittler und dadurch höheren Kosten.

Der Onshore-Offshore Kanal ist immer noch beschränkt und der kommunikative Flaschenhals des Projekts.

3. Dual-Shore-Offshoring

Entscheidende Faktoren: Dual-Shore-Offshoring basiert auf der Erkenntnis, dass die Weitergabe von Anforderungen über große Entfernungen zwischen Menschen mit unterschiedlichem kulturellen Hintergrund sehr schwierig ist. Daher werden die Teams gemischt. Dual-Shore-Offshoring ist dann erfolgreich, wenn es gelingt, echte Teams aufzubauen, die sich gegenseitig unterstützen. Nach Katzenbach und Smith zeichnet echte Teams aus, dass sie aus einer kleinen Anzahl von Leuten mit sich ergänzenden Fähigkeiten bestehen, die einen gemeinsamen Zweck verfolgen und für ihre Ziele und ihr Vorgehen geradestehen [Katzenbach u. Smith 1993].

Vorteile: Durch Dual-Shore-Offshoring wird der Onshore-Offshore-Kanal substanziell entlastet. Viele offene Fragen können direkt innerhalb der Teams geklärt werden. Wenn doch einmal teamübergreifende Kommunikation notwendig ist, kann sie – bei gemischten Teams – zwischen Mitgliedern eines Unternehmens erfolgen.

Durch den Onshore-Aufenthalt lernen Entwickler des Offshoring-Dienstleisters die zu unterstützenden fachlichen Prozesse besser kennen. Auch technisches Wissen kann direkt vermittelt werden. Onshore-Personal kann durch die Arbeit offshore die Probleme besser nachvollziehen, die bei der Umsetzung am entfernten Ort entstehen. Durch die engere Zusammenarbeit wächst in der Regel das gegenseitige Verständnis. Probleme können leichter behoben werden.

Nachteile: Durch die notwendigen Reisen und den Aufenthalt in einem fremden Land steigen die Kosten gegenüber Indirektem Offshoring.

Die direkte Zusammenarbeit mit Menschen aus einer anderen Kultur in einem Team erfordert Toleranz und eine gewisse Einarbeitungsphase. Für die am entfernten Standort arbeitenden Menschen kann dies in Verbindung mit der Reisetätigkeit und dem längeren Aufenthalt in einem zunächst fremden Land zu Problemen führen.

4.4.4 Eignung für Agiles Offshoring

Im bisherigen Verlauf der Untersuchung wurden die Kooperationsmodelle in Hinblick auf Offshoring im Allgemeinen diskutiert. In diesem Abschnitt wird explizit die Eignung der Modelle für Agiles Offshoring diskutiert. Zur systematischen Analyse wird dabei auf den Katalog von Themen, Problemen und Lösungsansätzen zurückgegriffen, der in Kapitel 3 entwickelt wurde. Im Rahmen der Diskussion werden die Auswirkungen der unterschiedlichen Kooperationsmodelle verglichen.

Kommunikation zwischen allen Beteiligten

In der Kategorie „Kommunikation zwischen allen Beteiligten“ lässt sich feststellen, dass beim Dual-Shore-Offshoring alle Probleme am besten gemeistert werden können. Durch die enge gemeinsame Arbeit in gemischten Teams fällt es leichter, einen Teamgeist und eine gemeinsame Vision zu entwickeln. Es ist ausreichend Raum für informelle Kommunikation und persönlichen Austausch gegeben. Kulturelle Unterschiede zeigen sich durch den direkten Umgang miteinander deutlich. Die Unterschiede können explizit gemacht und gemeinsam bewältigt werden. Gleiches gilt für Sprachschwierigkeiten. Dies ist ein wichtiger Unterschied zu Direktem und Indirektem Offshoring, bei denen diese Probleme hauptsächlich an der onshore-offshore Schnittstelle auftreten und unter Umständen längere Zeit verborgen bleiben können. Bei diesen Kooperationsmodellen fällt es insbesondere schwer, einen Teamgeist zu entwickeln, denn es handelt sich im Grunde um zwei getrennte Teams. Walther und Bazarova zeigen in [Walther u. Bazarova 2007] weitere mögliche Probleme verteilter Teams auf, z. B. dass Fehler oft auf den entfernten Partner geschoben werden.

Zusammenarbeit zwischen den Entwickler-Teams

Das Thema „Zusammenarbeit zwischen den Entwickler-Teams“ erhält erst mit Entwicklungstätigkeiten auf beiden Seiten eine wichtige Bedeutung. Beim Dual-Shore-Offshoring sind Koordination und Kooperation anspruchsvoller und anfangs schwieriger zu meistern. Die gemeinsame Übernahme von Verantwortung muss sich erst einspielen. Von der engen Zusammenarbeit der Teams profitiert der Wissenstransfer, da Informationen jetzt direkt vor Ort ausgetauscht werden können. Durch die Arbeit in gemischten Teams lernen die Beteiligten mehr, als es sonst möglich wäre. Der Wissenstransfer ist beim Direkten Offshoring noch schwieriger als beim Indirekten, bei dem der externe Dienstleister vermitteln kann. Die Aufgabenverteilung, die bei Direktem und Indirektem Offshoring klar geregelt ist, beansprucht beim Dual-Shore-Offshoring deutlich mehr Abstimmungsaufwand. Der Einsatz von agilen Techniken, welche die Wissensvermittlung und die Aufteilung von Aufgaben in Teilaufgaben unterstützen, kann hier helfen.

Einbeziehung des Kunden

Die bei agilen Methoden so wichtige „Einbeziehung des Kunden“ kann ebenfalls vom Dual-Shore-Offshoring profitieren. Je nach Gestaltung der Aufgaben onshore arbeiten die Offshore-Entwickler mehr oder weniger oft on-site beim Kunden. Der Kunde kann durch den persönlichen Umgang und den sichtbaren Arbeitsfortschritt einen guten Eindruck vom Offshoring-Dienstleister gewinnen. Durch Transparenz steigt das gegenseitige Vertrauen [Sheppard u. Sherman 1998]. Die Kommunikation kann direkt erfolgen, was die Anforderungsermittlung erleichtert. Da viel Wissen persönlich und mündlich und nicht nur indirekt und schriftlich weitergegeben werden kann, müssen

Spezifikationsdokumente nicht so umfangreich sein. Rückfragen können schnell geklärt werden. Beim Direkten Offshoring lassen sich alle diese Probleme schwerer meistern. Beim Indirekten Offshoring hängt die Etablierung einer guten Beziehung zwischen dem Kunden und dem Offshoring-Dienstleister zu einem großen Teil von der Kompetenz und den Fähigkeiten des externen Dienstleisters ab.

Ausbildung und Kenntnisse

Beim Dual-Shore-Offshoring können die Offshore-Entwickler direkt vor Ort in den Entwicklungsprozess eingeführt werden. Vor allem dann, wenn ein gemeinsames Kickoff-Treffen und eine erste gemeinsame Iteration onshore stattfinden, kann ein vergleichbarer Wissensstand unter allen Beteiligten erreicht werden. Wissens- und Qualifikationslücken können durch die Projektleitung erkannt und durch Schulungen oder Programmieren im Paar mit erfahreneren Kollegen verringert werden. Die Auswirkungen von Dual-Shore-Offshoring auf die Arbeitszufriedenheit sind schlecht abschätzbar. Zum einen kann die größere Eigenständigkeit, die höhere Transparenz des Entwicklungsprozesses und die intensive Zusammenarbeit mit den anderen Teams und dem Kunden die Zufriedenheit erhöhen, auch wenn dies kulturabhängig unterschiedlich wahrgenommen werden kann. Zum anderen kann die direkte Zusammenarbeit auch als fordernder und insbesondere durch die Reisetätigkeiten und den zeitweisen Aufenthalt in einem fremden Land als belastender wahrgenommen werden. Direktes und Indirektes Offshoring haben keinen Einfluss auf das Training der Entwickler und deren Arbeitszufriedenheit. Technische Schwächen der Offshore-Entwickler lassen sich im Vergleich mit Dual-Shore-Offshoring in diesen Kooperationsmodellen nur schwer verbessern. Zudem lassen sie sich häufig nur indirekt und spät durch Mängel an den gelieferten Artefakten aufdecken.

Infrastruktur

Alle Kooperationsmodelle stellen ähnliche Anforderungen an die Infrastruktur. Die beschriebenen Schwierigkeiten bezüglich der Kommunikationsinfrastruktur und heterogener Hard- und Software hängen nicht wesentlich vom gewählten Kooperationsmodell ab.

Projektmanagement und Qualitätssicherung

Das Projektmanagement und die Qualitätssicherung profitieren vom Dual-Shore-Offshoring. Durch die intensivere Zusammenarbeit wird die Fortschrittskontrolle vereinfacht, vor allem wenn oft neue Anwendungsversionen getestet werden können. Wenn Offshore-Kollegen bei der regelmäßigen Aufwandsschätzung zurate gezogen werden, kann eine realistischere Planung erreicht werden. Nicht zuletzt fällt es einfacher, mit persönlich bekannten Mitarbeitern über die Entfernung zusammenzuarbeiten als mit fremden. Beim Direkten Offshoring ist die Fortschrittskontrolle besonders schwierig.

Beim Indirekten Offshoring profitiert das Management und die Qualitätssicherung von der Erfahrung des externen Dienstleisters.

Tabelle 4.1 fasst die Ergebnisse für Direktes Offshoring, Indirektes Offshoring und Dual-Shore-Offshoring zusammen.

4.4.5 Empfehlungen für Agiles Offshoring

Aus den Untersuchungsergebnissen lassen sich konkrete Empfehlungen für geeignete Kooperationsmodelle in Abhängigkeit vom Projekttyp ableiten.

Direktes Offshoring bietet sich für Projekte an, bei denen die fachlichen Anforderungen einfach sind, genau und unmissverständlich festgelegt werden können und im Projektverlauf stabil sind. Dann besteht ein relativ geringer Kommunikationsbedarf. Das Projekt kann durch das Offshore-Team kostengünstig realisiert werden. Wenn diese Voraussetzungen jedoch nicht gegeben sind, ist das Risiko von Missverständnissen und Problemen und damit letztendlich des Scheiterns hoch.

Beim Vergleich der Beschreibung dieses Modells mit plangetriebenen Projekten (vgl. Abschnitt 3.2.2) wird deutlich, dass Direktes Offshoring in dieser Form mit agilem Vorgehen wenig zu tun hat. Trotzdem ist es sinnvoll, so weit wie möglich agile Techniken zur Erhöhung der Flexibilität und damit zur Risikominderung zu integrieren.

Seit einiger Zeit unternehmen Offshoring-Dienstleister Schritte, um Direktes Offshoring wieder attraktiver zu machen. Vor allem indische Unternehmen sind mit Niederlassungen verstärkt onshore in Ländern typischer Offshoring-Kunden präsent. Dadurch sinken die Kosten für die Kontaktaufnahme, den Vertragsabschluss sowie die Steuerung und Überwachung. Im Gegenzug steigen die Personalkosten. Mitarbeiter des Offshoring-Dienstleisters können persönlich vor Ort bei der Anforderungsermittlung mitwirken [Carmel u. Nicholson 2005]. Offshore-Unternehmen bilden verstärkt eigene Mitarbeiter mit speziellem Branchenwissen aus. Dadurch werden das gegenseitige Verständnis und die Wissensvermittlung bedeutend erleichtert. Das vom Kunden wahrgenommene Risiko sinkt.

In einer Studie aus dem Jahr 2007 wurde die Frage gestellt „Ist es für Sie wichtig, dass zusätzlich zu den Nearshore-/Offshore-Aktivitäten auch Leistungen in Ihrer Nähe, d. h. Onshore/On-site, erbracht werden, z. B. Service-Center ist vor Ort?“ 61,1% der Befragten antworteten mit Ja [Steria Mummert Consulting AG 2007]. Dies führt zum Modell des **Indirekten Offshoring**.

Mit Indirektem Offshoring lassen sich ähnliche Typen von Projekten meistern wie mit Direktem Offshoring. Auch hier ist der Umgang mit komplexen, sich ändernden Anforderungen sehr schwer. Der Kunde wird jedoch deutlich von Arbeit entlastet, wenn er einen erfahrenen externen Vermittler beauftragen kann. Durch die zusätzlichen, teuren Onshore-Kräfte steigen jedoch die Gesamtkosten von Projekten.

Kategorie	Themen	Direkt	Indirekt	Dual-Shore
Kommunikation zwischen allen Beteiligten	Teamgeist	−	−	+
	Gemeinsame Vision	○	○	+
	Kulturelle Unterschiede	○	○	+
	Sprachliche Verständigung	○	○	+
	Informelle Kommunikation	○	○	+
Zusammenarbeit zwischen den Entwickler-Teams	Kooperation und Koordination	○	○	±
	Wissenstransfer	−	○	+
	Gemeinsame Verantwortlichkeit	○	○	±
	Aufgabenverteilung	+	+	−
Einbeziehung des Kunden	Abstimmung mit dem Kunden	-	±	+
	Anforderungsermittlung und -spezifikation	-	○	+
	Aufbau von Vertrauen	-	±	+
Ausbildung und Kenntnisse	Training der Onshore- und Offshore-Entwickler	○	○	±
	Arbeitszufriedenheit	○	○	±
	Entwurfs- und Architekturkenntnisse	−	−	+
Infrastruktur	Kommunikationsinfrastruktur	○	○	○
	Hard- und Softwareausstattung	○	○	○
Projektmanagement und Qualitätssicherung	Management verteilter Teams und QS	○	+	+
	Aufwandsschätzung	○	○	+
	Fortschrittskontrolle	−	○	+

Tabelle 4.1: *Einfluss verschiedener Kooperationsmodelle auf das Offshoring der Anwendungsentwicklung. Ein „+“ bedeutet einen positiven Einfluss auf das Thema, ein „−“ einen negativen, ein „±“ einen Ausgleich von positiven und negativen Einflüssen und ein „○“ keinen nennenswerten Einfluss.*

Beim **Dual-Shore-Offshoring** kommt gegenüber Direktem und Indirektem Offshoring eine Teilung der Entwicklungsarbeit zwischen Onshore- und Offshore-Team hinzu, die mehr Koordinationsaufwand erfordert und Risiken bergen kann. Es ist mehr Personal- und Reiseaufwand nötig. Demgegenüber steht vor allem eine deutliche Verringerung der geografischen und zeitlichen Distanz zwischen beiden Seiten. Auf dem wichtigen Kommunikationskanal zwischen Onshore- und Offshore-Teams besteht diese Distanz zwar weiterhin, jedoch entfällt die Verteilung nach Funktionsbereichen, was insbesondere die Weitergabe von Anforderungen und den Wissenstransfer erleichtert. Durch die direkte Zusammenarbeit von Personal des Kunden, des externen Dienstleisters und des Offshoring-Dienstleisters lässt sich ein Umgang mit kulturellen Unterschieden finden. Dual-Shore-Offshoring ist damit das für Agiles Offshoring am besten geeignete Kooperationsmodell.

4.4.6 Anpassung des Dual-Shore-Modells auf mehrere globale Teams

Der Grundgedanke des Dual-Shore-Modells – Mitarbeiter werden zwischen den verteilten Entwicklerteams ausgetauscht, um die Kommunikationskanäle zu optimieren – ist auch auf agile global verteilte Projekte übertragbar, die nicht dem Offshoring zuzuordnen sind.

Bei mehr als zwei Teams stellt sich die Frage, wie das Modell sinnvoll angewendet werden kann. Es können zwei Fälle unterschieden werden, die sich nach der Kommunikationsstruktur richten. Häufig sind verteilte Projekte nach dem Hub-and-Spoke-Modell organisiert [Avritzer et al. 2008]. Dabei agiert ein zentrales Team als Drehkreuz und betreut mehrere verteilte Teams (vergleichbar mit Luftfahrt-Drehkreuzen). Das zentrale Team ist meist am Hauptsitz des Unternehmens angesiedelt, oft also onshore bei den Kunden. Die verteilten Teams kommunizieren hauptsächlich mit dem zentralen Team und nur selten oder gar nicht direkt untereinander. In diesem Fall ist zwischen dem zentralen Team und den einzelnen verteilten Teams jeweils das Dual-Shore-Modell anwendbar.

Ansonsten ist die Frage schwieriger zu beantworten. Allgemeine Hinweise sind nicht möglich. Die Ausgestaltung des Kooperationsmodells sollte sich danach richten, wie eng die einzelnen Teams zusammenarbeiten. Diese sollten dann untereinander Mitarbeiter austauschen.

Die Bezeichnung „Dual-Shore-Modell“ scheint bei mehr als zwei involvierten Standorten nicht mehr passend zu sein. „Multi-Shore-Modell“ wäre eine Alternative. Dieser Begriff wird jedoch schon von einigen Unternehmen als Bezeichnung für ihre Global Sourcing-Aktivitäten verwendet (siehe Abschnitt 2.2.1).

4.4.7 Realisierungsalternativen

In der Praxis sind die Kooperationsmodelle oft nicht in den hier diskutierten idealtypischen Formen anzutreffen. Die Kooperationsmodelle werden an die jeweiligen Projektsituationen angepasst. Zur Abrundung der Analyse werden daher mögliche Varianten und Alternativen bei der konkreten Realisierung in Offshoring-Projekten vorgestellt und bewertet. Wir beschränken uns dabei auf Varianten des Dual-Shore-Offshoring.

Auch wenn hier das Dual-Shore-Modell als Erweiterung des Indirekten Offshoring dargestellt wurde, ist es denkbar, dass der Kunde ohne den Umweg über den externen Dienstleister mit dem Offshoring-Dienstleister zusammenarbeitet. Dies kann sich beispielsweise anbieten, wenn der Offshoring-Dienstleister eine deutsche Filiale hat, deren Mitarbeiter über entsprechendes Fachwissen und Sprachkenntnisse verfügen. Dieser Weg ist meist nur bedingt zu empfehlen, da der Kunde ohne den externen Vermittler beim Dual-Shore-Offshoring dem Projekt mehr Personalressourcen zur Verfügung stellen und anspruchsvolle Kontroll- und Projektleitungsaufgaben selber übernehmen muss.

Wenn ein externer Dienstleister beteiligt wird, ist eine verbesserte Risikoverlagerung möglich. Bei dieser Variante agiert der externe Dienstleister nicht nur als Berater des Kunden, sondern auch als Schuldner der Leistung. Er schließt mit dem Offshoring-Dienstleister den Vertrag und übernimmt somit die Offshoring-spezifischen Risiken. Diese Form ist für den Kunden risikoärmer und bietet eine höhere Rechtssicherheit. Solche Kooperationsmodelle sind aber meist auch teurer.

Mehrere Varianten gibt es bei der Ausgestaltung der verteilten Entwicklung. Eine wesentliche Frage besteht darin, ob die Fach- und IT-Seite des Kunden und Onshore-Entwickler direkt mit den Offshore-Entwicklern kommunizieren können. Wenn dies trotz agiler Techniken wie gemeinsamen Iterationstreffen und regelmäßigem Feedback an sprachlichen und kulturellen Problemen scheitert, sollte die Kommunikation zwischen beiden Teams im Wesentlichen über die beiden Projektleiter ablaufen. Diese können sich aufeinander einstimmen und einen gemeinsamen Umgang mit den Offshoring-spezifischen Problemen finden. Dabei ist darauf zu achten, dass sich die Projektleiter nicht zum kommunikativen Flaschenhals entwickeln.

Häufig ist es sinnvoll, ein Entwicklerteam des Kunden in die Implementierung mit einzubeziehen. Zum einen können sie im Sinne eines Kunden-vor-Ort Fachwissen der Domäne einfließen lassen. Zum anderen lernen sie die technischen Details der Anwendung kennen und können nach dem Abschluss der Entwicklung mit weniger Einarbeitungsaufwand die Wartung oder Weiterentwicklung übernehmen. So kann die Abhängigkeit vom Offshoring-Dienstleister reduziert werden. Der direkteste Transfer von Fachwissen kann stattfinden, wenn Vertreter der Fachseite offshore arbeiten. Es ist meist allerdings nicht realisierbar, dafür Mitarbeiter des Kunden einzusetzen. Viele Auftraggeber haben schon mit der dauerhaften Abstellung eines qualifizierten Mitarbeiters für das Onshore-Team Probleme, da dieser währenddessen dem Fachbereich fehlt [Wolf et al. 2005]. Umso

schwieriger ist es, solche Mitarbeiter ins Ausland zu entsenden. Alternativ kann diese Rolle von Mitarbeitern des Offshoring-Dienstleisters wahrgenommen werden, nachdem sich diese on-site in die Fachlichkeit eingearbeitet haben.

4.5 Zusammenfassung

In diesem Kapitel wurde untersucht, wie Kooperationsmodelle die Probleme des Agilen Offshoring beeinflussen. Dazu wurden die Unterschiede zwischen den grundlegenden Begriffen Kommunikation, Koordination und Kooperation diskutiert und das 3K-Modell vorgestellt. Technische Mittel zur Unterstützung der Zusammenarbeit stehen auch dem Agilen Offshoring zur Verfügung, reichen aber oft nicht aus, um die Offshoring-spezifischen Probleme zu beherrschen. Zusätzlich sind organisatorische Vereinbarungen und konkrete Handlungsanweisungen nötig.

Nach einer Untersuchung von kulturellen Unterschieden wurden die drei Kooperationsmodelle Direktes Offshoring, Indirektes Offshoring und Dual-Shore-Offshoring mit ihren Auswirkungen auf die Kommunikationskanäle zwischen den Teams untersucht. Zur Veranschaulichung wurde eine geeignete grafische Notation entworfen und eingeführt. In der Untersuchung zeigte sich, dass Dual-Shore Offshoring mit Onshore- und Offshore-Entwicklungsteams sehr gute Voraussetzungen für ein flexibles Vorgehen und eine gute Zusammenarbeit aller Beteiligten bietet. Insbesondere kann gut mit Problemen umgegangen werden, die sich aus der geografischen, zeitlichen und organisatorischen Entfernung zwischen den Beteiligten ergeben. Aufgrund der vielen Vorteile des Dual-Shore-Modells und seiner besonderen Eignung für Agiles Offshoring, konzentrieren wir uns im weiteren Verlauf dieser Arbeit auf dieses Modell.

Ein Kooperationsmodell beschreibt nur die grundlegende Struktur der Zusammenarbeit. Zu einem praktisch nutzbaren Konzept gehören auch Details zu Kommunikations- und Abstimmungsprozessen. Aufgrund der Komplexität von Offshoring-Projekten werden weitere Empfehlungen und Vorgaben für die Prozesse der Kooperation benötigt. Beim Dual-Shore-Modell betrifft das vor allem die Zusammenarbeit zwischen den Entwickler-Teams. Es stellen sich mehrere Fragen auf der operativen Ebene:

- Wie kann eine gemeinsame Fachsprache für Kunden und Entwickler gefunden werden?
- Welche Rollen werden an welchem Brückenkopf wahrgenommen?
- Wie können die Aufgaben zwischen den Teams flexibel und sinnvoll aufgeteilt werden?
- Wie können sich die Teams flexibel abstimmen, z. B. bei der Übergabe von Aufgaben?
- Wie kann ein Ausgleich zwischen agiler, selbstverantwortlicher Arbeit und formeller Steuerung des Dienstleisters gefunden werden?

- Wie kann eine schrittweise Verlagerung von Aufgaben und Verantwortlichkeit vom Onshore- zum Offshore-Team erfolgen?

Im nächsten Kapitel wird eine Feldstudie vorgestellt, in der ein Lösungskonzept für die offenen Fragen der Zusammenarbeit entwickelt wird.

5 Architekturzentrierte Softwareentwicklung in global verteilten Projekten

In diesem Kapitel wird untersucht, mit welchen softwaretechnischen Techniken und Methoden agile global verteilte Entwicklung in der Praxis umgesetzt werden kann. Aufbauend auf den Erkenntnissen der letzten Kapitel zu typischen Problemen und Lösungsansätzen von agilen verteilten Entwicklungsprojekten und zu geeigneten Kooperationsmodellen wurde eine Feldstudie durchgeführt. Wichtige Entscheidungen im Entwicklungsprojekt der Feldstudie wurden vom Autor nach Methoden des Action Research beeinflusst. Während der Untersuchung stellte sich heraus, dass eine Konzentration auf Softwarearchitektur und architekturzentrierte Entwicklung den größten Erfolg versprach. Daher wurde dieser Ansatz weiter verfolgt und detailliert untersucht.

In den nächsten Abschnitten werden zunächst Grundkonzepte der Softwarearchitektur und das dieser Arbeit zugrunde liegende Verständnis von architekturzentrierter Softwareentwicklung erläutert. Danach wird ein Überblick über wissenschaftliche Untersuchungen zum Einsatz von Softwarearchitektur bei global verteilter Softwareentwicklung und Offshoring gegeben.

Anschließend werden die Forschungen im Empirieprojekt der Feldstudie vorgestellt und es wird beschrieben, wie architekturzentriertes Arbeiten in das Empirieprojekt eingeflossen ist. Aus den im Projekt gesammelten Erfahrungen werden Empfehlungen für geeignete Techniken der architekturzentrierten Entwicklung für agile global verteilte Entwicklung abgeleitet. Das Kapitel schließt mit weiterführenden Architekturthemen und einer Zusammenfassung.

5.1 Grundkonzepte der Softwarearchitektur

Zum besseren Verständnis der Einsatzzwecke und Besonderheiten von architekturzentrierter Entwicklung werden in diesem Abschnitt die wichtigsten Begriffe und Konzepte von Softwarearchitektur reflektiert.

Als wichtiges Artefakt ist die Softwarearchitektur gut für die Koordinierung und Abstimmung geeignet und damit auch für globale Softwareentwicklung interessant. Ein

erstes Beispiel zeigt die zentrale Bedeutung von Softwarearchitektur: Im Global Studio Project – einem Forschungsprojekt mit mehreren global verteilten studentischen Entwicklergruppen – wurden die einzelnen Arbeits- und Abstimmungsaufgaben detailliert erfasst. Die Auswertung ergab, dass die meisten teamübergreifenden Diskussionen im Projekt Design und Softwarearchitektur zum Thema hatten [Cataldo et al. 2007].

In den letzten Jahren ist die Softwarearchitektur als Schlüsselartefakt immer mehr in den Mittelpunkt der Softwaretechnik-Forschung gerückt. Die konzeptionellen Grundlagen bezüglich Konstruktion, Strukturierung, Entkoppelung und Beschreibung größerer Softwaresysteme wurden schon zwischen 1960 und 1980 in Werken von Dijkstra (Strukturierung, Schichtenbildung [Dijkstra 1968]), Myers (Kohäsion und Kopplung [Myers 1978]), Parnas (Geheimnisprinzip, Modularisierung [Parnas 1972]), Hoare (Abstraktion, Korrektheit von Algorithmen [Hoare 1969]) und anderen gelegt. In den 90er Jahren wurde das Gebiet in Forschung und Praxis vorangetrieben und später unter dem Begriff Softwarearchitektur diskutiert [Posch et al. 2004].

Bei der Klärung des Begriffs Softwarearchitektur halten wir uns an die mittlerweile weithin anerkannte Definition der ANSI/IEEE (ANSI/IEEE-Standard 1471-2000 on the Recommended Practice for Architectural Description of Software-Intensive Systems, [ANSI/IEEE-Standard-1471 2000]). Im Folgenden wird zur sprachlichen Vereinfachung an mehreren Stellen verkürzt der Begriff Architektur verwendet. Mit dem Begriff ist immer die Softwarearchitektur gemeint, im Gegensatz etwa zur Architektur von Gebäuden.

Definition Softwarearchitektur:

Software architecture subsumes "the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution".

Eine Erklärung für die weite Verbreitung dieser Definition ist ihre Allgemeinheit. Eine ebenfalls breit akzeptierte Definition aus [Bass et al. 2003, S. 21] ist etwas konkreter: "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."

Nach beiden Definitionen besitzt jedes Softwaresystem eine Architektur. Wenigstens implizit ist immer eine Softwarearchitektur vorhanden. Die Architektur sollte jedoch explizit entworfen und dokumentiert werden, um nützlich zu sein [Reussner u. Hasselbring 2009]. Die Erfahrung und das Können des Architekten spielen dabei eine große Rolle. Eine Softwarearchitektur ist immer ein Modell des Systems. Für dasselbe System sind verschiedene Architekturen möglich. Der Softwarearchitekt ist dafür zuständig, eine geeignete Architektur zu entwerfen.

Ein System wird durch eine Softwarearchitektur auf einer abstrakten Ebene beschrieben. Von Details wird dabei abstrahiert. So ist allgemein von Softwarekomponenten oder -elementen und ihren Beziehungen die Rede. Wie in [Bass et al. 2003] festgestellt wird, kommt es dabei im Sinne des Geheimnisprinzips[1] nur auf deren von außen sichtbare, nicht auf ihre inneren Eigenschaften an. Details einzelner Klassen (bei objektorientierter Programmierung) und konkrete Algorithmen sind somit für die Architektur nur bedeutsam, wenn sie für die Schnittstelle zwischen den Komponenten und externen Systemen relevant sind.

Perry und Wolf vergleichen in ihrem grundlegenden Artikel vom Anfang der 90er Jahre Softwarearchitektur mit anderen bekannten Architekturbegriffen: Hardwarearchitektur, Architektur von Netzen und Gebäudearchitektur [Perry u. Wolf 1992]. Während diese damals schon gut etabliert waren, gab es erst wenige Forscher, die sich explizit mit Softwarearchitektur beschäftigten. Dies hat sich in den letzten Jahren geändert. Um die heutige praktische Bedeutung von Softwarearchitektur für die Anwendungsentwicklung einschätzen zu können, hilft ein Blick auf die Arbeit von Kari Smolander. Er hat zehn Jahre nach Perry und Wolf aus 19 Interviews mit Architekten, Designern und Managern vier Metaphern für Softwarearchitekturen gewonnen [Smolander 2002]:

Architektur als Blaupause: Die Architektur gibt die Struktur des zu entwickelnden Systems vor. Sie dient als Spezifikation auf höchster Ebene durch Architekten und Designer, auf deren Grundlage die Entwickler das System im Detail implementieren können. Dazu muss die Architektur hinreichend formal und genau beschrieben werden.

Architektur als Literatur: Die Architektur dient zur Dokumentation, falls das System später angepasst oder erweitert werden muss. Sie hält technisches Wissen über die Struktur und Entwurfsgrundlagen des Systems fest und ermöglicht damit dessen Verständnis und ggf. eine Wiederverwendung von Systemkomponenten.

Architektur als Sprache: Über die Architektur wird eine Sprache für ein gemeinsames Verständnis des Systems definiert. Dies ermöglicht es unterschiedlichen Akteuren, sachkundig über die Strukturen und Entwurfsentscheidungen des derzeitigen Systems und späterer Ausbaustufen miteinander zu sprechen. Dazu sollte die Architektur verständlich beschrieben sein.

Architektur als Entscheidung: Die Architektur entsteht als Folge von teilweise unter Zielkonflikten getroffenen Entwurfsentscheidungen. Sie legt die erforderlichen Ressourcen für das System fest: benötigte Arbeitskräfte, deren besondere Fähigkeiten, sinnvolle Arbeitsteilungen, Lizenzkosten für von Dritten bezogene Komponenten u. Ä. Des Weiteren bildet sie die Basis für zukünftige Entscheidungen.

[1] Das Geheimnisprinzip (engl. information hiding) geht auf David Parnas zurück [Parnas 1972]. Es besagt, dass die internen Eigenschaften von Programmeinheiten vor dem unkontrollierten Zugriff von außen geschützt werden sollten.

Bevor wir uns genauer mit architekturzentrierter Softwareentwicklung beschäftigen, welche die zentrale Rolle der Softwarearchitektur anerkennt, werden im Folgenden wichtige Themen der Softwarearchitektur betrachtet: Architektursichten und -stile, die informelle und regelbasierte Dokumentation von Softwarearchitektur, die Evolution von Architekturen und das Verhältnis von Agilität und Softwarearchitektur.

5.1.1 Architektursichten und -dokumentation

Verschiedene Akteure haben an unterschiedlichen Aspekten einer Softwarearchitektur Interesse. Softwarearchitekturen können komplex sein und aus mehreren textuellen und grafischen Modellen bestehen. Um diesen beiden Tatsachen Rechnung zu tragen, ist im ANSI/IEEE-Standard 1471-2000 vorgesehen, dass mehrere Modelle zu einer Sicht auf das System (engl. view) für einen bestimmten Zweck oder eine bestimmte Zielgruppe zusammengefasst werden können [ANSI/IEEE-Standard-1471 2000].

Eine Schwierigkeit besteht darin, sich gut ergänzende Sichten zu entwickeln und konsistent zu halten. Philippe Kruchten hat 1995 mit dem 4+1 Sichten-Modell erstmals fünf Sichten zu einem Modell verbunden [Kruchten 1995], siehe Abbildung 5.1.

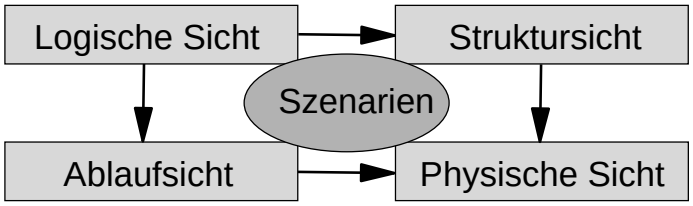


Abbildung 5.1: *Das 4+1 Sichten-Modell, nach [Kruchten 1995, S. 43]*

Die logische Sicht baut auf der Analyse auf. Sie zeigt die Funktionalität des Systems für Endbenutzer. Die Struktursicht stellt die Architekturelemente und ihre Schnittstellen für Programmierer und Manager dar. Die Ablaufsicht zeigt die dynamischen Prozesse. Die physische Sicht richtet sich mit ihrer Darstellung der Hardwarekomponenten und Netzwerke an Techniker. Übergreifende Szenarios zeigen anhand von Anwendungsfällen, wie die Architekturelemente zusammenspielen.

Es lassen sich verschiedene Ebenen von Architekturelementen betrachten. Auf einer detaillierten Ebene spricht man in der heutzutage vorherrschenden objektorientierten Programmierung von Klassen mit den wichtigsten Beziehungen „Benutzung“ und „Vererbung“. Auf einer abstrakteren Ebene werden Subsysteme betrachtet, auch Komponenten oder (etwas veraltet) Module genannt. Die wichtigsten Beziehungen zwischen Subsystemen sind „Benutzung“ und „Enthaltensein“ [Lilienthal 2008].

In den letzten Jahren haben sich aus dem 4+1 Sichten-Modell und anderen Modellen drei Sichten herausgebildet, die von den meisten Methoden zur Architekturbeschreibung unterstützt werden [Reussner u. Hasselbring 2009]:

Statische Sicht: Diese Sicht beschreibt die Struktur des Systems, die sich während der Laufzeit nicht ändert. Sie teilt das System in Elemente auf und stellt diese und ihre Beziehungen untereinander dar. Die statische Sicht ist die wichtigste, da die in ihr definierte Struktur des Systems als Grundlage für die anderen Sichten dient.

Dynamische Sicht: Bei dieser Sicht steht das Systemverhalten zur Laufzeit im Vordergrund. Beispielsweise werden die Kontrollflüsse des Systems sowie Interaktionen zwischen Komponenten bzw. Teilsystemen abgebildet.

Verteilungssicht: Diese Sicht bildet strukturelle Elemente des Systems auf Infrastruktur- und Hardwareeinheiten (Prozessoren, Netzwerke u. Ä.; auch: Deployment-Sicht) sowie auf organisatorische Einheiten (Entwickler, Teams oder externe Hersteller) ab.

Hinzu kommt bei einigen Autoren die **fachliche Sicht**, die den technischen Blickwinkel der anderen drei Sichten um den fachlichen Bezug zum Anwendungsgebiet ergänzt. Ausgehend von den Szenarios von Kruchten [Kruchten 1995] beziehen neuere Ansätze die Fachbegriffe und den Einsatzkontext der Anwendung mit ein und stellen die Verbindung zur technisch geprägten Architektur her (vgl. z. B. [Züllighoven 2004] und Abschnitt 5.2.2 in dieser Arbeit).

Die Beschreibung von Architekturen mit ihren verschiedenen Sichten ist eine wichtige und nicht einfache Aufgabe, da sie für verschiedene Akteure verständlich sein muss [Clements et al. 2002]: Architekten müssen mit Analysten die fachlichen Anforderungen und mit technischen Experten des Auftraggebers die technischen Anforderungen klären. Entwickler implementieren auf Grundlage der Architektur das System. Für Tester und Systemintegratoren müssen die Schnittstellen der Komponenten ersichtlich sein. Wartungsprogrammierer sollen die Auswirkungen von nötigen Änderungen abschätzen können. Entwickler angebundener Systeme interessieren sich für die externen Schnittstellen und Protokolle. Manager nutzen die Architekturbeschreibung, um die Arbeit in einzelne Pakete aufzuteilen, benötigte Ressourcen einzuplanen und den Projektfortschritt zu messen. Das Qualitätssicherungsteam muss sicherstellen, dass die Implementierung den Vorgaben der Architektur entspricht.

Zur Dokumentation der Architektursichten wurden Architekturbeschreibungssprachen entwickelt. Zur Quasi-Standardsprache für die heutzutage vorherrschenden objektorientierten Systeme ist die Mitte der 90er Jahre entstandene Unified Modeling Language (UML) [Object Management Group 2009] geworden. Diese Sprache stellt eine Reihe von Struktur- und Verhaltensdiagrammen zur Verfügung, mit denen die Sichten geeignet dargestellt werden können. Die statische Sicht wird typischerweise durch Klassendiagramme abgebildet. Für die dynamische Sicht bieten sich Aktivitäts-, Sequenz- und

Zustandsdiagramme an. Für die Verteilungssicht gibt es einen eigenen Diagrammtyp, das sogenannte Verteilungsdiagramm. Die fachliche Sicht kann auf eine recht abstrakte Weise in Anwendungsfalldiagrammen erfasst werden. Zur Konkretisierung der fachlichen Sicht bieten sich Modelltypen an, die nicht aus der UML stammen, sondern z. B. aus der Geschäftsprozessmodellierung (siehe die Beschreibung der Methode eGPM in Abschnitt 4.3.2 als ein Beispiel).

Eine wichtige Frage ist, wie viel Dokumentation der Architektur nötig ist. Während bei plangetriebenen Projekten häufig sehr umfangreich dokumentiert wird, versucht man bei agilen Methoden, leichtgewichtiger vorzugehen. So meint David Parnas [Ågerfalk u. Fitzgerald 2006, S. 29]: "The real grand challenge is not to find ways to avoid documenting, but to find ways to produce useful documents—documents that take time but save more time. We will find that real agility comes from good design that is well documented in precise, lean documentation." Bei der Diskussion der Feldstudien im weiteren Verlauf dieser Arbeit wird diese Frage wieder aufgegriffen.

5.1.2 Architekturstile und -analyse

Damit nicht für jedes System eine komplett neue Architektur entworfen werden muss, können bewährte Architekturen als Vorlage benutzt werden. Als eine solche Vorlage dienen Architekturstile. Dabei handelt es sich um Muster für eine Reihe gleichartiger Systeme [Bass et al. 2003], [Shaw u. Garlan 1996]. Formal kann ein Architekturstil als die Festlegung von Syntax und Semantik betrachtet werden. Die Syntax bestimmt Elemente, Beziehungen und mögliche Kombinationen. Die Semantik definiert deren Bedeutung und Verwendung [Abowd et al. 1993]. Bekannte Stile sind z. B. der Pipes & Filter-Stil und die Schichtenarchitektur (siehe [Buschmann et al. 1996] für eine ausführliche Diskussion dieser Stile).

Ein Architekturstil schränkt die freie Kombinierbarkeit von Architekturelementen bewusst ein. Zulässige Kombinationen werden durch Architekturregeln festgelegt. Diese lassen sich am Beispiel der Schichtenarchitektur gut erläutern. Bei dieser werden die Quelltextelemente Schichten zugeordnet, welche die möglichen Beziehungen beschränken. So sind Aufwärtsbeziehungen von Elementen einer niedrigeren Schicht in eine höhere Schicht verboten. Bei strengen Schichtenarchitekturen dürfen bei Abwärtsbeziehungen auch keine Schichten übersprungen und nur Elemente benutzt werden, die sich in derselben oder der direkt darunter befindlichen Schicht befinden [Bass et al. 2003]. Neben diesen Regeln, die sich aus dem Architekturstil ergeben, sind auch Regeln zu beachten, die sich aus der Soll-Architektur des Systems ergeben. Die Soll-Architektur wird von den Architekten des Systems festgelegt. Sie beschreibt die gewünschte Struktur, indem sie den gewählten Architekturstil detailliert und ihn um weitere Vorgaben und zusätzliche Regeln anreichert.

Referenzarchitekturen stellen besondere, komplexere Architekturstile dar [Lilienthal 2008]. Bei ihnen werden Architekturelemente in verschiedene Elementarten eingeteilt,

für die jeweils spezifische Regeln für ihre Zusammensetzung und die Interaktion mit anderen Elementarten definiert werden. Es lassen sich drei Regelarten unterscheiden: Elementregeln spezifizieren die Schnittstelle oder den Aufbau einer Elementart. Gebotsregeln schreiben notwendige Beziehungen zwischen bestimmten Elementarten vor, während Verbotsregeln diese verbieten. Diese Regeln können detailliert und komplex sein, sodass eine automatisierte Überprüfung sinnvoll ist [Becker-Pechau et al. 2006]. Ein gutes Beispiel für eine Referenzarchitektur ist die Modellarchitektur des Werkzeug & Material-Ansatzes, siehe S. 134.

Um Architekturregeln überprüfen zu können, ist eine Zuordnung von Quelltextelementen zu Architekturelementen erforderlich. Murphy, Notkin und Sullivan beschreiben mit ihren Reflexionsmodellen, wie die Lücke zwischen Quelltext und abstrakten Modellen überbrückt werden kann, siehe [Murphy et al. 1995] und Abbildung 5.2. Hier nennt

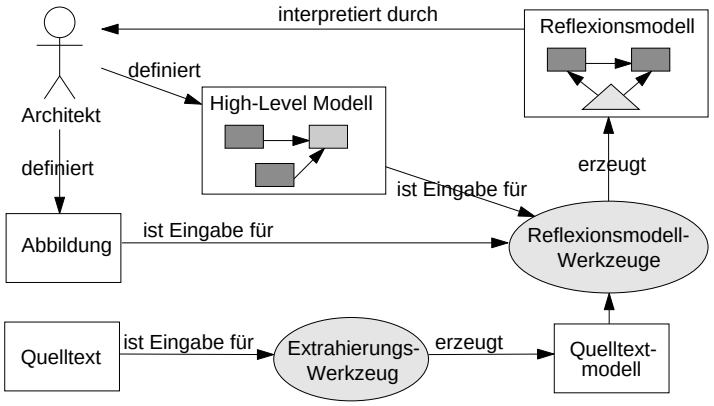


Abbildung 5.2: *Reflexionsmodelle zur Verbindung von Quelltext- und abstrakteren Modellen, nach [Murphy et al. 1995, S. 19]*

sich die Soll-Architektur High-Level Modell. Dieses Modell wird zusammen mit den Abbildungsregeln vom Architekten des Systems definiert. Aus dem Quelltext wird mithilfe von geeigneten Werkzeugen ein Quelltextmodell extrahiert. High-Level und Quelltextmodell werden über die Abbildungsregeln zusammengebracht. Daraus entsteht ein Reflexionsmodell (auch als Ist-Architektur bezeichnet), das – vom Architekten sinnvoll interpretiert – Rückschlüsse über die Qualität des implementierten Systems erlaubt. Genauer gesagt, lässt sich die *innere* Qualität der Anwendung bewerten. Diese wird auch Produktqualität genannt. Sie bezieht sich auf die Konstruktion des Systems und lässt sich anhand innerer Qualitätsmerkmale überprüfen. Über die Gebrauchsqualität, welche die Eignung eines Systems für die Verwendung im Einsatzkontext und seine Benutzbarkeit betrifft, kann keine Aussage getroffen werden [Züllighoven u. Raasch 2006].

Bei der Überprüfung von Architekturen ist man schon bei mittelgroßen Systemen auf die Unterstützung von Softwarewerkzeugen angewiesen, um die Komplexität be-

herrschen zu können. Damit lassen sich auch Erkenntnisse über große Systeme, die nicht mehr von einer einzigen Person in ihrer Gesamtheit verstanden werden können, gewinnen [Bischofberger et al. 2004]. Neben der Prüfung, ob die Architekturregeln eingehalten werden, ob also die Ist-Architektur zum vorgegebenen Stil passt und sich die vorgegebene Architektur im Quelltext wiederfinden lässt, erfüllt eine Architekturanalyse noch einen weiteren Zweck. Über Metriken und Untersuchungen im Detail lassen sich gewisse Qualitätsaspekte überprüfen und beispielsweise unerwünschte zyklische Abhängigkeiten, zu große und komplexe Elemente, mögliche Flaschenhälse und andere potenziell problematische Eigenschaften finden, die unabhängig vom gewählten Architekturstil und der Soll-Architektur sind. Eine durch Methoden der Architekturanalyse geprüfte Architektur ermöglicht es, für ein System zu belegen, dass es bestimmten Anforderungen bezüglich seines Verhaltens und seiner inneren und äußeren Qualität genügt [Bass et al. 2003].

5.1.3 Evolution einer Architektur

In agilen Projekten sind Ist- und Soll-Architektur nicht unveränderlich. Sie entwickeln sich in enger Verzahnung weiter. Zu keinem Zeitpunkt ist die Architektur vollständig spezifiziert. Selbst nach dem Projektende kann sie durch Wartungsarbeiten und andere Pflegemaßnahmen noch verändert werden. Dies birgt die Gefahr, dass die Soll-Architektur nicht mehr gut genug gepflegt wird und die Ist-Architektur immer weiter von ihr abweicht. Damit verliert die Soll-Architektur ihren Nutzen für die Ist-Architektur, die sich ohne sinnvolle Vorlage schon nach kurzer Zeit nicht länger wartbar und weiterentwickelbar halten lässt.

Eine Investition in die Pflege und den Erhalt der Soll-Architektur zahlt sich daher aus. Die fortwährende Weiterentwicklung der Architektur bezeichnet man als Architekturevolution [Reussner u. Hasselbring 2009]. Die Evolution der Architektur bedeutet einen permanenten Kommunikationsaufwand. Es reicht nicht aus, die Soll-Architektur zu Beginn des Projekts zu erläutern. Änderungen der Architektur müssen in regelmäßigen Abständen kommuniziert werden. Die Einhaltung von Architekturregeln in der Ist-Architektur sollte fortlaufend überprüft werden, um die Qualität zu sichern.

5.1.4 Architektur und Agilität

Die Themen Softwarearchitektur und Agilität stehen in einem gewissen Spannungsfeld zueinander. Abrahamsson et al. haben dieses in einem kürzlich erschienenen Artikel aufgegriffen und diskutiert [Abrahamsson et al. 2010].

Eine wichtige Basis von allen agilen Methoden besteht darin, dass sie adaptiv sind und die Akteure darin unterstützen, den Entwicklungsprozess permanent an sich verändernde Gegebenheiten anzupassen. Mittels Softwarearchitektur wird dagegen ein Rahmen für die Entwicklung gesetzt. Zukünftige Erweiterungen an der Struktur der

Anwendung sollen zumindest zu einem gewissen Teil vorausgeplant werden. Anhänger agiler Entwicklung sehen diesen Entwurf der Softwarearchitektur im Voraus kritisch, weil es die Entscheidungsmöglichkeiten der Entwickler einengen kann. Die Architektur sollte ihrer Meinung nach schrittweise durch kleine Änderungen in jeder Iteration entstehen.

Nach Ansicht des Autors dieser Arbeit kann in der Praxis gut ein Mittelweg gefunden werden: Es sollte nicht schon die komplette Softwarearchitektur zu Beginn eines Projekts entworfen werden. Andererseits sollte sie auch nicht erst während der Entwicklung entstehen. Selbst wenn die Vorgabe einer Softwarearchitektur die Agilität etwas einschränkt, überwiegen in der Regel die Vorteile. Dies trifft insbesondere auf global verteilte Entwicklung zu, bei der ohnehin mehr Vorausplanung und mehr Dokumentation erforderlich ist.

5.2 Architekturzentrierte Softwareentwicklung

Für diese Arbeit ist vor allem der Umgang mit Softwarearchitektur im Entwicklungsprozess interessant. Daher werden hier ausführlich Eigenschaften und Konzepte der architekturzentrierten Softwareentwicklung (AZSE) diskutiert, bei der die Softwarearchitektur eine herausragende Stellung in Entwicklungsprojekten einnimmt.

In diesem Abschnitt wird zunächst die historische Entwicklung architekturzentrierten Arbeitens beschrieben. Anschließend wird auf der Grundlage von Forschungsleistungen an der Universität Hamburg beschrieben, was architekturzentriertes Arbeiten ausmacht. Zuletzt werden die positiven Auswirkungen von Architekturzentrierung in der Softwareentwicklung zusammengestellt.

5.2.1 Historische Entwicklung

Das Konzept und die Praxis der architekturzentrierten Entwicklung haben sich hauptsächlich aus zwei parallelen Ursprüngen entwickelt. Zum einen ist dies das Architecture-Centered Software Development des Rational Unified Process/Unified Process, zum anderen das Architecture-Based Development des Software Engineering Institute der Carnegie Mellon University. Ergänzend wird hier die Arbeit von Paulish [Paulish 2001] aufgegriffen, der das Projektmanagement mit einbezieht.

Architecture-Centered Software Development im Rational Unified Process/Unified Process

Im Rational Unified Process (RUP) wurde 1995 erstmals von architekturzentrierter Entwicklung (engl. architecture-centered software development) gesprochen. Der RUP beruht auf dem generischen Unified Process (UP). Er wurde von der Firma Rational

Software entwickelt [Scott 2002]. Diese Firma wurde 2003 von IBM aufgekauft. Eine angepasste Version des RUP wird heutzutage von IBM Rational auch bei GSD eingesetzt [Parvathanathan et al. 2007].

Der RUP/UP wird als anwendungsfallgetriebener, iterativer, inkrementeller und *architekturzentrierter* Prozess beschrieben [Jacobson et al. 1999]. Er besteht aus vier Phasen: der Konzeptionsphase (engl. inception), der Entwurfsphase (elaboration), der Konstruktionsphase (construction) und der Übergabephase (transition). Die Softwarearchitektur umfasst beim RUP/UP eine Reihe von maßgeblichen Entscheidungen über die Gestaltung des zu entwickelnden Systems. Sie soll im gesamten Entwicklungsprozess als zentrales Artefakt verwendet und aktuell gehalten werden.

Als Hauptvorteile einer architekturzentrierten Entwicklung werden im RUP/UP genannt [Kruchten 2003]:

- Mithilfe von Architektur können ein Projekt gesteuert, dessen Komplexität beherrscht und die Systemintegrität erhalten werden.
- Architektur ermöglicht eine effektive Wiederverwendung von erfolgreichen Strukturen in mehreren Projekten.
- Architektur bildet eine Grundlage für das Projektmanagement.
- Komponentenbasierte Entwicklung wird durch Architektur erleichtert.

Im RUP/UP wird die Architektur als Komplement zu Anwendungsfällen gesehen, die im RUP/UP eine zentrale Rolle spielen. Die Anwendungsfälle beschreiben die Funktionalität eines Systems, die Architektur seine Struktur. Beim Vorgehen nach RUP/UP sollte eine zu den Anwendungsfällen passende Architektur gewählt werden. Umgekehrt kann eine vorhandene Architektur zu einer Anpassung von Anwendungsfällen führen.

Die Architektur wird iterativ in der Entwurfsphase auf Basis der wichtigsten Anwendungsfälle erstellt. Am Ende dieser Phase stehen ihre Grundeigenschaften (engl. architecture baseline) fest. Anhand einer ersten, nur intern verwendeten Version der Anwendung kann die Architektur beurteilt und verbessert werden. Die Grundeigenschaften der Architektur sollten unverändert bleiben, da spätere Anpassungen aufwendig sein können. In der Konstruktionsphase wird die Architektur auf Basis der internen Version umgesetzt. Der iterative Prozess des RUP/UP drückt sich darin aus, dass die Phasen zyklisch wiederholt werden. Die Architektur wird so fortwährend angepasst und weiterentwickelt.

Um eine geeignete Architektur entwerfen zu können, benötigen die Architekten fachliches und technisches Wissen. Die Sichten der Architektur werden in mehreren UML-Diagrammen dokumentiert. Die Beschreibung soll einen Plan des Systems darstellen, keine komplette Spezifikation. Daher behandelt sie das System auf hoher Ebene ohne Details. Diese Dokumentation wird während der Entwicklung des Systems bei Bedarf aktualisiert [Jacobson et al. 1999].

Architecture-based development am SEI, Carnegie Mellon University

Am Software Engineering Institute der Carnegie Mellon University haben Len Bass und Rick Kazman parallel zum RUP/UP das Konzept der architekturbasierten Entwicklung (engl. architecture-based development) entwickelt [Bass u. Kazman 1999].

Dabei wird im Gegensatz zu herkömmlicher Entwicklung die Softwarearchitektur in den Mittelpunkt der Entwicklung gestellt. Von einer Konzentration auf die Softwarearchitektur versprechen sich die Wissenschaftler eine Verbesserung des entwickelten Systems in Bezug auf Performanz, Änderbarkeit, Sicherheit und Verlässlichkeit. Genauso wie die Autoren des RUP/UP gehen Bass und Kazman davon aus, dass Projektteams mithilfe eines architekturzentrierten Vorgehens auch große, komplexe Systeme beherrschen können. Softwarearchitektur dient während des gesamten Entwicklungsprozesses als Blaupause für alle Aktivitäten, u. a. für die Arbeitsteilung, die Integrations- und Testplanung, das Konfigurationsmanagement und die Dokumentation.

Bass und Kazman orientierten sich bei der Beschreibung architekturzentrierter Entwicklung am tatsächlichen Umgang mit Architektur in mehreren großen Projekten aus der Praxis. Daraus haben sie folgende idealisierte Prozessschritte abgeleitet:

1. **Ermittlung der Anforderungen an die Architektur:** Die Ermittlung der Anforderungen wird maßgeblich von der technischen Umgebung und der Erfahrung des Architekten bestimmt. Zuerst werden die Haupttreiber der Architektur bestimmt (z. B. Einbindung einer Datenbank). Anschließend werden daraus die Anforderungen an die Architektur abgeleitet. Bass und Kazman gehen davon aus, dass maximal 20 Anforderungen ausreichen. Dies sind deutlich weniger als die funktionalen Anforderungen. Anforderungen an die Architektur und funktionale Anforderungen sind gleich wichtig. Für beide Arten werden Anwendungsfälle geschrieben, welche die Anforderungen verschiedener Akteure beinhalten. Für die Architektur gibt es explizite Qualitätsszenarios, die Vorgaben in den Bereichen Änderbarkeit, Performanz, Sicherheit und Zuverlässigkeit definieren.

2. **Entwurf der Architektur:** Auf Grundlage der Anforderungen trifft und begründet der Architekt Entwurfsentscheidungen. Dabei kann er sein Wissen über Architekturstile, Entwurfsmuster und spezielle Werkzeuge anwenden. Er setzt iterativ verschiedene Anforderungen um und entwirft und verbessert in diesem Prozess verschiedene Architektursichten. Bass und Kazman orientieren sich bezüglich der Sichten am 4+1 Sichten-Modell von Kruchten (siehe S. 122). Die Qualitätsszenarios dienen zur Validierung des Entwurfs.

3. **Dokumentation der Architektur:** Der Architekt ist auch für die Dokumentation der Architektur zuständig. Alle Architekturdokumente sollten stets aktuell und für alle Akteure zugreifbar sein. Neben einer technischen Beschreibung der Architektur in verschiedenen Sichten sollte aus der Dokumentation auch hervorgehen, wie die fachlichen Anwendungsfälle in der Architektur berücksichtigt werden.

4. **Analyse der Architektur:** Die Architekturanalysen haben zum Ziel, potenzielle Risiken aufzudecken und die Umsetzung der Anforderungen an die Qualität zu überprüfen. Neben dem Architekten sollten sich bei großen Systemen auch die Designer der größeren Subsysteme, Tester, Administratoren, Anwender, Kunden und Manager an den Analysen beteiligen. Bei fragebasierten Analysen wird eine Liste von Fragen oder Szenarios abgearbeitet. Bass und Kazman empfehlen dafür die Architecture Tradeoff Analysis Method (ATAM), vgl. [Bass u. Kazman 1999]. Eine Alternative ist die quantitative Untersuchung ausgewählter Eigenschaften. Diese Methode bietet sich beispielsweise bei der Performanz- oder Zuverlässigkeitsanalyse an.

5. **Umsetzung der Architektur:** Bei der Implementierung sollten grundlegende Prinzipien der Softwaretechnik und des Projektmanagements angewendet werden. Architekturzentrierte Entwicklung hilft bei der organisatorischen Aufteilung der Teams. Bass und Kazman zitieren hierfür Conway (vgl. die Diskussion von Conway's Law auf Seite 74). Die Teams entwickeln einzelne Subsysteme. Wenn das System gut entworfen wurde, ist zwischen den Teams nur wenig Kommunikation notwendig.

6. **Erhalt der Architektur:** Die Architektur muss fortwährend an Änderungen angepasst werden. Dies ist recht einfach, wenn sie gut dokumentiert ist. Ist dies nicht der Fall, so muss sie eventuell unter Beteiligung des ursprünglichen Architekten rekonstruiert werden. Dies ist auch dann nötig, wenn ein bestehendes, schlecht dokumentiertes System weiterentwickelt, in anderen Zusammenhängen verwendet oder in Bezug auf seine technischen Eigenschaften beurteilt werden soll.

Beim Vergleich dieses idealisierten Prozesses mit dem von RUP/UP fällt auf, dass er weniger stark iterativ-inkrementelle Ansätze berücksichtigt. Bass und Kazman sehen nur in den Schritten zwei bis vier ein iteratives Vorgehen vor [Bass u. Kazman 1999]. Sobald die Architektur erstellt und validiert worden ist, wird sie umgesetzt und nicht mehr wesentlich verändert.

Architekturzentriertes Management von Softwareentwicklungsprojekten

Dan Paulish erweitert die Verwendung von Softwarearchitektur auf andere Aktivitäten in Softwareentwicklungsprojekten, die nicht direkt mit Entwurf und Implementierung zusammenhängen [Paulish 2001]. Dabei bezieht er trotz der Bezeichnung „architekturzentriertes Management“ auch allgemeine, nicht an Architektur orientierte Techniken und Ratschläge für das Management mit ein.

Basierend auf Erfahrungen bei Siemens hat Paulish eine Softwareentwicklungsmethodik zur Planung von Projekten entwickelt. Dabei vertritt er die These, dass Softwarearchitektur nicht nur für den Architekten eine zentrale Bedeutung besitzt, sondern auch für den Projektleiter. Eine Übersichtsarchitektur (engl. high-level architecture) dient zur

Projektplanung und Aufwandsschätzung, als Basis für Entwickler und die Projektorganisation, für die Qualitätssicherung und die Iterations- und Releaseplanung.

Paulish beschreibt einen – nach seiner Einschätzung – mittelgewichtigen Prozess, der zwischen agilen, leichtgewichtigen und plangetriebenen, schwergewichtigen Prozessen liegt. So erfolgt der Entwurf der Architektur, bevor der Großteil des Quelltextes geschrieben wird. Die Entwicklung findet dann iterativ-inkrementell statt. Die Grundannahmen der Architektur werden in der ersten Iteration anhand eines vertikalen Prototyps überprüft. Ein Großteil der Projektplanung erfolgt erst nach dem Entwurf der Systemarchitektur, da der Projektplan, der Zeitplan und der Ressourcenplan stark von der Architektur abhängen. Der Projektleiter und der (Chef-)Architekt arbeiten eng zusammen. In kleinen Projekten können diese Rollen auch personell zusammenfallen. Paulish geht aber implizit meist von größeren und sehr großen Projekten aus. In diesen ist der Chef-Architekt maßgeblich für den Erfolg verantwortlich. Er muss eine Vision vermitteln; Mitarbeiter führen, coachen und überwachen; Entscheidungen treffen und die Mitglieder des Teams koordinieren.

Paulish unterscheidet zwischen der Übersichtsarchitektur, die als Referenzpunkt Stabilität vermitteln und auf einem Blatt darstellbar sein soll, und den detaillierten Architekturen der einzelnen Komponenten. Die benötigten Komponenten werden aus der Übersichtsarchitektur abgeleitet und eigenständigen Teams zugewiesen. Somit dient die Softwarearchitektur als Grundlage für die Definition der Projektorganisation. Mitglieder des Teams, das die Übersichtsarchitektur entworfen hat, eignen sich gut für die Leitung der Komponententeams, da sie ein gutes Verständnis der Architektur haben. Der Aufwand für die Entwicklung der einzelnen Komponenten wird bottom-up geschätzt, der Zeitplan für alle Komponenten hingegen top-down.

Während der Implementierung dient die Architekturbeschreibung hauptsächlich als Kommunikationsmittel. Daher muss sie nicht exakt mit der Implementierung übereinstimmen. Es findet kein Abgleich von Soll- und Ist-Architektur statt. Stattdessen werden aussagekräftige Metriken, die wichtige Systemeigenschaften messen, erhoben und überprüft.

5.2.2 Erweitertes Verständnis von architekturzentrierter Entwicklung

Das Verständnis von architekturzentrierter Entwicklung in dieser Arbeit orientiert sich an den am Arbeitsbereich Softwaretechnik im Department Informatik an der Universität Hamburg entwickelten Konzepten. Diese Konzepte wurden u. a. im Rahmen von Projekten der C1 WPS GmbH in der Praxis eingesetzt. Sie wurden zuletzt von Christiane Floyd auf der SE 2007 vorgestellt [Floyd 2007] und ausführlicher in der Dissertation von Carola Lilienthal beschrieben [Lilienthal 2008].

Dieses Verständnis von architekturzentrierter Entwicklung folgt ähnlichen Grundgedanken wie RUP/UP und das architecture-based development. Es beruht auf zwei

Säulen: STEPS und WAM. Gemeinsame Grundlage beider Säulen bildet eine softwaretechnische Betrachtungsweise, die Agilität und Flexibilität als hohe Werte begreift und auf einem iterativ-inkrementellen Prozess mit intensiver Zusammenarbeit von Entwicklern und Anwendern beruht. Fachliches und technisches Wissen werden bei der Softwareentwicklung zusammengeführt. Die empirische Basis stellen kleinere und mittlere Entwicklungsprojekte von Anwendungssoftware dar. Mit diesen Eigenschaften ist der Ansatz aus Hamburg ausgezeichnet für die Betrachtung in dieser Arbeit geeignet.

Der Methodenrahmen STEPS

STEPS (Softwareentwicklung für evolutionäre, partizipative Systementwicklung) wurde unter der Leitung von Christiane Floyd an der TU Berlin entwickelt [Floyd et al. 1989]. Softwareentwicklung ist nach STEPS hauptsächlich ein Lern- und Kommunikationsprozess aller Beteiligter, bei der die Menschen und die fachliche Anwendungswelt im Mittelpunkt stehen. Bekannt ist STEPS insbesondere durch das zyklische Projektmodell, das Softwareentwicklung als fortlaufende Abfolge von Entwicklung, Anwendung und Überarbeitung sieht (siehe Abbildung 5.3).

In STEPS verläuft die Softwareentwicklung iterativ-inkrementell. Herstellung und Einsatz wechseln sich in jedem Zyklus ab. Nach der Projektetablierung wird die erste Produktversion entworfen, spezifiziert und entwickelt. Parallel zur Entwicklung wird der Einsatz bei den Anwendern vorbereitet. Die entwickelte neue Systemversion wird von den Anwendern genutzt und von den Entwicklern gepflegt. Erkenntnisse aus dem Einsatz fließen in den nächsten Zyklus ein, der mit einer Revisionsetablierung beginnt. In jedem Zyklus werden die Dokumente fortgeschrieben und das System weiterentwickelt. Die Entwicklung endet mit dem Projektabschluss. Diese Phase ist wie die vorhergehenden Etablierungsschritte eine partizipative Aufgabe von Entwicklern und Anwendern, die im gesamten Prozess eng zusammenarbeiten.

Carola Lilienthal hat in enger Zusammenarbeit mit Christiane Floyd architekturzentrierte Softwareentwicklung in STEPS integriert [Lilienthal 2008]. Sie unterscheidet drei Stadien der architekturzentrierten Entwicklung: Entwerfen der Architektur, Erhalten der Architektur und Erneuern der Architektur. Die Unterscheidung verschiedener Stadien beruht auf der Erfahrung, dass mit Softwarearchitektur in den jeweiligen Zyklen unterschiedlich umgegangen wird.

Das erste Stadium entspricht der Neuentwicklung eines Softwaresystems. Die Architektur muss von Grund auf neu entworfen werden. Bei der Projektetablierung wird die Basis für architekturzentriertes Arbeiten gelegt. Der Nutzen aber auch die Kosten werden diskutiert. Bei der Systemgestaltung wird die Soll-Architektur konzipiert, in der Regel auf Basis eines Architekturstils. Anschließend werden das System und seine Architektur parallel entwickelt. Bei der Pflege des Systems wird auch die Ist-Architektur im Hinblick auf nötige Refactorings im nächsten Zyklus untersucht. Im zweiten Stadium,

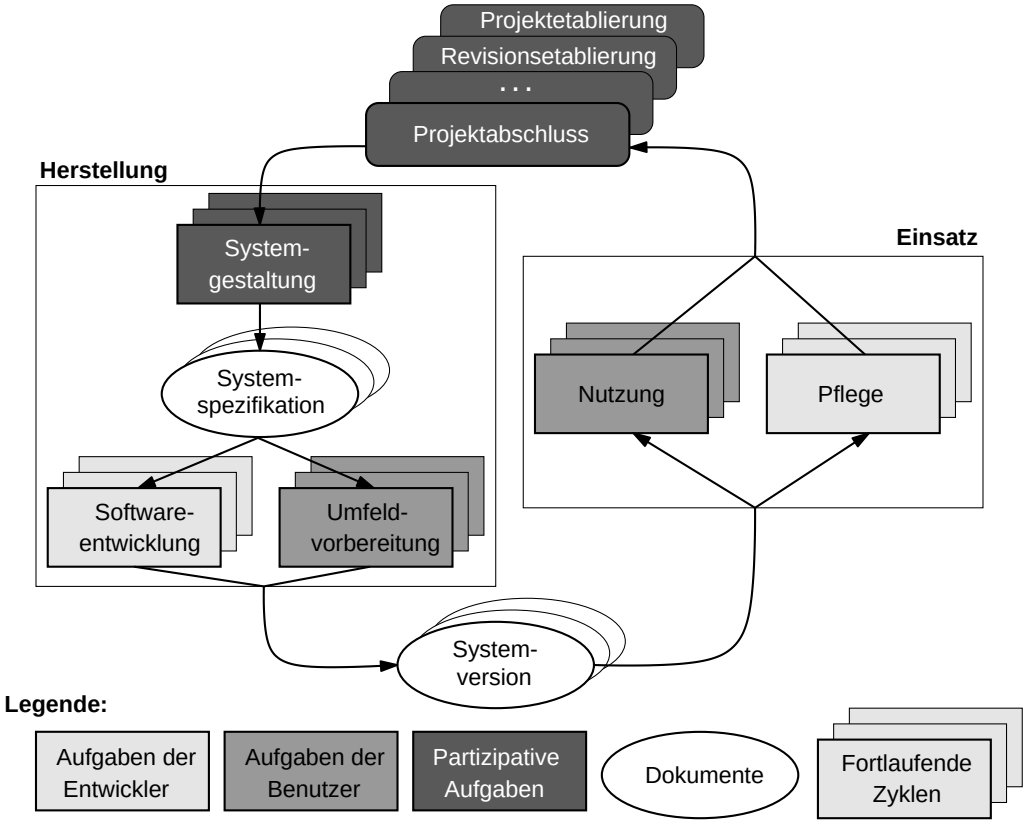


Abbildung 5.3: *Das zyklische Projektmodell von STEPS, in Anlehnung an [Floyd et al. 1989, S. 57] und [Lilienthal 2008, S. 140]*

dem Erhalten der Architektur, geht es um die Weiterentwicklung eines vorhandenen Systems. Die nötigen Erweiterungen müssen in die vorhandene Architektur integriert werden. Wenn das nicht möglich ist, muss die Architektur geändert werden. Dies kann schwierig sein, wenn die Soll-Architektur nicht oder nur unzureichend dokumentiert wurde. Die Überarbeitung und Anpassung der bestehenden Architektur finden als Refactorings durch die Entwickler statt. Nutzen und Kosten dieser Refactorings müssen im Rahmen der Revisionsetablierung mit allen Beteiligten diskutiert werden. Wenn die Architektur eines Systems nicht mehr erhalten werden kann, kommt es zum dritten Stadium, dem Erneuern der Architektur. Nach Lilienthal gibt es mehrere mögliche Ursachen, die eine Erneuerung der Architektur erforderlich machen: Die ursprüngliche Entwicklung erfolgte ohne eine gemeinsame Architekturvorstellung, die Soll-Architektur wurde nicht vermittelt, der Architekturstil war nicht ausreichend bekannt, nötige Änderungen an der Architektur wurden nicht durchgeführt oder der gewählte Architekturstil hat den Entwicklern zu viele Freiheiten gelassen [Lilienthal 2008]. In diesen Fällen muss bei der Systemgestaltung die Soll-Architektur neu konzipiert werden. Dabei sollte auch der vorhandene Architekturstil hinterfragt werden.

Christiane Floyd geht in ihrer neueren Arbeit noch einen Schritt über AZSE hinaus und prägt den Begriff der architekturzentrierten *Softwaretechnik* [Floyd 2007]. Bei dieser wird die Architektur als Mittelpunkt der gesamten Softwareentwicklung gesehen. Floyd schreibt dazu: „Das Thema Architektur in den Mittelpunkt zu stellen, erlaubt eine Sichtweise auf Software, welche die zeitliche Beschränkung auf das einzelne Projekt übersteigt, die Entwicklung und die verschiedenen Formen der Weiterentwicklung von Software in einem gemeinsamen Bezugsrahmen gleichrangig berücksichtigt und das einzelne Produkt im Zusammenhang mit anderen zu sehen gestattet.“ [Floyd 2007, S. 22]

Während früher alle softwaretechnischen Themen in den Rahmen eines Softwareprojekts eingebettet waren, ist es heutzutage die Architektur, die diese zentrale Rolle einnimmt. Dieser Wandel dauert noch an. Floyd plädiert für einen achtsamen Umgang mit Software und fordert, eine nachhaltige Softwareentwicklung zu betreiben, Qualität zu gewährleisten, änderungsfreundliche Systeme zu erstellen und das System über seine gesamte Lebensdauer zu betrachten. Zentrale Voraussetzung dafür ist die beständige Pflege der Softwarearchitektur.

Der Werkzeug & Material-Ansatz

Der Werkzeug & Material-Ansatz (WAM-Ansatz) stellt die zweite Säule des in dieser Arbeit vertretenen Verständnisses von architekturzentrierter Softwareentwicklung dar. Der WAM-Ansatz ist eine Softwareentwurfs- und Konstruktionstechnik. Er wurde 1991 in der Gesellschaft für Mathematik und Datenverarbeitung (GMD) durch Arbeiten von Heinz Züllighoven, Reinhard Budde und Karl-Heinz Sylla begründet. Er verfolgt das Ziel, anwendungsorientierte Software mit hoher Gebrauchsqualität zu entwickeln [Züllighoven 1998], [Züllighoven 2004]. Eine präzise und häufige Kommunikation zwischen

allen Beteiligten steht im Mittelpunkt des Entwicklungsprozesses. Daher werden bei Entwicklungsprojekten nach dem WAM-Ansatz in der Regel agile Methoden eingesetzt, vor allem Extreme Programming [Lippert et al. 2003a].

Der WAM-Ansatz basiert auf einer soliden softwaretechnischen Grundlage. Diese Grundlage besteht u. a. aus einem Objekt-Metamodell, Leitbildern und Entwurfsmetaphern, Entwurfsmustern und Rahmenwerken sowie Modellen und Dokumenten für den Entwicklungsprozess. Dadurch eignet sich der WAM-Ansatz sowohl für Forschungsarbeiten als auch für Entwicklungsprojekte in der Praxis.

Im WAM-Ansatz spielt Softwarearchitektur im gesamten Entwicklungsprozess eine wichtige Rolle. Architekturzentriertes Arbeiten wird im WAM-Ansatz durch eine Reihe von Konzepten und Methoden unterstützt:

Entwurfsmetaphern: Durch die ausgeprägte Verwendung von Metaphern unterscheidet sich der WAM-Ansatz von anderen Ansätzen zur Softwaremodellierung und -entwicklung. Denn obwohl Metaphern zur Veranschaulichung bei der Softwareentwicklung empfohlen werden, haben sie sich nicht nennenswert verbreitet. So wurde „Metapher“ in [Ramachandran u. Shukla 2002] als die unbeliebteste Technik des Extreme Programming identifiziert.

Eine zentrale Idee des WAM-Ansatzes ist die Herstellung einer Strukturähnlichkeit zwischen Gegenständen und Begriffen des realen Anwendungsbereiches und ihrer Realisierung im Softwaresystem. Die Entwickler besitzen dadurch eine Entscheidungsgrundlage für den Systementwurf und Anwender können sich besser im System orientieren. Der Entwurf wird durch Metaphern unterstützt. Entwurfsmetaphern machen die Handhabung und Funktionalität für alle Beteiligten verständlicher. Die wichtigsten Entwurfsmetaphern im WAM-Ansatz sind Werkzeug, Material, Automat und Behälter. Sie sind so einfach gewählt, dass Kunden und Entwickler eine intuitive Vorstellung von ihrer Bedeutung besitzen. Auf ähnlicher Stufe wie die Entwurfsmetaphern stehen eher an der Konstruktion orientierte Konzepte von WAM wie Fachwerte und Services. Die Implementierung von beiden Elementarten wird durch Konstruktionsanleitungen und Entwurfsmuster unterstützt [Züllighoven 1998], [Lippert et al. 2003b].

Durch die gesammelten und weitergegebenen Erfahrungen in der Implementierung von Entwurfsmetaphern entstehen „Mikro-Architekturen“, welche die Entwickler auf der untersten Ebene der Umsetzung leiten. Abbildung 5.4 zeigt die Beziehungen zwischen wichtigen WAM-Elementen. Die Pfeile geben an, welche Elemente andere Elemente benutzen. So dürfen Werkzeuge auf Services zugreifen, die bestimmte fachliche Funktionen zur Verfügung stellen. Die umgekehrte Beziehung ist nicht erlaubt. Behälter enthalten Materialien und bieten fachlich motivierte Operationen zum Zugriff auf diese an.

Modellarchitektur: Um Architekten und Entwickler auch auf einer höheren Ebene bei der Realisierung eines Systems zu unterstützen, wird in WAM das Kon-

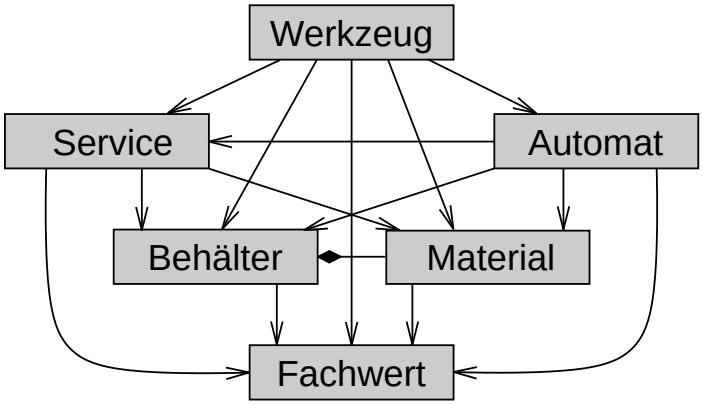


Abbildung 5.4: *Die Beziehungen der WAM-Elemente*

zept der Modellarchitektur verwendet. Eine Modellarchitektur beschreibt die grundlegenden Elemente einer Softwarearchitektur mit deren Verknüpfungen und Regeln [Züllighoven 1998]. Sie stellt damit eine Referenzarchitektur im Sinne von Abschnitt 5.1.2 dar.

So wie Gegenstände und Begriffe des Anwendungsbereichs Vorbilder für einfache Systemeinheiten sind, eignen sich Organisationsstrukturen als Grundlage für größere Einheiten. In der WAM-Modellarchitektur sind dies der Gegenstandsbereich, der Produktbereich und der Einsatzkontext. Der Gegenstandsbereich umfasst grundlegende fachliche Konzepte des Unternehmens. Der Produktbereich enthält Objekte von Produkt- oder Dienstleistungsarten des Unternehmens. Der konkrete Arbeitszusammenhang der Anwendungssysteme wird in Einsatzkontexten abgebildet. Ein Einsatzkontext kann Produkte und Dienstleistungen aus unterschiedlichen Produktbereichen verwenden.

Zusätzlich enthält die WAM-Modellarchitektur technische Schichten, die als Grundlage zur Realisierung der fachlichen Schichten dienen. Dies sind die Systembasisschicht, die Technologieschicht und die Handhabungs- und Präsentationsschicht. Abbildung 5.5 zeigt den Zusammenhang der Schichten.

Rahmenwerk: Die Implementierung einer Modellarchitektur kann durch ein geeignetes Anwendungsrahmenwerk bedeutend erleichtert werden. Nach [Züllighoven 1998, S. 119] ist ein Anwendungsrahmenwerk „für die Entwicklung von Anwendungssoftware in einem fachlich eingegrenzten Anwendungsbereich gedacht. Es gibt die softwaretechnische Architektur und die relevanten fachlichen Abstraktionen in Form einer generischen Lösung vor.“ Die Entwicklung von Rahmenwerken ist eine softwaretechnisch anspruchsvolle Aufgabe, die viel Erfahrung in der jeweiligen Anwendungsdomäne erfordert. Für Anwendungen nach dem WAM-

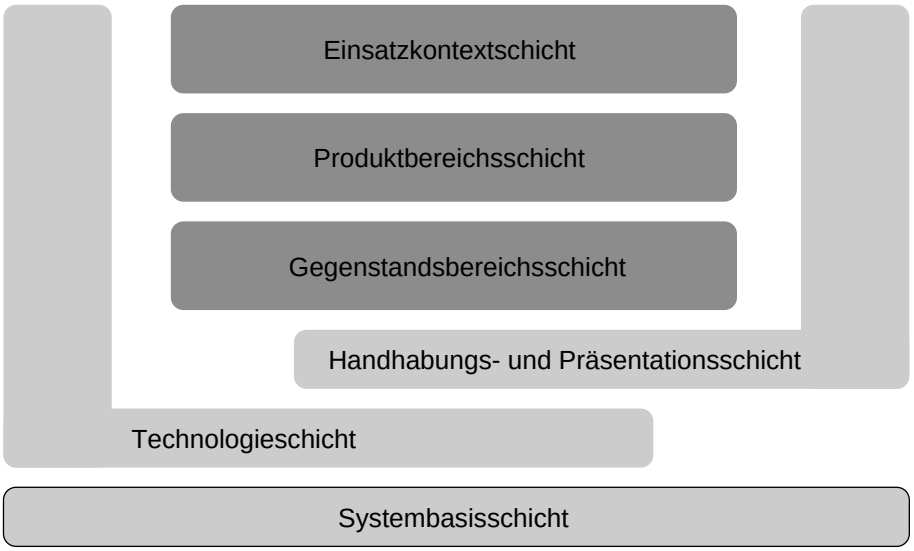


Abbildung 5.5: *Die Schichten der WAM-Modellarchitektur, nach [Züllighoven 1998, S. 361]*

Ansatz steht das JWAM-Rahmenwerk[2] zur Verfügung. Es implementiert die WAM-Modellarchitektur in der Programmiersprache Java.

Architekturregeln: Die Notwendigkeit von Architekturregeln wurde schon auf Seite 124 begründet. Im WAM-Ansatz werden Regeln auf verschiedenen Ebenen verwendet. Auf der obersten Ebene kann die korrekte Schichtung der WAM-Modellarchitektur überprüft werden. Die Regeln der Soll-Architektur einer Anwendung werden darauf aufbauend durch den Systemarchitekten vorgegeben. Auf der untersten Ebene werden Regeln für die korrekte Verwendung der Entwurfsmetaphern geprüft. So dürfen beispielsweise Werkzeuge auf Materialien zugreifen, umgekehrt aber nicht. Für viele dieser Regeln werden Softwarewerkzeuge zur automatisierten Überprüfung bereitgestellt.

Besonderheiten gegenüber den anderen Ansätzen

Mit der Kombination von STEPS und WAM ist ein Ansatz entstanden, der sowohl Unterstützung für den Umgang mit Architektur im Entwicklungsprozess auf hoher Ebene als auch für architekturzentriertes Arbeiten in Detail bietet.

Im Folgenden werden zur Abgrenzung gegenüber den anderen in dieser Arbeit vorgestellten Ansätzen (RUP/UP, Bass/Kazman, Paulish) Besonderheiten aufgegriffen und Unterschiede herausgestellt. Dabei fließen auch Erfahrungen aus der Projektpraxis mit STEPS und WAM ein.

[2]Das JWAM-Rahmenwerk ist als Open-Source-Software verfügbar, siehe `http://sourceforge.net/projects/jwamtoolconstr/`.

Bei der Kombination STEPS/WAM steht ein iteratives und evolutionäres Vorgehen im Vordergrund. Dieses wird auch auf die Arbeit mit Architektur bezogen. Die Architektur wird nicht zu Beginn des Projekts festgelegt, sondern unterliegt einer Evolution, bei der die Architekturdokumentation fortgeschrieben wird. Erfahrungen aus dem Einsatz in der Praxis und dem Feedback der Anwender können sich auch auf die Architektur auswirken. In vielen Projekten wird die Architektur nicht von Grund auf entworfen, sondern baut auf der WAM-Modellarchitektur als Referenz auf. Während andere Ansätze wenig detaillierte Vorgaben zur Architektur machen, liegen der Kombination STEPS/WAM konkrete Metaphern, Architekturvorstellungen und -regeln zugrunde.

Mit den Ansätzen RUP/UP, Bass/Kazman und Paulish werden meist größere Projekte abgewickelt. Das System lässt sich in klar voneinander abgrenzbare, von unterschiedlichen Teams in eigenständiger Verantwortung umzusetzende Subsysteme unterteilen. Die Architektur legt dann die Beziehungen dieser Subsysteme fest. Typisch für diese Art der Architekturbeschreibung scheint eine recht einfache Schichtenarchitektur mit wenigen Subsystemen zu sein. So wurden z. B. beim in [Sangwan et al. 2006] beschriebenen Global Studio Project von Siemens, das vom SEI geleitet wurde, nur sieben Subsysteme definiert und in der Architektur behandelt. In einer typischen WAM-Architektur finden sich deutlich mehr, dafür aber kleinere Subsysteme. Deren Beziehungen werden detailliert festgelegt und sind mithilfe von Architekturregeln überprüfbar.

Bei Projekten nach WAM und STEPS spielen Agilität und ein leichtgewichtiges Vorgehen eine wichtige Rolle. Die durchgeführten Projekte sind in der Regel von kleiner bis mittlerer Größe und haben meist KMU als Auftraggeber (vgl. die Größeneinstufung in Abschnitt 3.4.3).

5.2.3 Vorhandene wissenschaftliche Erkenntnisse

Obwohl architekturzentrierte Entwicklung bei der Softwareentwicklung immer weitere Verbreitung findet, gibt es nur wenige Konzepte und praktische Erfahrungen mit architekturzentrierter Softwareentwicklung in verteilten Entwicklungsprojekten. Diese sind aus Sicht des Autors dieser Arbeit nötig, da verteilte Entwicklung besondere softwaretechnische Herausforderungen mit sich bringt und unklar ist, inwieweit Techniken dort anwendbar sind, die in nicht-verteilten Projekten funktionieren.

Viele Veröffentlichungen im Bereich der Softwarearchitektur machen keine oder nur ungenaue Aussagen zu globaler Softwareentwicklung und Offshoring. So geht beispielsweise Dan Paulish in seinem oben vorgestellten Buch zu architekturzentriertem Management nur wenig auf verteilte Entwicklung ein (vgl. [Paulish 2001]). Er betont die Wichtigkeit von Kommunikation und Koordination bei verteilter Entwicklung und erläutert die Möglichkeit verteilter Quelltextanalysen per Videokonferenz. Obwohl er feststellt, dass einige Architekturstile besser als andere für verteilte Entwicklung geeignet sind (“Certain architectures will be more suitable to global development than others, and it is important for the architecture design team to realize this.” [Paulish

2001, S. 100]), gibt er nur zwei Beispiele für geeignete Architekturstile: lose gekoppelte Subsysteme oder eine Aufteilung nach Schichten. Weitere konkrete Hinweise für die Gestaltung und den Umgang mit Architektur in verteilten Projekten fehlen. Auch in STEPS und WAM finden global verteilte Softwareentwicklung und Offshoring noch keine Berücksichtigung.

Die folgende Literaturrecherche soll den Stand der Forschung im Bereich architekturzentrierte Entwicklung bei verteilter Entwicklung aufzeigen. Die vorhandene Literatur lässt sich in vier Kategorien einteilen. Publikationen der ersten Kategorie geben einen Überblick über die Einsatzmöglichkeiten von Softwarearchitektur bei verteilter Softwareentwicklung. Die der zweiten Kategorie untersuchen die positiven Auswirkungen auf die Kommunikation. Publikationen der dritten Kategorie beschreiben, wie mit Softwarearchitektur eine geeignete Organisationsstruktur für verteilte Projekte gefunden und die Arbeit auf die Teams aufgeteilt werden kann. In der letzten Kategorie geht es um Möglichkeiten zur Unterstützung der verteilten Entwicklung.

1. Nutzung von Softwarearchitektur bei verteilter Softwareentwicklung

Abbas, Dart und Kazmierczak haben schon 1998 die wichtige Rolle erkannt, welche die Softwarearchitektur beim Outsourcing und verteilter Entwicklung spielen sollte [Abbas et al. 1998a]. Mittels Softwarearchitektur können technische Anforderungen (hauptsächlich die Eigenschaften des zu entwickelnden Systems) mit Anforderungen des Managements (hauptsächlich bezüglich der Organisation der verteilten Entwicklung) in Einklang gebracht werden. Bedingt durch das Offshoring muss die Architektur eine Reihe von Gesichtspunkten berücksichtigen. Sie muss die Anforderungen verschiedener Akteure widerspiegeln und für alle verständlich, klar und konsistent sein. Sie muss berücksichtigen, dass Teams mit potenziell unterschiedlichen Entwicklungsparadigmen beteiligt sind. Sie muss so gestaltet sein, dass eine klare Aufgabenteilung zwischen Teams und parallele Entwicklungsarbeiten möglich sind. Sie muss die Kopplung zwischen Subsystemen und damit den nötigen Koordinationsaufwand minimieren. Und sie sollte so flexibel sein, dass sie leicht an Änderungen anpassbar ist.

De Faria und Adler haben später den Begriff der architekturzentrierten globalen Softwareprozesse geprägt, basierend auf ihren industriellen Erfahrungen [Rocha de Faria u. Adler 2006]. Sie sehen in einer stärkeren Orientierung an Architektur hauptsächlich vier Vorteile: 1. Mit einer gut entworfenen Architektur wird die parallele Implementierung durch geografisch verteilte Entwicklungsteams produktiver und flexibler. 2. Informationen über die Architektur können in einem zentralen Konfigurationsmanagementprozess verwaltet werden und stehen allen Gruppen als Leitlinie zur Verfügung. 3. Architektur hilft auch bei einer pragmatischen und nachvollziehbaren Qualitätssicherung. 4. Die Kommunikation zwischen den verteilten Teams wird mit einer durch die Architektur definierten gemeinsamen Sprache verbessert. Aufgrund dieser positiven Auswirkungen kann die Architektur alle Entwicklungsaktivitäten unterstützen und die Koordination erleichtern.

Ali Babar betont, dass die Softwarearchitektur auch bei global verteilter Entwicklung regelmäßig nach formalen Kriterien geprüft werden sollte. Dafür wurde die Methode „Software Architecture Evaluation Process“ so angepasst, dass sie auch mittels Groupware-Systemen durchgeführt werden kann [Ali Babar 2009].

2. Verbesserung der Kommunikation

Einige Arbeiten beschäftigen sich im Detail mit den positiven Auswirkungen einer gemeinsamen Architektur auf die Kommunikation zwischen verteilten Entwicklerteams.

Ovaska, Rossi and Marttiin haben die Möglichkeiten untersucht, die Architektur für die Koordination bei der Softwareentwicklung bietet [Ovaska et al. 2003]. Sie stützen sich auf die häufig zitierte Untersuchung von Kraut und Streeter, nach der das Ziel bei der Softwareentwicklung nicht sein sollte, persönliche Kommunikation zu unterbinden. Stattdessen sollte die Kommunikation effizienter und erfolgreicher werden [Kraut u. Streeter 1995]. In diesem Sinne lautet die Kernaussage von Ovaska et al., dass Architekten und Entwickler ein gemeinsames Verständnis der Architektur entwickeln sollten. Dadurch können sie die verteilte Entwicklung besser koordinieren. Eine wichtige Rolle fällt dem Chef-Architekten zu, der die Integrität der Architektur gewährleisten muss. Dazu muss er die Architektur und einzuhaltende Regeln allen Beteiligten vermitteln können.

In drei Studien in einem großen Telekommunikationsunternehmen wurden weitere Einflüsse auf die Koordination von verteilten Softwareteams untersucht [Espinosa et al. 2002]. Die Forscher kommen zu dem Ergebnis, dass der Erfolg maßgeblich davon abhängt, wie es den Teammitgliedern gelingt, eine Vertrautheit mit der Arbeit (engl. work familiarity) und ein gemeinsames mentales Modell (engl. shared mental model) zu entwickeln. Die Vertrautheit mit der Arbeit umfasst Wissen über Werkzeuge, Arbeitsgegenstände, Aufgaben und Mitarbeiter. Dieses Wissen ist Voraussetzung für eine erfolgreiche Koordination der Teams. Weitere Ergebnisse sind, dass die Entwicklungszeit bei Projekten mit verteilten Teams gegenüber nicht-verteilten Projekten ansteigt und dass die Vertrautheit mit der Arbeit bei verteilten Teams eine stärkere positive Auswirkung hat.

Matthew Bass hat die Überlegungen zu gemeinsamen mentalen Modellen aufgegriffen und auf das Projektmanagement übertragen [Bass 2006]. Als eine der Hauptstrategien zur Verbesserung der Kommunikation sieht er die Entkopplung der Arbeit anhand der von der Architektur vorgegebenen Schnittstellen. Dies deckt sich mit Erfahrungen aus der Arbeit mit Architektur in anderen komplexen technischen Bereichen, beispielsweise mit der Entwicklung von Flugzeugmotoren, mit denen sich Paasivaara auseinandergesetzt hat ([Paasivaara 2005], siehe auch [Sosa 2000]). Paasivaara kommt zu dem Schluss, dass sich die Kommunikation zwischen verteilten Teams nach den Schnittstellen der von ihnen entworfenen Subsysteme richten sollte.

Diese Ergebnisse führen direkt zur nächsten Kategorie, in die Publikationen fallen, die den Nutzen von Architektur für die Organisation der Softwareentwicklung und die Aufteilung der Aufgaben untersuchen.

3. Organisationsstruktur und Aufteilung der Arbeit

Schon zu Beginn dieses Kapitels wurde dargelegt, dass in den sechziger und siebziger Jahren durch Dijkstra, Parnas, Myers et al. wichtige Grundlagen zur inneren Gliederung eines Softwaresystems gelegt wurden. Das damalige Hauptanliegen war es, die sogenannte Softwarekrise durch Verringerung der Komplexität der Software zu überwinden. Insbesondere Parnas hatte schon damals erkannt, dass architektonische Entscheidungen zur Strukturierung eines Systems wichtige Auswirkungen auf die Arbeitsteilung besitzen [Parnas 1972].

Abbas et al. haben die Arbeitsteilung beim Outsourcing der Softwareentwicklung untersucht und darauf hingewiesen, wie wichtig es ist, Abhängigkeiten zwischen Subsystemen genau zu planen, um auf Änderungen reagieren zu können [Abbas et al. 1998b, S. 636]: "The organisation of the architectural components of the system and the interfaces between them need to be carefully considered to minimise dependencies and side-effects of change."

In Abschnitt 3.4.2 wurde Conway's Law zum Zusammenhang der Strukturen einer Organisation und des von ihr entwickelten Systementwurfs zitiert. Darauf aufbauend haben Herbsleb und Grinter die Aufteilung eines Systems in Subsysteme und die Auswirkungen auf die Arbeitsteilung in einer Reihe von Studien untersucht. Sie kommen zu dem Schluss, dass die Berücksichtigung von Architektur helfen kann: Zum einen kann mit Architektur die Entwicklungsarbeit in kleinere Aufgaben heruntergebrochen werden, die weitgehend unabhängig voneinander umgesetzt werden können. Zum anderen lassen sich durch eine geeignete Architektur Abhängigkeiten zwischen unterschiedlichen Teams und damit der Aufwand für Kommunikation und Koordination verringern [Herbsleb u. Grinter 1999a].

Herbsleb und Grinter haben zusammen mit Perry weitere Möglichkeiten zur Aufteilung der Entwicklung untersucht [Grinter et al. 1999]. Sie konzentrieren sich auf Forschung und Entwicklung, doch sollten ihre Ergebnisse auch auf andere Bereiche übertragbar sein. Demnach lassen sich drei weitere Modelle neben der Produktarchitektur unterscheiden: Aufteilung nach Sachgebieten oder nach Prozessschritten sowie kundenspezifische Anpassungen. Vgl. auch die schon auf Seite 25 diskutierten Unterteilungsvarianten beim Outsourcing nach Abbas et al.

Bei der Aufteilung nach der durch die Architektur bestimmten Produktstruktur müssen die Schnittstellen klar definiert sein. Daraus ergibt sich der Vorteil, dass an den verschiedenen Standorten unterschiedliche Prozesse und Entwicklungswerkzeuge eingesetzt werden können. Nachteilig ist, dass verschiedene Komponenten richtig zusammenspielen

müssen, um Features der Anwendung zu realisieren. Die Integration des Gesamtprodukts kann schwierig sein.

Eine Aufteilung nach Sachgebieten bietet Vorteile, wenn alle Experten mit Fachkenntnissen in einem bestimmten Bereich an einem Standort versammelt sind und sie so gut zusammenarbeiten können. Ein Nachteil ist, dass dadurch bei den meisten Projekten mehrere Standorte zusammenarbeiten müssen, mit den in dieser Arbeit schon diskutierten Schwierigkeiten. Bei der Aufteilung nach Prozessschritten werden Schritte wie Entwicklung und Test voneinander abgegrenzt und an verschiedenen Standorten bearbeitet. Dadurch lassen sich knappe Ressourcen, wie z. B. Testlabore, gut auslasten und Arbeiten, die regionales Wissen erfordern, an geeignete Standorte verlagern. Problematisch kann die genaue Spezifikation und Einhaltung der Übergabepunkte sein. Verzögerungen an einem Standort führen zu Verzögerungen im gesamten Prozess, die schlecht ausgleichbar sind. Das Modell der kundenspezifischen Anpassungen sieht vor, dass an einem Standort das Kernprodukt entwickelt wird. An weiteren Standorten wird es an die Bedürfnisse der regionalen Kunden angepasst. Dieses Modell kombiniert die Unterteilung nach der Architektur mit der Unterteilung nach Prozessschritten. Wenn das Modell korrekt umgesetzt wird, führt es zu einem guten internen Entwurf der Anwendung. Die einzelnen Standorte sind jeweils nah an ihren Kunden und deren Anforderungen. Doch es gibt auch eine Reihe von Schwierigkeiten. Die Prozesse sind schwierig zu koordinieren, die Kompatibilität der Infrastruktur muss sichergestellt werden, es muss genaues Wissen über die Kernanwendung an die einzelnen Standorte weitergegeben werden und die Beteiligten müssen sich gegenseitig vertrauen. Herbsleb et al. kommen zu dem Schluss, dass kein Modell alle Koordinationsprobleme lösen kann. In großen Projekten kann mehr als ein Modell zur Arbeitsteilung eingesetzt werden.

Carmel et al. nennen ein weiteres Modell: integrierte Entwicklung nach dem „Follow the Sun“-Prinzip [Carmel et al. 2009]. Dabei findet täglich eine Übergabe zwischen den Teams statt, die in sehr weit entfernten Zeitzonen arbeiten (vgl. die Vorstellung auf Seite 31). Dem Vorteil der kontinuierlichen Arbeit an einem System steht der schwerwiegende Nachteil entgegen, dass tägliche Übergaben zu bewältigen sind. Eine gemeinsame Architekturvorstellung ist sehr wichtig, um so eng gekoppelt arbeiten zu können.

4. Unterstützung bei der Entwicklung

Mehrere Forschungsarbeiten beschäftigen sich mit der Frage, wie verteilte Entwicklung unterstützt werden kann. Dabei geht es zum einen um geeignete Werkzeuge, zum anderen um optimierte Prozesse.

Eine gute Entwicklungsumgebung ist gerade für die Arbeit in verteilten Teams wichtig. In letzter Zeit stand das von IBM Rational unter der Regie von John Wiegand und Erich Gamma entwickelte, auf der populären Entwicklungsumgebung Eclipse aufbauende „Jazz“ im Mittelpunkt der Aufmerksamkeit des Fachpublikums. Jazz enthält

mehrere Komponenten, welche die Zusammenarbeit verteilter Teams unterstützen sollen. Die Verbesserung, die in realen Projekten durch den Einsatz von Jazz und anderen Werkzeugen erreicht werden kann, hängt maßgeblich davon ab, inwieweit der praktizierte Entwicklungsstil mit den Werkzeugen abgebildet werden kann und wie groß die Bereitschaft zur Anpassung an die bei Jazz hinterlegten Referenzprozesse ist [Frost 2007]. Jazz ist ein kommerzielles Werkzeug. Im Umfeld haben sich jedoch eine Reihe von Forschungsprojekten etabliert, die mit einer für akademische Zwecke freigegebenen Version (Rational Team Concert) arbeiten. Eine Zusammenstellung dieser Projekte findet sich unter `http://jazz.net/community/academic/relatedResearchProjects.jsp`. Viele Projekte befinden sich allerdings noch in einer frühen Forschungsphase.

De Souza et al. haben dagegen schon Ergebnisse ihrer Arbeit vorgelegt [de Souza et al. 2007]. Ihre empirische Forschung hat zwei wesentliche Bedürfnisse aufgedeckt, die sich für Entwickler aus der verteilten Arbeit ergeben. Zum einen ist es für die Entwickler wichtig, innere Abhängigkeiten der Anwendung und Auswirkungen ihrer Änderungen auf die Arbeit anderer Entwickler zu verstehen. Zum anderen sollten Entwurfsentscheidungen nachvollziehbar sein und sich Kundenanforderungen oder internen Festlegungen zuordnen lassen. Zur Unterstützung der Entwickler wurden zwei Werkzeuge entwickelt. „Ariadne“ analysiert automatisiert Versionsinformationen aus zentralen Archiven, um Abhängigkeiten zwischen Komponenten festzustellen. Basierend auf einer Zuordnung von Entwicklern zu den von ihnen bearbeiteten Komponenten werden dann mögliche Kommunikationsbeziehungen zwischen Entwicklern aufgezeigt. „TraVis“ verarbeitet Daten aus Versionsverwaltungssystemen, Entwicklerwikis und -foren sowie manuell eingegebenen Abhängigkeitsbeziehungen und erstellt daraus eine Übersichtsgrafik. Diese Grafik kann genutzt werden, um Grundlagen von Entscheidungen und Diskussionen zu bestimmten Komponenten nachvollziehen zu können.

Einen ähnlichen Ansatz wie Ariadne verfolgen auch Wagstrom und Herbsleb [Wagstrom u. Herbsleb 2006]. Neben der automatisierten Erstellung einer Grafik mit Kommunikationsbeziehungen unter Entwicklern soll durch eine Analyse der Architekturelemente auch der zukünftige Kommunikationsbedarf vorhergesagt werden. Mit dieser Information kann z. B. die Verteilung der Aufgaben auf die Teams optimiert werden. Bass et al. weisen darauf hin, dass architektonische Abhängigkeiten zwischen verteilten Teams nicht nur durch Schnittstellen und Aufrufbeziehungen von Komponenten entstehen können [Bass et al. 2007]. Sie haben Projekte untersucht, bei denen auch Leistungs- und Speicherplatzanforderungen sowie Stabilitätsprobleme durch komplexe Synchronisationsmechanismen oder inkompatible Annahmen über den Zustand von Komponenten zu teilweise schwierig zu lösenden Koordinationsproblemen zwischen verteilten Teams geführt haben. Daher sollten verteilte Teams auch diese und weitere Abhängigkeiten im Auge behalten.

Curtis, Krasner und Iscoe haben sich mit der Entwicklung von großen Systemen beschäftigt [Curtis et al. 1988]. Sie betonen die wichtige Rolle, die Softwarearchitekten dabei als Vermittler spielen. Corry, Hansen und Svensson haben das Modell der „herumreisenden Architekten“ (engl. traveling architects) entwickelt, damit auch zwischen mehreren

verteilten Teams vermittelt werden kann. Herumreisende Architekten sind "a group of architects responsible for maintaining software architectural assets in a distributed development project by visiting development sites in order to design, evaluate, and enforce a software architecture in active collaboration with developers and possibly end users." [Corry et al. 2006] Die Autoren sehen darin Vorteile in der Wissensvermittlung, einer Intensivierung der Kommunikation und einer besseren Dokumentation der Softwareentwürfe. Außerdem sammeln die Architekten Erfahrungen, wie die Entwickler mit der Architektur zurechtkommen.

Ein weiteres Thema in der Literatur ist die Frage, wie verbindlich die Architektur vorgegeben und überprüft werden soll. Herbsleb und Grinter warnen davor, zu viele Details auszuspezifizieren und damit die innovativen Leistungen der Teams zu unterbinden [Herbsleb u. Grinter 1999b]. Vergleiche die Diskussion zur engen oder flexiblen Steuerung der Zusammenarbeit in Abschnitt 3.4.5. Wenn unabhängiges, situatives Handeln bei globaler Softwareentwicklung stark betont wird, sollten zwei mögliche Folgen bedacht werden. Zum einen müssen Abhängigkeiten zu Aufgaben von entfernt arbeitenden Kollegen berücksichtig werden. Solche Interdependenzen sind in verteilten Projekten in der Regel weniger transparent als bei der nicht-verteilten Arbeitsweise. Zum anderen müssen Änderungen am vorgegebenen Plan an die anderen Teams so kommuniziert werden, dass alle sich daran anpassen können. In nicht-verteilten Projekten werden viele dieser Koordinationsprobleme ohne großen Aufwand durch informelle Kommunikation gelöst. Informelle Kommunikation wird durch eine verteilte Arbeitsweise allerdings erheblich erschwert, wie bereits in dieser Arbeit aufgezeigt wurde. Ein von zentraler Stelle vorgegebener architektonischer Rahmen mit einem Integrationsplan und einer Beschreibung der Schnittstellen der Komponenten kann helfen. Er sollte jedoch ausreichend Raum für eigenständige Implementierungen lassen.

Eine Möglichkeit, diesen Rahmen zu realisieren, besteht in der Vorgabe von Architekturregeln, wie Clerc et al. festgestellt haben [Clerc et al. 2007]. Nach einer empirischen Untersuchung in zwei Organisationen kommen sie zu dem Schluss, dass Architekturregeln bei verteilter Softwareentwicklung gut eingesetzt werden können, dass aber noch genauere Forschung nötig ist, um zu erklären, welche Regeln sinnvoll sind und wie sie im Entwicklungsprozess eingesetzt werden können. Architekturregeln alleine reichen nach ihrer Untersuchung nicht aus. So lassen sich z. B. kulturelle Unterschiede und weitere Schwierigkeiten bei der Zusammenarbeit der Teams durch sie nicht adressieren.

Ebenfalls von Clerc stammen erste Überlegungen zum Management von Architekturwissen für die globale Softwareentwicklung [Clerc et al. 2009]. Basierend auf einer Literaturstudie wurden sieben förderliche Praktiken der Anforderungsermittlung identifiziert und auf das Management von Architekturwissen bezogen: häufige Interaktionen zwischen den Arbeitsorten, Austausch von Mitarbeitern zwischen den Arbeitsorten, gemeinsames Kickoff-Treffen, priorisierte Behandlung dringender Anfragen, gemeinsame Entwicklung der Übersichtsarchitektur an einem Ort, übersichtliche Organisationsstruktur mit klaren Ansprechpartnern sowie Nutzung eines gemeinsamen Archivs für Architekturartefakte. Fast alle dieser Praktiken sind nicht nur für den Umgang

mit Architektur nützlich. Viele wurden in dieser Arbeit schon bei der Analyse von Erfahrungsberichten in Abschnitt 3.3 und an anderen Stellen beschrieben.

5.2.4 Zusammenfassung des Forschungsstandes

Die vorgestellten Ergebnisse zeigen, dass es hilfreich für die verteilte Entwicklung sein kann, sich mit Softwarearchitektur zu beschäftigen. Die aktuelle Forschung in der Literatur konzentriert sich auf bestimmte Themen. Insbesondere stehen oft die aus einer Architektur abgeleitete Organisationsstruktur und die Aufteilung der Arbeit auf verschiedene Teams anhand der Architektur und die sich daraus ergebenden Kommunikationsbeziehungen im Mittelpunkt.

Eine große Forschungslücke besteht noch bezüglich des konkreten Umgangs mit einer vorgegebenen Softwarearchitektur beim verteilten Arbeiten, dem architekturzentrierten Entwickeln. Bietet dieses ähnliche Vorteile wie in nicht-verteilten Projekten? Wie kann z. B. ein Softwarearchitekt entfernten Entwicklern seine Vision vermitteln und die Umsetzung steuern? Es gibt bisher keine Untersuchung der dafür einsetzbaren agilen Techniken und Artefakte und zu ihrem Zusammenspiel im Sinne eines Gesamtkonzepts. Die Möglichkeiten von agiler Softwareentwicklung werden ebenfalls nur in sehr wenigen Arbeiten betrachtet.

Auffällig ist auch, dass viele der bisherigen Forschungsergebnisse auf empirischen Erfahrungen aus Projekten mit Studierenden beruhen und nicht auf professioneller Softwareentwicklung. Viele der veröffentlichten Methoden und Werkzeuge sind in der Praxis noch nicht ausreichend evaluiert worden.

5.3 Das Empirieprojekt Alpha

Im Empirieprojekt Alpha wurde architekturzentriertes Arbeiten in einem global verteilten Kontext erprobt.

Gegenstand des Projekts Alpha war die Entwicklung eines Prototyps für ein deutsches, mittelständisches Versandhandelsunternehmen. Dieses wollte seine Anwendungslandschaft erneuern, beginnend mit dem Bestellerfassungs- und Kundeninformationssystem.

5.3.1 Einsatz von Action Research

Als Forschungsmethode wurde in der Feldstudie Action Research eingesetzt (siehe Abschnitt 1.3.2 für eine Erläuterung der Methode). Beim Action Research bewegt sich der Forscher immer im Spannungsfeld zwischen eigenen Handlungen und reflektierender

Forschung. Dabei besteht das Risiko, dass die Forschungsergebnisse durch das Handeln des Forschers verfälscht werden [Susman u. Evered 1978].

Der Forscher kann im Projekt entweder als (passiver) Beobachter, als reguläres Teammitglied oder in leitender Funktion auftreten. Nach Ottosson et al. bietet die Rolle des Projektleiters dem Forscher gerade in Projekten, in denen Produkte entwickelt werden, viele Vorteile. Er gelangt am besten an Informationen und kann die Ergebnisse seiner Untersuchungen direkt wieder in das Projekt einbringen [Ottosson et al. 2006].

In diesem Sinne nahm der Autor dieser Arbeit im hier beschriebenen Empirieprojekt die Rolle des Onshore-Softwarearchitekten und -Projektleiters wahr. Er konnte die eingesetzten Konzepte und agilen Techniken mitbestimmen und die Auswirkungen von Maßnahmen direkt überprüfen. Mit fast allen Beteiligten war ein intensiver Austausch möglich. Die wichtigsten Dokumente waren einsehbar, u. a. Abnahmeprotokolle, Quelltexte, Ergebnisse von Reviews und ein umfangreicher E-Mail-Verkehr. Während des Projekts führte der Autor ein Forschungstagebuch.

Trotz eines teils hektischen Projektverlaufs konnte Zeit für die bei Action Research vorgesehenen Reflexions- und Auswertungsstufen gefunden werden. Dabei wurden die Erkenntnisse und Empfehlungen der letzten Kapitel berücksichtigt. Die Ergebnisse der Forschungsaktivitäten flossen durch einen Prozess der schrittweisen Verbesserung direkt wieder in das Projekt ein und konnten dort evaluiert werden.

Für die anderen Projektbeteiligten war die Doppelrolle des Autors während des Projekts nicht offensichtlich. Das Forschungstagebuch wurde neben der Arbeit, unbemerkt von den anderen Beteiligten geführt. Auch bei den Gesprächen während des Projekts trat der Autor immer in der Rolle des Softwarearchitekten bzw. Projektleiters auf. Erst nach Abschluss des Projekts führte er einige Gespräche in der Forscherrolle. Durch dieses verdeckte Vorgehen wurde vermieden, dass sich die anderen Projektbeteiligten unter wissenschaftlicher Beobachtung wähnten und ihr Verhalten änderten.

5.3.2 Ablauf und Rollen des Projekts

Der Auftrag für den Prototyp des Bestellerfassungs- und Kundeninformationssystems ging an eine Arbeitsgemeinschaft mehrerer Firmen. Ein großer internationaler Konzern übernahm die Generalunternehmerschaft. Ein deutsches Beratungshaus fungierte als externer Onshore-Dienstleister. Die Offshore-Entwicklung führte eine Projektgruppe eines großen indischen Offshoring-Dienstleisters durch. Die Offshore-Komponente war vom Auftraggeber aus Kostengründen explizit erwünscht. Die meisten Manager des Auftraggebers und die späteren Benutzer der Anwendung sprachen und verstanden nur schlecht Englisch, sodass sie nicht direkt mit dem Offshore-Dienstleister kommunizieren konnten. Vom Onshore-Dienstleister erwartete der Auftraggeber daher die Rolle des zentralen Ansprechpartners, der die Arbeiten des Offshore-Teams koordiniert. Die Entwicklung des Prototyps dauerte vier Monate.

In Abbildung 5.6 sind die Akteure des Projekts in der in Abschnitt 4.3.2 erläuterten Notation dargestellt. Ihre Aufgaben werden im Folgenden beschrieben:

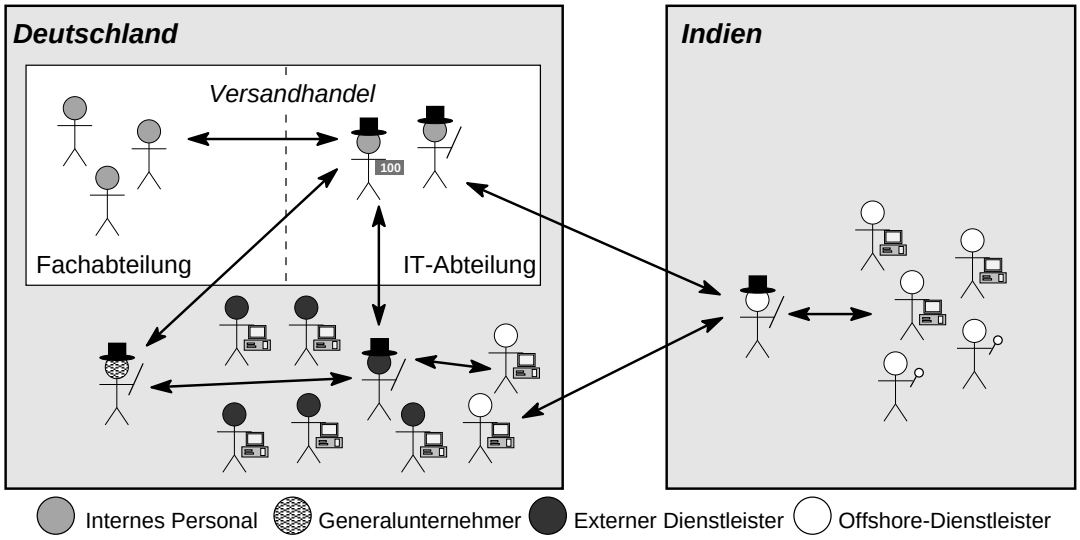


Abbildung 5.6: *Kooperationsmodell des Empirieprojekts Alpha*

Auftraggeber

Der Auftraggeber stellte fachliche und technische Ansprechpartner für das Projekt. Die fachlichen Ansprechpartner gehörten der Managementebene an. Direkten Kontakt zu den späteren Anwendern des Systems erhielten die Dienstleister nicht. Die technischen Ansprechpartner konnten Auskunft über das Altsystem und über die bestehende Systemlandschaft, insbesondere das zentrale Host-System, geben.

Das gesamte Entwicklungsprojekt stand unter intensiver Beobachtung des IT-Managements des Auftraggebers, da die Prototypentwicklung der erste Schritt einer grundlegenden Modernisierung der IT-Landschaft sein sollte.

Generalunternehmer

Der Projektmanager des Generalunternehmers war zuständig für die Budget- und Zeitplanung. Er hielt den Kontakt mit dem IT-Management des Kunden und kümmerte sich um die vertragliche Abwicklung. Die anderen Akteure des Generalunternehmers sind für die Diskussion in dieser Arbeit nicht relevant.

Onshore-Dienstleister

Das Onshore-Team bestand aus einem Softwarearchitekten und einem Team von bis zu fünf Entwicklern. Der Architekt programmierte im Team mit. Gleichzeitig fungierte er als Projektleiter aufseiten des Onshore-Dienstleisters. Er war verantwortlich für den Kontakt mit dem Offshore-Dienstleister und die Qualitätssicherung des offshore erstellten Quelltextes. Im Sinne des Organisationsmusters „Architect Controls Product“ von Coplien [Coplien u. Harrison 2004] steuerte er die strategische technische Ausrichtung des Projekts.

Das Onshore-Team wurde im Verlauf des Projekts vergrößert. In der ersten Iteration waren neben dem Architekten zwei Onshore-Entwickler beteiligt. In späteren Iterationen kamen drei weitere Entwickler hinzu. Alle Onshore-Entwickler konnten die Ansprechpartner des Auftraggebers kontaktieren. Meist wurden Fragen jedoch gesammelt und durch den Softwarearchitekten mit dem Auftraggeber geklärt.

Offshore-Dienstleister

Das Offshore-Team bestand anfangs aus zwei, später aus bis zu sechs Mitarbeitern. Offiziell war einer von ihnen der Projektleiter, die anderen Entwickler. Während der Entwicklung wurde erkennbar, dass die Entwickler intern verschiedene Rollen ausübten. So war beispielsweise die Implementierung neuer Funktionen von der Implementierung der Komponententests getrennt.

Ein deutschstämmiger Manager des indischen Dienstleisters war im deutschsprachigen Raum stationiert. Er diente als Brückenkopf der Offshore-Entwickler, arbeitete eng mit dem Onshore-Team zusammen und vermittelte bei Problemen zwischen den Teams.

5.3.3 Einordnung in die Übersichtskarte des IT-Sourcings

In Abbildung 5.7 ist das Projekt in die Übersichtskarte des IT-Sourcings (siehe Abschnitt 2.2.2) eingeordnet.

Es handelt sich bei diesem Projekt um Application Outsourcing, da eine neue Anwendung entwickelt wurde *(Grad Geschäftsorientierung)*. Mit dem Auftraggeber und dem Onshore-Team in Deutschland und dem Offshore-Team in Indien ist das Projekt als Offshore Sourcing einzustufen *(Standort)*. Der Generalunternehmer, das deutsche Beratungsunternehmen und der indische Dienstleister gehören nicht zum Konzern des Auftraggebers, daher ist das Outsourcing extern *(Finanzielle Abhängigkeit)*. Da fast die gesamte Entwicklung der Anwendung outgesourct wurde, handelt es sich um Totales Outsourcing *(Grad externer Leistungsbezug)*. Bezüglich des *zeitlichen Aspekts* fällt das Projekt in die Kategorie Outsourcing, da die Leistung an einen externen Dienstleister vergeben wurde. Mit dem Onshore- und dem Offshore-Team gibt es zwei Leistungersteller, daher spricht man von Double Sourcing *(Anzahl Leistungsersteller)*.

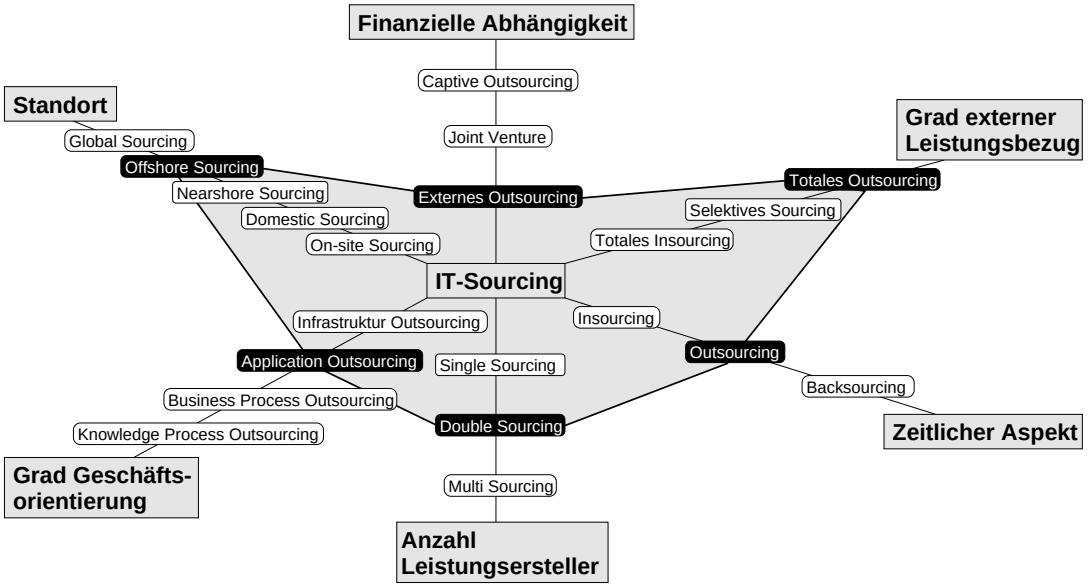


Abbildung 5.7: *Einordnung von Projekt Alpha in die IT-Sourcing-Map*

5.3.4 Der WAM-Ansatz als Methodenrahmen

Als softwaretechnische Grundlage des Empirieprojekts Alpha diente der in Abschnitt 5.2.2 vorgestellte Werkzeug & Material-Ansatz.

In verteilten Projekten wurde der WAM-Ansatz bisher noch nicht eingesetzt. Da im WAM-Ansatz eine offene Kommunikation zwischen allen Beteiligten angestrebt und explizit unterstützt wird, schien er gut mit dem Dual-Shore-Modell vereinbar zu sein. Aufgrund dieser Überlegungen wurde der WAM-Methodenrahmen im Vorfeld als sehr gut geeignet für global verteilte Entwicklungsprojekte nach den in dieser Arbeit erarbeiteten Konzepten eingeschätzt und daher als Basis für das Projekt eingesetzt. Als dazu passendes agiles Vorgehensmodell wurde Extreme Programming gewählt.

5.3.5 Gestaltung der verteilten Arbeit im Empirieprojekt

Im Rückblick lässt sich das Projekt in Orientierung an die Anpassungen des Entwicklungsprozesses in die Vorbereitungsphase und drei Entwicklungsphasen einteilen. Bezüglich vieler Aspekte war ein großer Unterschied zwischen der ersten, onshore gemeinsam durchgeführten Iteration und folgenden Iterationen mit verteilter Arbeit bemerkbar.

Vorbereitungsphase

In der Vorbereitungsphase wurden die Grundlagen für die Projektarbeit und die Zusammenarbeit der Teams gelegt. Der zeitliche Ablauf des Projekts und die Basisanforderungen an den Prototyp wurden mit dem Auftraggeber bei verschiedenen Treffen diskutiert. Dabei entstanden mehrere Powerpoint-Prototypen, welche die gewünschte Funktionalität umrissen.

Gleichzeitig nahmen Onshore- und Offshore-Teams jeweils an Kulturtrainings teil. Dieses bestand für das Onshore-Team aus einem eintägigen Workshop mit dem in Deutschland stationierten Manager des indischen Dienstleisters. Für das Offshore-Team wurde ein mehrtägiger Workshop durchgeführt. Inhalte der Kultrainings waren Besonderheiten der verteilten Entwicklung im Allgemeinen und spezifische Verhaltensweisen, Denkmuster und Regeln von Indern und Deutschen im Speziellen. Der Manager des indischen Dienstleisters kannte beide Länder aus seiner langjährigen Tätigkeit und konnte auch spezielle kulturelle Fragen beantworten. Durch diese Workshops konnten Vorurteile und Berührungsängste im Vorfeld abgebaut werden.

Anschließend fand ein Kickoff-Treffen der wichtigsten Akteure on-site beim Auftraggeber statt. Nach dem persönlichen Kennenlernen und einer Vorstellung des geplanten Projektablaufs erhielten die Entwickler erste Einblicke in die fachlichen Prozesse des Auftraggebers.

Erste Entwicklungsphase: Gemeinsame Arbeit onshore

In der ersten Iteration wurde nur onshore entwickelt. Die Onshore-Entwickler wurden dabei im Sinne des Dual-Shore-Modells von zwei Offshore-Entwicklern unterstützt. Ziele dieser ersten Iteration waren die Entwicklung der Basisarchitektur und eines ersten Releases der Anwendung, die Etablierung des Entwicklungsprozesses und ein Einarbeiten in die Technik und die fachlichen Prozesse.

Im Verlauf der Iteration stellte sich heraus, dass die Offshore-Entwickler Kenntnislücken bei den im Projekt eingesetzten Programmiersprachen und Programmbibliotheken aufwiesen. Auch fiel es ihnen schwer, sich die Fachlichkeit zu erarbeiten. Daher diente diese Phase auch zur Schulung der Offshore-Entwickler.

Zusammenarbeit mit agilen Methoden vor Ort: Das Onshore-Entwicklungsteam hatte langjährige Erfahrung mit agilen Methoden sowohl aus eigenen Entwicklungstätigkeiten als auch aus Beratung, Schulung und Coaching. Das Offshore-Team hingegen war agiles Arbeiten nicht gewohnt. Viele agile Techniken waren neu für die Offshore-Entwickler. Ihr Unternehmen war CMMI Level 5 zertifiziert und arbeitete in der Regel nach dem Wasserfallmodell (vgl. die Diskussion von Agilität und CMM in Abschnitt 3.2.4).

Die indischen Entwickler wurden nicht nur in fachlichen und technischen Fragen, sondern auch im agilen Vorgehen geschult. Da sie diesem Vorgehen aufgeschlossen gegenüberstanden, lernten sie schnell und konnten nach einer gewissen Eingewöhnungsphase der agilen Vorgehensweise folgen. Sie gaben an, dass ihnen die Arbeit Spaß machte. Vor allem waren sie erstaunt, wie viele Freiheiten ihnen beim Vorgehen und bei der Implementierung gewährt wurden. Sie konnten alle Tätigkeiten ausüben, vom Entwurf über Entwicklung und Test bis zur Integration ins aktuelle Release. Aus Indien waren sie ein stark arbeitsteiliges Vorgehen mit einer ausgeprägten Hierarchie von Verantwortlichkeiten gewohnt. Im Forschungstagebuch ist dazu angemerkt: „Aufgrund der starken Arbeitsteilung kennen sie [die Inder] einige Techniken nicht, z. B. sind Design Patterns dem Namen nach bekannt, sonst aber Sache der Project Manager."

Es war schwierig, den Kunden am Entwicklungsprozess zu beteiligen. Die fachlichen und technischen Ansprechpartner mussten sich erst daran gewöhnen, sich über die Erstellung eines Pflichtenheftes und die Abnahme von Releases hinaus einzubringen. Auf Beschluss des Managements auf Kundenseite, das dem Projekt eine hohe Bedeutung zumaß, konnten die Entwickler nicht direkt mit den Anwendern reden. Stattdessen wurde die Definition von Anforderungen und Abstimmung mit dem Management des Kunden vorgenommen. Viele Fragen konnten dadurch nicht kurzfristig telefonisch geklärt werden. Anfragen mussten schriftlich per E-Mail gestellt werden und wurden erst nach bis zu zwei Tagen beantwortet.

Die Mitwirkung am Projekt wurde von einigen Mitarbeitern des Kunden anfangs auch eher als Pflicht und nicht als Chance wahrgenommen, wie folgender schriftlicher Kommentar zu einer Präsentation zeigt, bei der um Kommentare zur Gestaltung einer Maske gebeten wurde: „Irritierend ist schon die Frage nach dem ‚Vorschlag von uns, wie so etwas aussehen könnte'. Gerade das ist doch Aufgabe [des Dienstleisters] und dafür werden sie doch auch u. a. bezahlt. Basis: Pflichtenheft und aktuelle Anwendung."

Kulturelle Unterschiede zwischen den Entwicklern: Wie in Kapitel 2.2.1 dargestellt wurde, ist Indien das häufigste Zielland von Offshoring-Aktivitäten, trotz einer von Europa oder Nordamerika stark abweichenden Kultur. Die kulturellen Unterschiede zwischen den Akteuren von unterschiedlicher Nationalität wurden bei der gemeinsamen Entwicklungsarbeit am selben Rechner sehr deutlich. Ein Auszug aus dem Forschungstagebuch zeigt z. B. unterschiedliche Erwartungen an die Arbeitszeiten: „Sie haben in Indien viele Feiertage pro Monat, an denen nicht gearbeitet wird, ansonsten sind Überstunden normal – das muss einkalkuliert werden."

Aus Sicht des Onshore-Teams machten sich vor allem Unterschiede in Bezug auf die Gründlichkeit der Aufgabenerledigung störend bemerkbar. So fanden sich in den Implementierungen des Offshore-Teams häufig unbenutzte Variablen, auskommentierte Zeilen und vergessene Fälle in Fallunterscheidungen und Komponententests. Auch Wissenslücken und mangelnde Implementierungserfahrungen konnten nicht verborgen

werden. Durch den offenen Umgang miteinander konnte man sich aber gegenseitig helfen.

Weitere Auszüge aus dem Forschungstagebuch zeigen, dass die Onshore-Entwickler öfter auf ihre indischen Kollegen warten mussten: „[Die indischen Entwickler sind] 1/2-Stunde zu spät, wie gestern." Zur Einordnung und Bewältigung solcher Vorkommnisse erwiesen sich die zur Vorbereitung durchgeführten Kulturtrainings als sehr hilfreich, genauso wie das Selbststudium mit entsprechender Literatur, z. B. [Kobayashi-Hillary 2004]. So konnten die Teams trotz dieser Probleme gut zusammenarbeiten: „Kommunikation klappt trotz starkem Akzent ganz gut. Atmosphäre sehr freundlich."

Fachliche Anforderungen: Die fachliche Domäne dieses Projekts – die telefonische Erfassung von Bestellungen – war verhältnismäßig komplex. Die Anwender sollten die Kunden- und Artikeldaten schnell und in einer beliebigen Reihenfolge eingeben und auch spezielle Kundennachfragen beantworten können. Daher waren weitreichende Bedienunterstützungen und Automatismen vorgesehen, z. B. eine unkomplizierte Übernahme von Daten in weitere Masken, Dialoge für zusätzliche Eingaben in bestimmten Bestellsituationen und automatische Änderungen auf nicht-aktiven Registerkarten bei Auswahl bestimmter Operationen.

Wegen dieser Komplexität waren die Anforderungen für die Offshore-Entwickler nicht einfach zu verstehen. Viele von ihnen hatten vorher an recht einfachen, genau spezifizierten Webanwendungen mit stabilen Anforderungen gearbeitet. Durch die gemeinsame Arbeit am Rechner konnte fachliches und technisches Wissen jedoch direkt vermittelt werden.

Entwurf der Softwarearchitektur: Die Definition der grundlegenden Softwarearchitektur wurde von einem erfahrenen Team des Onshore-Dienstleisters durchgeführt. Wie Erfahrungen aus anderen Projekten zeigen, ist es wichtig, dass das Architekturteam dabei von Angesicht zu Angesicht kommuniziert (vgl. [Leszak u. Meier 2007]). Die Offshore-Entwickler wurden an der Architekturdefinition nicht direkt beteiligt.

Es entstand eine Schichtenarchitektur mit zehn Schichten, deren Grundstruktur über die Laufzeit des Projekts im Wesentlichen unverändert blieb. Innerhalb der Schichten wurden Subsysteme angeordnet. Während der Entwicklung kamen mehrere Subsysteme hinzu. Die Softwarearchitektur wurde in verschiedenen Sichten dokumentiert: Die statische Sicht wurde vorwiegend genutzt, um den Entwicklern die Softwarearchitektur zu verdeutlichen und Architekturanalysen durchzuführen. In Anhang B.1 wird die statische Architektur der Anwendung mit Abbildungen und einem erläuternden Text ausführlicher vorgestellt.

Die dynamische Sicht wurde für ausgewählte Abläufe während der Entwicklung genauer betrachtet. Die Verteilungssicht unterstützte insbesondere Diskussionen mit dem Auftraggeber über die Einbindung des bestehenden Host-Systems. Die fachliche Sicht mit

Anwendungsfällen und Geschäftsprozessen wurde bei den Iterationsplanungsrunden zwischen Anwendern und Entwicklern diskutiert.

Zweite Entwicklungsphase: Erste Erfahrungen mit verteilter Arbeit

In der zweiten Entwicklungsphase kehrten die beiden Offshore-Entwickler nach Indien zurück und stellten dort ein Team zusammen. Anschließend wurden zwei Iterationen mit getrennten Teams durchgeführt. Die Entwicklungsgeschwindigkeit war anfangs deutlich geringer als zuvor, da die neuen Entwickler eingearbeitet werden mussten. Dies traf vor allem auf das Offshore-Team zu, aber auch onshore kamen neue Entwickler hinzu.

Beide Teams arbeiteten auf derselben Quelltextbasis. Bei der Auswahl der Aufgaben wurde darauf geachtet, dass die betroffenen Stellen im Quelltext möglichst disjunkt waren. Die Offshore-Entwickler checkten ihre Implementierungen nicht direkt in das CVS ein. Sie schicken sie per E-Mail an den Softwarearchitekten, der sie abnahm und in die gemeinsame Quelltextbasis integrierte. Während die onshore entwickelten Teile der Anwendung eine gute Qualität aufwiesen, gab es Probleme mit den Implementierungen des Offshore-Teams. Diese erfüllten teilweise die Anforderungen nicht, hatten eine schlechte innere Qualität oder kamen verspätet.

Zusammenarbeit nach dem Dual-Shore-Modell: Die Zusammenarbeit der Teams wurde hauptsächlich durch ihre geografische und organisatorische Distanz bestimmt. Durch die geografische Entfernung zwischen Deutschland und Indien kannten die meisten Entwickler das andere Team nicht. Kulturelle Unterschiede wirkten sich deutlich aus. Die organisatorische Teilung war bei der Kommunikation spürbar. Gerade auf der indischen Seite wurde Wert auf formale Prozesse gelegt. Die zeitliche Distanz fiel dagegen nicht besonders ins Gewicht. Abstimmungen waren wegen des geringen Zeitunterschieds mit etwas Planung möglich. So wurden die meisten Besprechungen in einem Zeitfenster durchgeführt, in dem in Indien früher Nachmittag, in Deutschland Vormittag war.

Bei der Umsetzung des Dual-Shore-Modells waren Einschränkungen durch die recht geringe Laufzeit des Projekts und das knappe Budget nötig. So konnten Besuche beim anderen Team nur in eingeschränktem Umfang erfolgen. Während beim Kickoff-Treffen und in der ersten Entwicklungsphase Offshore- und Onshore-Entwickler onshore zusammenarbeiten konnten, konnten in den folgenden Entwicklungsphasen Aufenthalte beim anderen Team nur begrenzt finanziert werden. Entschärfend auf dieses Problem wirkte, dass der onshore arbeitende Manager des Offshore-Dienstleisters den Kontakt zum Offshore-Team hielt.

Die Kommunikationskanäle zwischen den Teams wurden unterschiedlich stark genutzt (vgl. Abbildung 5.6). Die Kommunikation zwischen Onshore- und Offshore-Entwicklungsteam erfolgte in der Regel über die Projektleiter. Nur in seltenen Fällen diskutierten

die Entwickler beider Seiten Fragestellungen direkt miteinander. Der Kanal zwischen dem Offshore-Team und dem Auftraggeber wurde kaum genutzt. Stattdessen lief die Kommunikation über den externen Dienstleister oder den Generalunternehmer. Das war darin begründet, dass die Mitarbeiter des Auftraggebers ungern Englisch sprachen und dass der Generalunternehmer bei Entscheidungen beteiligt werden wollte.

Während in der ersten Entwicklungsphase Informationen vor Ort von Angesicht zu Angesicht weitergegeben werden konnten, mussten Fragen ab der zweiten Phase über die Distanz beantwortet werden. Das Offshore-Team musste daher auf die Beantwortung von Fragen und Feedback zu Implementierungen länger warten.

Infrastruktur: Das Hauptproblem im Bereich der Infrastruktur war die Netzanbindung des Offshore-Teams. Zitat aus dem Tagebuch: „Sie haben in Indien schnelle Rechner, aber [eine] sehr langsame Internetverbindung." Als Konsequenz bevorzugten die Akteure oft asynchrone Kommunikation per E-Mail anstelle von synchroner per Instant Messaging. Im Projekt wurde bevorzugt Open-Source-Software verwendet. Daher konnten Onshore- und Offshore-Teams meist dieselbe Software einsetzen und Probleme wegen inkompatibler Programme vermeiden.

Aufgabenteilung und Einsatz agiler Methoden: Zwischen der ersten und den beiden folgenden Entwicklungsphasen lassen sich weitere deutliche Unterschiede feststellen. Die Offshore-Entwickler, die während der ersten Iteration vor Ort arbeiteten und auch den Auftraggeber kennenlernen konnten, waren sichtlich motiviert. Offshore-Entwicklern, die in späteren Iterationen ins Team kamen, fehlte dieser direkte Kontakt zum Auftraggeber und zum Onshore-Team. Der Offshore-Projektleiter berichtete davon, dass es ihm schwerfiel, diese Entwickler in das Team zu integrieren und zu motivieren.

Aus Kostengründen konnte den Offshore-Entwicklern kein erfahrener Coach für agile Entwicklungsprojekte zur Seite gestellt werden. Es gelang dem Offshore-Team nur zum Teil, agile Techniken einzuführen. Im Offshore-Arbeitsumfeld stieß man auf Widerstand bei Kollegen und dem Management. Viele waren nicht dazu bereit, ihre Arbeitsweisen anzupassen. So fielen auch die onshore eingearbeiteten indischen Mitarbeiter schnell in alte, gewohnte Prozesse zurück.

Die Aufgabenverteilung sah vor, dass die Anforderungen vom Onshore-Team aufgenommen werden. Die Iterationsplanungstreffen fanden on-site beim Auftraggeber ohne Beteiligung des Offshore-Teams in Abständen von etwa drei Wochen statt. An diesen Sitzungen nahmen viele Akteure aller beteiligten Unternehmen teil. Benutzer der Anwendung waren jedoch weiterhin nicht darunter, sodass bei den Treffen teilweise fachliches Wissen fehlte. Der Auftraggeber nahm das agile Vorgehen im Projektverlauf besser an und beteiligte sich zunehmend am Entwicklungsprozess. Aufgaben, die direkte Kommunikation mit dem Auftraggeber erforderten, wurden während des gesamten Projekts vom Onshore-Team wahrgenommen.

Anforderungsübermittlung und Implementierung: Der Softwarearchitekt bereitete die Anforderungen für das Offshore-Team als User Storys in erprobter agiler Weise auf. Obwohl die User Storys ausführlicher als in früheren Projekten des Onshore-Dienstleisters waren, kam das Offshore-Team mit ihnen nicht gut zurecht. Man hatte Probleme damit, umfangreichere Storys in Tasks aufzuteilen und diese in mehreren Gruppen parallel zu bearbeiten. Zudem implementierte man so, wie die User Storys interpretiert wurden. Diese Interpretation war oft nicht das, was das Onshore-Team und der Auftraggeber gemeint hatten. Das Onshore-Team führte dieses Verhalten auf mangelnde Eigenverantwortung zurück, da sich die Offshore-Entwickler bei Verständnisproblemen nicht aktiv um Klärung bemühten.

Durch diese Probleme mussten Funktionen mehrfach überarbeitet werden. Da auch die technische Qualität der Implementierungen des Offshore-Teams schlechter als erwartet war, kam es zu größeren Verzögerungen. Der Softwarearchitekt war einen großen Teil der Zeit mit der Klarstellung der Anforderungen, der Analyse des ihm zugeschickten Quelltextes und der Anpassung und Integration des Quelltextes in das Versionsverwaltungssystem beschäftigt.

Dem Offshore-Team wurden als Konsequenz daraus übersichtlichere Storys geschickt. Dies verursachte jedoch deutliche Mehrarbeit für das Onshore-Team und schränkte die Freiheit der Offshore-Entwickler ein. In der Folge wurde das Onshore-Team um weitere Entwickler ergänzt, um onshore größere Teile der Anwendung als ursprünglich vorgesehen zu implementieren.

Die Arbeit mit einer gemeinsamen Softwarearchitektur funktionierte nach einer Phase der Eingewöhnung zufriedenstellend. Die Einhaltung der Architekturregeln wurde regelmäßig überprüft. Sowohl das Onshore- als auch das Offshore-Team begingen kaum Architekturverletzungen. Diejenigen Verletzungen, die doch auftraten, ließen sich schnell korrigieren. Durch diese Regeln und ein einheitliches Vorgehen bei der Programmierung konnten die Teams trotz ihrer räumlichen und zeitlichen Entfernung gut mit einer gemeinsamen Softwarearchitektur arbeiten (vgl. auch das Organisationsmuster „Standards Linking Locations“ bei [Coplien u. Harrison 2004]).

Dritte Entwicklungsphase: Architekturzentrierte Entwicklung

Die ersten positiven Erfahrungen mit der Arbeit mit einer gemeinsamen Softwarearchitektur führten zur dritten Entwicklungsphase, in der architekturzentriertes Arbeiten explizit gemacht wurde. Die Unterscheidung von zweiter und dritter Entwicklungsphase war nicht von vornherein geplant. Sie wurde während der Projektlaufzeit auch nicht offiziell vorgenommen. Sie ist vielmehr ein Ergebnis der nachträglichen Analyse, bei der ein Einschnitt im Projektablauf deutlich wurde.

Aus der Reflexion über den Forschungsgegenstand, die bei der Action Research Methode regelmäßig erfolgt, entstanden Maßnahmen zur Verbesserung des Entwicklungsprozesses. Als wesentliche Änderung wurde architekturzentriertes Arbeiten forciert. Dabei

wurden Component Tasks als Erweiterung von Story-Cards eingeführt. Die Anforderungen konnten durch den Einsatz von Component Tasks formaler und detaillierter beschrieben werden. So gelang es, die beschriebenen Qualitätsprobleme der zweiten Entwicklungsphase anzugehen. Zusätzlich wurde die Abnahmeprozedur durch den Softwarearchitekten ausgeweitet und es wurden mehr Rückkopplungszyklen eingeführt.

Die Maßnahmen und Ergebnisse aus dieser Entwicklungsphase werden in den nächsten Abschnitten detailliert diskutiert.

5.3.6 Architekturzentrierte Entwicklung

Die Softwarearchitektur von Empirieprojekt Alpha wurde von Beginn an onshore entwickelt und vom Onshore-Softwarearchitekten gepflegt. Änderungen an der Architektur, die sich durch die Entwickler ergaben, wurden regelmäßig eingepflegt. Diese Arbeiten mit der Architektur wurden auch schon in den ersten Entwicklungsphasen durchgeführt. Ab der dritten Entwicklungsphase wurden sie bewusst verstärkt und den Entwicklern gegenüber expliziter gemacht. Die Entwickler wurden aufgefordert, sich mehr Gedanken um die Architektur zu machen. Begriffe und Konzepte der Architektur spielten in mehr Bereichen der Entwicklung eine Rolle.

Im Folgenden werden die Maßnahmen im Einzelnen diskutiert.

Vermittlung von Wissen über die Architektur

Die Architekturdokumentation wurde schon in den ersten Entwicklungsphasen aktuell gehalten. Sie stand über das Versionsverwaltungssystem allen Entwicklern zur Verfügung. Im Verlauf der zweiten Phase wurde aber deutlich, dass gerade die Offshore-Entwickler Defizite im Verständnis der Architektur hatten und die Dokumente nicht ausreichend beachteten.

Daher wurde allen Entwicklern zu Beginn der dritten Phase noch einmal die Architektur erläutert. Sie wurden aufgefordert, sich Änderungen an der Architekturdokumentation zu erarbeiten und die Architekturregeln strikter zu beachten.

Die Möglichkeiten der Entwickler, die Architektur selbstständig weiterzuentwickeln, werden in Abschnitt 5.4.2 detailliert beschrieben.

Aufteilung in Komponenten

Die Softwarearchitektur der Anwendung war von Anfang an aus Komponenten aufgebaut. Komponenten sind von ihrer Definition her gekapselt, sodass sie weitgehend unabhängig voneinander entwickelt werden können. Ein weiterer Vorteil von Komponenten für global verteilte Entwicklungsprojekte liegt darin, dass Komponenten

unterschiedliche Größen haben und hierarchisch aufgebaut sein können. Dadurch können Aufgaben für unterschiedliche Teams zugeschnitten werden. Es können zunehmend komplexere Aufgaben vom Onshore- zum Offshore-Team abgegeben werden. Auch mit einer agilen Entwicklung sind Komponenten gut vereinbar, wie Stojanovic untersucht hat [Stojanovic et al. 2003b], [Stojanovic et al. 2003a].

Komponenten sollten sich an einer Referenzarchitektur orientieren. Dies gibt den Entwicklern eine zusätzliche Orientierung und erleichtert die Planung. Im Fall des Empirieprojekts Alpha ist die Referenzarchitektur die WAM-Modellarchitektur. WAM ist durch seinen Aufbau mit klar benannten Architekturelementen und explizit beschriebenen Abhängigkeiten zwischen ihnen gut geeignet für eine komponentenorientierte Entwicklung. Das Rahmenwerk JWAM ist selbst aus Komponenten aufgebaut [Wolf et al. 2001].

In der dritten Entwicklungsphase wurde auf Basis der Komponentenstruktur die gemeinsame Verantwortlichkeit eingeschränkt. Änderungen an einigen zentralen Komponenten wurden fortan nur durch bestimmte Teams durchgeführt. Dadurch sank zwar die Flexibilität der Entwickler. Es brachte aber den Vorteil, dass diese wichtigen Teile der Anwendung stabiler und qualitativ besser wurden.

Im nächsten Abschnitt wird gezeigt, wie Komponenten bei der Spezifikation von Anforderungen eingesetzt werden können.

Einführung von Component Tasks

Im Projekt hatten sich einige Nachteile des Einsatzes von Story-Cards gezeigt. So waren die auf diese Art definierten Aufgaben recht umfangreich und erforderten eine eigenständige Erarbeitung. Das Onshore-Team kam mit dieser leichtgewichtigen Anforderungsspezifikation gut zurecht. Für das Offshore-Team war der Umgang mit den Story-Cards nicht so einfach. Das selbstständige Erarbeiten komplexerer Lösungen fiel den Offshore-Entwicklern schwer. Eine solche Selbstständigkeit entsprach nicht ihren gewohnten Arbeitsprozessen, bei denen die Aufgaben genau spezifiziert wurden. Die Beschreibungen der Anforderungen auf den Story-Cards waren für das Offshore-Team nicht detailliert genug. Verweise auf nur in deutscher Sprache vorliegende Dokumente wie Benutzerhandbücher des alten Systems und das Grobpflichtenheft der Neuentwicklung waren mangels Sprachkenntnissen nicht hilfreich. Im Gegensatz zu den Onshore-Entwicklern war es für die Offshore-Entwickler nicht möglich, bei fachlichen Ansprechpartnern des Kunden nachzufragen. Sie baten auch nicht aus eigenem Antrieb das Onshore-Team um Hilfe.

Die Story-Cards enthielten keine expliziten Abnahmekriterien oder Akzeptanztests. So lieferten die Offshore-Entwickler häufig Quelltexte ab, welche die Anforderungen nicht voll erfüllten. Dies führte zu großem Aufwand bei der Qualitätssicherung durch das deutsche Team und zu unnötigen Überarbeitungszyklen.

Nachdem diese Probleme erkannt worden waren, wurden die Anforderungsdefinitionen auf den Story-Cards um Kontext- und Detailinformationen erweitert, um dem Offshore-Team die Arbeit zu erleichtern und Missverständnisse zu verringern. Tabelle 5.1 zeigt beispielhaft, um welche Aspekte die Anforderungsbeschreibung ergänzt wurde. Die zugehörige Skizze findet sich in Abbildung 5.8. Die Struktur der Component Tasks wurde im Projektverlauf aufgrund der gesammelten Erfahrungen verfeinert.

Gliederungspunkt	**Inhalt**
Related story:	Tool for terms of delivery
Type:	Sub tool
Task:	Implementation
Description:	Implement a tool for the input of account information that is integrated into the tool for terms of delivery and automatically shown when the customer chooses to pay by direct debit. The following fields and check routines have to be implemented (see sketch for layout): • account number (German: Kontonummer), based on account number domainvalue • bank routing number (German: Bankleitzahl), based on routing number domainvalue ...
Tests:	• All fields have to be filled in for OK to be enabled. • All domain values have to be valid for OK to be enabled. ...
Effort points:	2
Recorded by:	Joachim Sauer, 11.03.2005
Status:	approved, 01.04.2005

Tabelle 5.1: *Beispiel für einen Component Task*

Eine vergleichbare Anreicherung von Story-Cards um weitere Informationen für Offshoring-Projekte wurde auch von Matthew Simons beschrieben [Ågerfalk u. Fitzgerald 2006]. Solche weiteren Informationen können eine ausführlichere Beschreibung der

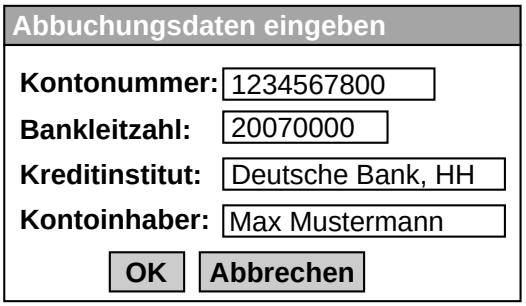


Abbildung 5.8: *Skizze zum Component Task-Beispiel*

Funktionen, Abhängigkeiten zu anderen Systemteilen oder eine Menge von funktionalen Tests sein. Diese Tests können als Akzeptanzkriterium dienen.

Die Besonderheit von Component Tasks besteht in ihrer Beziehung zur Softwarearchitektur. Wie der Begriff „*Component* Task“ deutlich machen soll, werden die Anforderungen nicht nach fachlichen Aspekten, sondern nach Komponenten geschnitten. Die Offshore-Entwickler wurden dabei schrittweise stärker an der Entwicklung beteiligt. Sie steuerten erst einfache Komponenten bei, z. B. Fachwerte für Artikelnummern oder ein Material, das ein Bankkonto abbildet. Später entwickelten sie komplexere Komponenten, z. B. ein Werkzeug zur Eingabe und Anzeige von Lieferkonditionen.

Die Implementierung der Komponenten fiel den Offshore-Entwicklern deutlich leichter als die früheren Story-Cards. Durch die Orientierung an Komponenten gab es weniger Realisierungsalternativen, die abzuwägen waren. Auf diese Art und Weise konnten sinnvolle Vorgaben gemacht werden, ohne die Freiheit der Entwickler zu sehr einzuschränken.

Die Offshore-Entwickler konnten die ihnen zugewiesenen Aufgaben schneller umsetzen. Die Teams waren unabhängiger voneinander. Bei den bis zur zweiten Entwicklungsphase eingesetzten Storys wurde die gleiche Stelle im Quelltext in mehreren Fällen parallel von beiden Teams geändert. Durch den Einsatz von Component Tasks gab es weniger Überschneidungen. Die Teams mussten sich weniger abstimmen. Sofern sie die Schnittstellen der Komponenten beachteten, konnten sie bei der Entwicklung auf ihre eigenen Prozesse und Entwicklungstechniken zurückgreifen.

Ausweitung der Qualitätssicherung

Im Zusammenhang mit der Einführung von Component Tasks wurden auch der Abnahmeprozess und die Qualitätssicherung des offshore erstellten Quelltextes formaler gestaltet. Die fertiggestellten Komponenten wurden eingereicht und dann nach einem festgelegten Plan bewertet.

Dadurch wurde der Prozess der Qualitätssicherung klarer. Die entwickelten Komponenten ließen sich besser unabhängig voneinander testen. Durch die auf den Component Tasks spezifizierten Tests war die grundsätzliche Funktion sichergestellt (siehe Anhang B.3 für ein Beispiel für einen zusätzlichen manuellen Akzeptanztest). Der Softwarearchitekt musste in der Folge seltener Entwicklungen des Offshore-Teams beanstanden.

In Abschnitt 5.4.3 werden weitere Aspekte der Qualitätssicherung diskutiert.

Bewertung

Aus den Trackingdaten ist zu erkennen, dass die beschriebenen Maßnahmen griffen. Es gab in der Folge weniger Fehler. Zudem wurde die Entwicklung effizienter. Im weiteren Verlauf der dritten Entwicklungsphase wurden dem Offshore-Team größere Aufgaben zur eigenständigen Bearbeitung überlassen.

Das Projekt wurde vom Auftraggeber und den beteiligten Organisationen nach seinem Abschluss als erfolgreich bewertet. Der geplante Zeitrahmen wurde zwar um einige Wochen überschritten. Dafür konnten alle Kernanforderungen umgesetzt und die Durchführbarkeit eines Entwicklungsprojekts mit Offshoring-Anteilen nachgewiesen werden. In der Folge entschied sich der Auftraggeber, auch die Entwicklung der produktiven Anwendung mit Offshoring-Anteilen in weiteren Projekten durchzuführen.

Das Dual-Shore-Kooperationsmodell und architekturzentrierte Entwicklung ergänzten sich sehr gut. Mit einer Kombination der Ansätze konnten Onshore- und Offshore-Teams in der dritten Entwicklungsphase eine produktive Zusammenarbeit ohne bedeutende Konflikte umsetzen. Die Kommunikation der Teams mit dem Auftraggeber und dem Generalunternehmer wurde ebenfalls gut unterstützt. Hervorzuheben ist dabei, dass ein sehr agil arbeitendes Onshore-Team mit einem eher noch plangetrieben denkendem Offshore-Team zusammenarbeiten konnte. Über die gemeinsame Architektur waren die Teams miteinander verbunden. Sie konnten ihre jeweilige Arbeitsaufgabe abgrenzen und über architektonische Schnittstellen sicherstellen, dass die Implementierungen zueinanderpassten. Gleichzeitig konnten die Teams ihre internen Prozesse an die jeweiligen lokalen Gegebenheiten, insbesondere die Kenntnisse und Vorerfahrungen der Entwickler und kulturelle Eigenheiten der Unternehmensorganisation und des Hierarchiedenkens, anpassen.

Die Eignung von architekturzentrierter Entwicklung für agile global verteilte Entwicklung zeigt ein Vergleich mit den in Kapitel 3 identifizierten Themen, siehe Tabelle 3.3. Durch die Hinzunahme von architekturzentrierter Entwicklung können im Empirieprojekt Alpha mehrere Themen, die eine eher schlechte Bewertung erhalten haben, positiver bewertet werden. Architekturzentrierte Entwicklung hat insbesondere die sinnvolle Verteilung von Aufgaben zwischen den Teams unterstützt. Die positiven Auswirkungen auf die Kooperation und die Koordination der Teams sowie auf das Training der Onshore- und Offshore-Entwickler wurden im Projekt ebenfalls deutlich. Die erste Phase mit der gemeinsamen Arbeit onshore lief in diesen Aspekten jedoch

wesentlich besser als die späteren. Die richtigen Probleme zeigten sich erst bei der verteilten Arbeit. Die gemeinsame Verantwortlichkeit und die Arbeitszufriedenheit können beim Gesamtansatz ebenfalls besser bewertet werden.

Die Erfahrungen im Empirieprojekt haben gezeigt, wie wichtig eine permanente Adaption des Entwicklungsprozesses und eine Anpassung der verwendeten Techniken sind. So konnten mit der Einführung von Component Tasks einige Schwierigkeiten des Projekts ausgeräumt werden. Die Anforderungsweitergabe mit Component Tasks versetzte die Offshore-Entwickler in die Lage, nicht nur einfache Arbeitsaufträge abzuarbeiten, sondern selber Entwurfsentscheidungen zu treffen und sie innerhalb eines vorgegebenen Rahmens zu implementieren.

Insgesamt kann aus dem Projekt auch aus wissenschaftlicher Sicht ein sehr positives Fazit gezogen werden. Es konnten viele wichtige Erkenntnisse gewonnen werden. Vieles konnte durch Anpassung des Prozesses und der Techniken direkt im Projekt evaluiert werden.

5.3.7 Problemfelder der architekturzentrierten Entwicklung

Aufgrund der Erfahrungen im Empirieprojekt konnten fünf Problemfelder der architekturzentrierten Entwicklung in global verteilten Softwareprojekten identifiziert werden. Diese sind für ein Gelingen der Projekte wesentlich.

Evolution und Dokumentation der Architektur: Man muss die Entwicklung der Architektur steuern, damit sie softwaretechnischen und fachlichen Anforderungen genügt. Dabei sind auch zukünftige Anforderungen einzuplanen. Inkonsistente Erweiterungen können schwer behebbare Architekturverletzungen zur Folge haben. Architekturvorgaben müssen für alle Beteiligten verständlich sein und den Entwicklern ausreichende Freiheiten bei der Entwicklung lassen.

Innere Qualität und Komponententests: Der qualitative Zustand der Anwendung muss permanent geprüft werden, da interne Schwierigkeiten des anderen Teams und Kompromisse bei der Qualität durch die Entfernung nicht sofort sichtbar werden können. Zu einer guten inneren Qualität tragen Komponententests bei. Wenn die Testabdeckung zu gering ist, kann die Fehlerbehebung großen Aufwand nach sich ziehen.

Anforderungsdefinition: Die Anforderungen müssen für die Entwickler aufbereitet werden. Dabei ist der richtige Detailgrad zu wählen. Der fachliche und technische Hintergrund muss klar vermittelt werden, um Fehlentwicklungen und unnötige Rückfragen zu vermeiden. Die Abnahme der umgesetzten Anforderungen muss durch einen kontrollierten Prozess erfolgen.

Kontrolle und Steuerung der Entwicklung: Es kann durch die verteilte Entwicklung schwer sein, einen Einblick in den Entwicklungsstand beim entfernten Team zu gewinnen. Diesem Team sollte ausreichend Raum für die eigene Gestaltung des

Entwicklungsprozesses gelassen werden. Gleichzeitig sollte die Entwicklung aber noch kontrollier- und steuerbar bleiben.

Aufgabenverteilung auf die Teams: Die Aufgaben sollten so auf die Teams aufgeteilt werden, dass sie eigenständig arbeiten können und nicht zu viel über die Entfernung kommunizieren müssen. Daher ist eine geeignete Aufteilung der Aufgaben zu entwickeln. Bei Abstimmungsproblemen zwischen den Teams müssen strittige Punkte geklärt und ggf. Konflikte gelöst werden. Eine entsprechende Architektur kann die Teilbarkeit der Aufgaben verbessern.

5.3.8 Aufgaben eines Softwarearchitekten

In Abbildung 5.9 sind den Problemfeldern entsprechende Aufgaben eines Softwarearchitekten in global verteilten Entwicklungsprojekten zugeordnet. Die Aufgaben und ihre Zuordnung wurden aus der wissenschaftlichen Reflexion der Geschehnisse des Empirieprojekts abgeleitet.

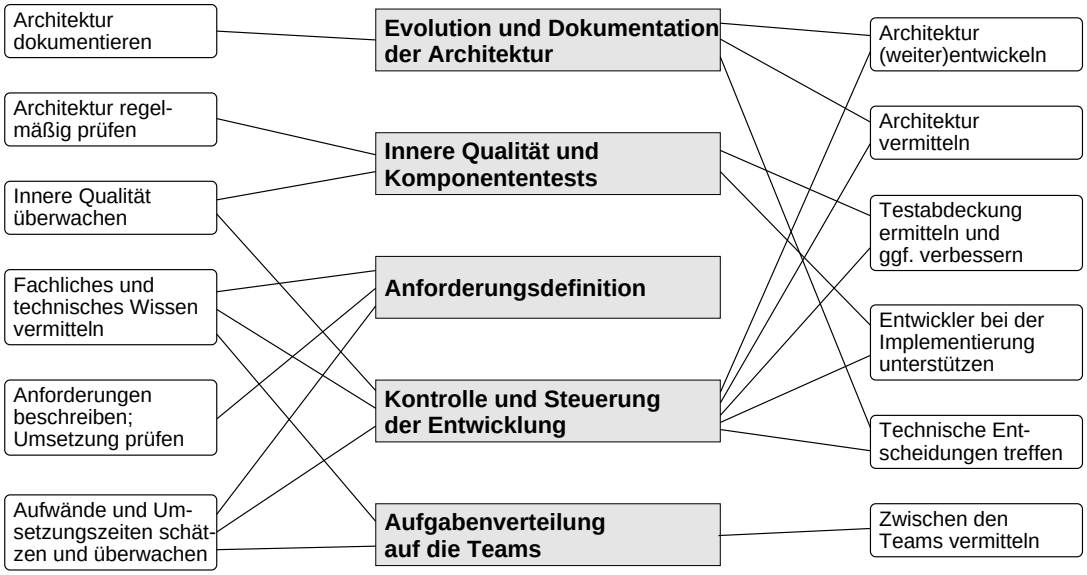


Abbildung 5.9: *Problemfelder (mittig) und Aufgaben eines Softwarearchitekten (links und rechts) in global verteilten Entwicklungsprojekten*

In der nun folgenden Betrachtung dieser Problemfelder und Aufgaben werden jeweils die Unterschiede gegenüber der Arbeit von Softwarearchitekten in nicht-verteilten Projekten diskutiert.

- **Architektur dokumentieren:** In die Dokumentation der Architektur muss mehr Aufwand gesteckt werden, damit sie auch für entfernte Entwickler ver-

ständlich ist und Missverständnisse und damit Fehlentwicklungen verringert werden.

- **Architektur regelmäßig prüfen:** Da ein Architekt für entfernte Mitarbeiter bei Fragen nicht immer ansprechbar ist und die Architekturdokumentation falsch interpretiert werden kann, können leichter Architekturverletzungen entstehen. Daher sollte die Architektur regelmäßig überprüft werden. Bei der Prüfung sollten Besonderheiten von global verteilten Projekten berücksichtigt werden; z. B. sollten Dokumente für alle Beteiligten verständlich sein [Salger 2009].

- **Innere Qualität überwachen:** Neben einer Prüfung der Architektur sollten auch andere Eigenschaften der Anwendung überwacht werden, um eine gute innere Qualität sicherzustellen. Bei Verschlechterungen der inneren Qualität müssen Refactorings geplant werden.

- **Fachliches und technisches Wissen vermitteln:** Die Vermittlung von fachlichem und technischem Wissen an das entfernte Team muss gut geplant werden. Vergleiche die Diskussion der Hauptnutzen eines Softwarearchitekten im nächsten Abschnitt.

- **Anforderungen beschreiben; Umsetzung prüfen:** Die Beschreibung der Anforderungen und die Abnahme ihrer Umsetzung ist in der Regel eine Aufgabe des Projektleiters. Ein Architekt sollte beteiligt werden, da er technisches Wissen mitbringt und die Anforderungen unmissverständlich übermitteln kann. Dies ist nötig, da Offshore-Entwickler vom Kunden und seinen Bedürfnissen stärker als in herkömmlichen Projekten entkoppelt sind. Oft fällt in den hier betrachteten kleinen und mittleren Projekten auch die Rolle des Projektleiters mit der Rolle des Softwarearchitekten zusammen.

- **Aufwände und Umsetzungszeiten schätzen und überwachen:** In Offshoring-Projekten ist mehr Kontrolle nötig. Aufwände und die für ihre Umsetzung benötigte Zeit sind genau zu schätzen und zu überwachen, um den Projektfortschritt beurteilen zu können.

- **Architektur (weiter)entwickeln:** Bei der Entwicklung der Architektur muss der Architekt darauf achten, dass sie sich auch für verteilte Entwicklung eignet, z. B. durch Kapselung und entkoppelte Komponenten. Die Weiterentwicklung muss gesteuert werden, damit alle Entwickler über Änderungen informiert sind.

- **Architektur vermitteln:** Der Architekt muss verhindern, dass die Teams unterschiedliche Vorstellungen von der Architektur entwickeln. In Projekten mit global verteilten Teams und einer Evolution der Architektur kann das schwierig sein.

- **Testabdeckung ermitteln und ggf. verbessern:** Wie sich in der Feldstudie gezeigt hat, sollte eine hohe Abdeckung mit Komponententests von Anfang an gewährleistet werden.

- **Entwickler bei der Implementierung unterstützen:** Ein Softwarearchitekt sollte den Entwicklern bei schwierigen technischen Fragen zur Seite stehen. Durch die Verteilung ist es schwieriger, Beratungsbedarf zu erkennen und den Entwicklern zu helfen.

- **Technische Entscheidungen treffen:** Bei vielen technischen Entscheidungen ist eine gute Kenntnis der Fachlichkeit und der zukünftigen Entwicklungsrichtung eines Systems wichtig. Diese besitzen Offshore-Entwickler oft nicht. Daher sollte ein Softwarearchitekt die Kontrolle über wichtige Entscheidungen behalten.

- **Zwischen den Teams vermitteln:** Ein Architekt muss bei Konflikten zwischen den Teams vermitteln. Durch die Verteilung sieht man sich – trotz Dual-Shore-Kooperationsmodell – seltener. Eine objektive technische Bewertung und Entscheidung von Streitfragen durch einen Architekten kann Probleme lösen. Vergleiche die Diskussion der Hauptnutzen eines Softwarearchitekten im nächsten Abschnitt.

Zusammenfassend kann man feststellen, dass viele Aufgaben durch die Verteilung mehr Zeit benötigen und stringenter durchgeführt werden müssen, da Probleme durch die Verteilung erst spät sichtbar werden können. Zusätzlich kommen neue Aufgaben hinzu, insbesondere im Bereich der Vermittlung von fachlichem und technischem Wissen, der unmissverständlichen Übermittlung von Anforderungen und der Aufgabenverteilung und Abstimmung der verteilten Teams.

5.3.9 Hauptnutzen eines Softwarearchitekten

Die Untersuchung hat neben diesen fünf Problemfeldern und zwölf Aufgaben drei Hauptnutzen der Arbeit eines Softwarearchitekten bei global verteilter Entwicklung ergeben:

1. Verbinden von fachlichem mit technischem Wissen:

Viele der in der Feldstudie aufgetretenen Probleme erklären sich dadurch, dass die beteiligten Akteure nicht über das benötigte fachliche und/oder technische Wissen verfügen. Daher sollte der Vermittlung von fachlichem und technischem Wissen besondere Aufmerksamkeit gewidmet werden. Hierzu gehört die richtige Organisationsstruktur, wie schon Conway festgestellt hat (siehe Abschnitt 3.4.2). Der Entwurf der Anwendung sollte durch die Fachlichkeit getrieben werden (vgl. [Evans 2003]). Es werden daher Akteure benötigt, die sowohl die fachliche Domäne kennen als auch etwas von der technischen Umsetzung verstehen. Ein Softwarearchitekt kann genau dies leisten und fachliches und technisches Wissen verbinden und den anderen Akteuren vermitteln.

2. Finden einer Balance zwischen starren Vorgaben und Eigenverantwortung der Entwickler:

Wie in Abbildung 5.9 dargestellt wird, ist die Kontrolle und Steuerung der Entwicklung ein zentrales Problemfeld, das Berührungspunkte mit vielen Aufgaben des Softwarearchitekten in global verteilten Entwicklungsprojekten aufweist. Ein Softwarearchitekt kann die Vorgaben machen, innerhalb derer die Entwickler eigenständig arbeiten können. Dies ist insbesondere bei Agilem Offshoring wichtig, wo die direkte Kommunikation zu kurz kommt. Ein Architekt hat die Stellung und das Wissen, um diese Aufgabe zu erfüllen.

3. Vermitteln und Moderieren zwischen den Teams:

Oft ist es verteilten Teams nicht möglich, auftretende Probleme und Meinungsverschiedenheiten untereinander zu lösen. Dies kann zum einen auf den durch die Verteilung fehlenden Gesamtüberblick zurückgeführt werden. Zum anderen beeinflussen fehlende direkte Kommunikation und die Offshoring-bedingte organisatorische Trennung die Problemlösung negativ.

Ein Softwarearchitekt kann in seiner besonderen Rolle zwischen den Teams vermitteln und solche Fragen klären. Er hat neben organisatorischem und fachlichem Wissen auch dafür oft benötigtes technisches Wissen. Dies zeichnet ihn gegenüber Projektleitern aus, die nicht direkt an der Entwicklung beteiligt sind. Wichtig für Architekten ist es, unabhängig zu bleiben und nicht ein Team durch räumliche oder kulturelle Nähe o. Ä. zu bevorzugen.

Ein Vergleich mit den in Kapitel 3 aufgeworfenen Fragestellungen zeigt, dass diese drei Nutzen zentrale Probleme der global verteilten Entwicklung mit agilen Methoden adressieren und Lösungsmöglichkeiten bieten.

5.4 Weiterführende Architekturthemen

Bei der Analyse des Empirieprojekts und in ergänzenden Untersuchungen konnten neue Erkenntnisse zu weiteren interessanten Aspekten mit Bezug zu architekturzentrierter Entwicklung gewonnen werden: Wie kann Architekturwissen in global verteilten Entwicklungsprojekten vermittelt werden? Wie kann die Architekturevolution gestaltet werden? Was ist bei der Qualitätssicherung zu beachten?

Diese Erkenntnisse bieten Hilfestellungen bei der praktischen Umsetzung von architekturzentrierter Entwicklung in global verteilten Projekten. Sie werden in den folgenden Abschnitten diskutiert.

5.4.1 Vermittlung von Architekturwissen

Wie oben ausgeführt wurde, besteht eine der wichtigsten Aufgaben des Softwarearchitekten in global verteilten Entwicklungsprojekten im Management des Wissenstransfers. In diesem Abschnitt werden Erfahrungen auf diesem Gebiet aus zwei Projekten untersucht. Das erste Projekt ist das Empirieprojekt Alpha. Bei diesem wurden Offshore-Entwickler onshore ausgebildet. Darauf folgt die Untersuchung eines zweiten Projekts, bei dem ein System an ein nearshore arbeitendes Team zur Wartung und Weiterentwicklung übergeben wurde. Das Wissen wurde hauptsächlich in Telefonkonferenzen und mit Application Sharing vermittelt.

Erfahrungen aus dem Empirieprojekt Alpha

In der ersten Entwicklungsphase von Projekt Alpha wurden zwei Offshore-Entwickler onshore in die Architektur und die Technik des zu entwickelnden Systems eingearbeitet. Die Basisarchitektur befand sich zu diesem Zeitpunkt noch in der Entwicklung. Sie basierte auf der WAM-Modellarchitektur. Den Offshore-Entwicklern wurde ausreichend Zeit zum Erlernen von architekturzentrierter Entwicklung und der Modellarchitektur eingeräumt. Auf dieser Basis erfolgte der weitere Wissenstransfer. Die Offshore-Entwickler hatten noch keine Erfahrungen mit WAM und der im Projekt praktizierten Form von architekturzentrierter Entwicklung.

Im Einzelnen können die folgenden fünf Schritte der initialen Wissensvermittlung im diskutierten Projekt unterschieden werden:

1. Im ersten Schritt wurden den Offshore-Entwicklern die *Metaphern* des WAM-Ansatzes und spezifische Metaphern der Anwendung erläutert. Die Metaphern bilden eine leicht verständliche Brücke zwischen Konzepten der Anwendungsdomäne und ihrer Ausprägung in der Softwarearchitektur.

2. Ausgehend von den Metaphern erfolgte die Einführung in die im Projekt eingesetzte *Referenzarchitektur*. Durch die überschaubare Menge an Elementen und Beziehungen kann diese recht einfach verstanden werden. Zur WAM-Modellarchitektur gibt es auch gute Dokumentation für den Einstieg im Selbststudium, u. a. Schulungsfolien und Zeitschriftenveröffentlichungen.

3. Den Übergang von den Konzepten der Referenzarchitektur zur Implementierung im Quelltext erleichtern *Rahmenwerke*. Für den WAM-Ansatz liegt mit JWAM ein solches Rahmenwerk vor. Die Offshore-Entwickler konnten sich anhand kleiner Beispielanwendungen in JWAM einarbeiten. Ein solches Rahmenwerk muss nicht schon zu Beginn eines Projekts in allen Einzelheiten verstanden werden. Nach der Erfahrung im Projekt Alpha reicht ein Verständnis der Grundlagen, damit die Entwickler später weitgehend eigenständig arbeiten können.

4. Nach Erarbeitung dieser Grundlagen folgte eine Einführung in die *projektspezifische Architektur*. Bevor die Offshore-Entwickler am Projekt mitwirkten, war ein Teil der Basisarchitektur bereits festgelegt und es gab erste Implementierungen.

5. Im letzten Schritt wurde die *Implementierung* im Detail vermittelt. Anhand kleiner Aufgaben wurden die Offshore-Entwickler in Fachlichkeit, Technik und Architektur eingearbeitet.

Die Erfahrung im Projekt zeigte, dass die Offshore-Entwickler durch diese Schritte in die Lage versetzt wurden, an der architekturzentrierten Entwicklung im Projekt teilzunehmen.

Erfahrungen bei der Übergabe eines Systems

Die folgende Untersuchung basiert auf einem Interview, das der Autor mit einem Softwarearchitekten, der zugleich technischer Projektleiter war, im April 2009 geführt hat. Es wurde ein Einzelinterview auf Basis eines Interviewleitfadens durchgeführt und aufgezeichnet, anschließend in den wichtigsten Teilen transkribiert und danach ausgewertet.

Gegenstand des Interviews waren die Erfahrungen bei der Neuentwicklung einer Anwendung in Deutschland und die spätere Verlegung der Wartung und Weiterentwicklung nach Polen. Die Verlegung war von Anfang an geplant, da man durch sie Kostenvorteile erzielen wollte. Dadurch konnten sich die Entwicklungsteams rechtzeitig darauf einstellen. Um die Übergangsphase möglichst reibungslos zu gestalten, wurden das Dual-Shore-Kooperationsmodell und architekturzentrierte Softwareentwicklung genutzt.

Dual-Shore-Entwicklung: Bis zur Produktivstellung der ersten Version der Anwendung Mitte 2007 wurde mit fünf Mitarbeitern (Vollzeitäquivalent etwa drei Mitarbeiter) ausschließlich onshore gearbeitet. Das polnische Team war bis auf einen einwöchigen Aufenthalt eines Entwicklers beim Onshore-Team, bei dem er über den aktuellen Stand der Entwicklung und die Architektur der Anwendung informiert wurde, nicht beteiligt. Im Anschluss erfolgte die Weiterentwicklung in geringerem Umfang mit drei Onshore-Entwicklern, später nur noch mit einem. Die Offshore-Entwickler wurden regelmäßig telefonisch über den Fortschritt des Projekts informiert. Aufgrund ihrer Auslastung durch andere Projekte konnten sie nicht intensiver beteiligt werden.

Im April 2008 wurden drei Entwickler des polnischen Teams onshore in Deutschland für etwa drei Wochen eingearbeitet. Anschließend entwickelten Onshore- und Offshore-Teams parallel an ihren jeweiligen Standorten. Während das Onshore-Team regulär eingeplante Anforderungen bearbeitete, wurden dem Offshore-Team zunächst spezielle Aufgaben zugeteilt, die der Einarbeitung dienten. Die Kommunikation zwischen den Teams erfolgte in täglichen Telefonaten, bei denen oft Teile der Implementierung

durchgesprochen wurden. Zusätzlich war der Softwarearchitekt zweimal am Offshore-Standort, um dort Architekturanalysen durchzuführen. Bei der Zusammenarbeit zeigten sich nach Ansicht des Softwarearchitekten keine kulturellen Unterschiede. Die Teams kommunizierten auf Englisch und Deutsch ohne Verständigungsprobleme. Darüber hinaus erwies sich auch die technische Infrastruktur als sehr stabil.

Zum Erfolg der Offshore-Entwicklung trugen häufige, teilweise wöchentliche Releases bei. Die Anwendung wurde als Webanwendung realisiert. Die aktuelle Entwicklungsversion war im Web jederzeit von allen Entwicklerteams und von den Gamma-Kunden aufrufbar. Beide Teams arbeiteten agil. Sie konnten schnell einen gemeinsamen Entwicklungsprozess umsetzen. Meist standen sie fast den ganzen Tag über Application Sharing mit paralleler ständiger Sprachverbindung in Kontakt, programmierten in verteilten Paaren und diskutierten anhand der Quelltexte. Nachdem das Budget für die Onshore-Entwickler erschöpft war, wurde ab Anfang 2009 ausschließlich in Polen entwickelt. Der Softwarearchitekt unterstützte das Offshore-Team weiterhin bei Fragen.

Architekturzentrierte Softwareentwicklung: Die Onshore-Entwickler waren gewohnt, architekturzentriert zu arbeiten. Da sie wussten, dass die Anwendung später von fremden Entwicklern gewartet werden sollte, achteten sie besonders auf eine konsistente, gut dokumentierte Architektur und verständliche Quelltexte. Die Architektur war an WAM angelehnt. Aufgrund der spezifischen Gegebenheiten des Projekts, vor allem der zentralen Anbindung eines komplexen externen Webservices, gab es einige architektonische Besonderheiten. So wurden nur wenige Materialien mit wenigen fachlichen Operationen implementiert. Bei diesen wurde ein generischer Ansatz gewählt, der eine automatische Adaption an Änderungen der externen Produktpalette ermöglichte. Dafür wurde spezifisches fachliches Wissen in einer Reihe von Services implementiert. Diese wurden – ähnlich wie im Empirieprojekt Alpha – nach weitgehend voneinander unabhängigen fachlichen Bereichen und nicht nach technischen Erfordernissen geschnitten. Dadurch war die Anwendung gut verständlich. Paralleles Arbeiten wurde durch die lose gekoppelten Services erleichtert, da Services weitgehend unabhängig voneinander entwickelt werden konnten.

Bei der Vermittlung von Architekturwissen setzte der Softwarearchitekt Skizzen von UML-Diagrammen ein. Hauptsächlich coachte er die Offshore-Entwickler jedoch anhand ihrer konkreten Fragen durch gemeinsames Implementieren und Reviews der Quelltexte. Für die ersten Implementierungsaufgaben der Offshore-Entwickler wurde das Refactoring einer zentralen Komponente ausgewählt. Diese Aufgabe war eigens für diesen Zweck zurückgestellt worden. Da diese Aufgabe Änderungen an allen wichtigen Systemkomponenten – von der Benutzungsoberfläche bis zur Persistenzschicht – erforderte, konnten die Offshore-Entwickler die Systemarchitektur in kurzer Zeit kennenlernen. Details zum hier bewusst eingesetzten aufgabengetriebenen Prozess der Wissensweitergabe sind in [Koch u. Sauer 2010] zu finden.

Die Entwicklung der Anwendung und die Übergabe an ein entfernt arbeitendes Team wurden von allen Beteiligten als sehr erfolgreich angesehen. Als wesentliche Faktoren für diesen Erfolg können der Einsatz des Dual-Shore-Modells und architekturzentrierte Entwicklung gesehen werden. Sehr hilfreich war auch, dass die spätere Verlegung schon zu Projektbeginn bekannt war, entsprechend gut vorbereitet werden konnte und von allen Beteiligten akzeptiert wurde. Der Erfolg des Projekts ist ein Hinweis darauf, dass die empfohlenen Techniken nicht nur bei der verteilten Entwicklung einer Anwendung, sondern auch bei der Übergabe einer Anwendung an ein anderes Team nützlich sein können.

5.4.2 Architekturevolution durch Entwickler

In Abschnitt 5.1.3 wurde der Begriff der Architekturevolution eingeführt. Die Weiterentwicklung von Ist- und Sollarchitektur ist bei verteilten Projekten besonders kritisch. Wenn Architekten und Entwickler räumlich getrennt sind, ist es schwieriger, Änderungen zu kommunizieren.

Es wäre einfacher, wenn Änderungen an der Architektur nur vom Softwarearchitekten vorgenommen werden dürften. Entwickler würden dann nur innerhalb dieser Vorgabe arbeiten. Ein Ansatz, mit dem diese Art von Architekturevolution möglich ist, ist die modellgetriebene Softwareentwicklung (engl. model driven architecture, MDA) [Kleppe et al. 2003]. Dabei gibt der Architekt die Gesamtarchitektur in einem MDA-Blueprint vor. Die Offshore-Entwickler setzen sie um. Diese Arbeitsteilung ist einfach verständlich. MDA wurde schon in einigen Offshoring-Projekten erfolgreich eingesetzt [Slama 2006]. Bei größeren Änderungen der Anforderungen muss jedoch das Architekturmodell aufwendig durch den Architekten neu generiert werden. Dadurch ist der MDA-Ansatz in vielen Projekten nicht praktikabel. Dies gilt auch für Empirieprojekt Alpha, bei dem sich die Anforderungen an die Anwendung und damit auch an die Architektur während der Entwicklung iterativ ergaben und daher recht häufig änderten.

Im Empirieprojekt Alpha wurde den Entwicklern das Recht eingeräumt, architektonische Änderungen eigenständig durchzuführen. Dadurch entwickelte sich die Architektur aufgrund konkreter Anforderungen der Implementierung weiter. Um die Entwickler nicht zu überfordern, wies die Architektur eine Reihe von Hotspots[3] auf, an denen Erweiterungen vorgenommen werden konnten.

Die Entwickler konnten eine Reihe von Änderungen an der Architektur eigenständig durchführen:

- Neue Subsysteme hinzufügen oder nicht mehr benutzte löschen.
- Zu groß gewordene Subsysteme in mehrere kleinere aufteilen oder mehrere Subsysteme zu einem größeren zusammenfassen.

[3] Als Hotspot bezeichnet man eine Stelle, die einen Ansatzpunkt für die Weiterentwicklung bietet. Im Idealfall muss so der bestehende Quelltext bei Erweiterungen nicht verändert werden [Pree 1995].

- Neue Schichten einführen oder mehrere Schichten zu einer zusammenfassen.
- Die Zuordnung von Subsystemen zu Schichten ändern.
- Beziehungen zwischen Elementen erlauben, die vorher verboten waren, oder Beziehungen verbieten, die vorher erlaubt waren.

Durch die verteilte Entwicklung und der dadurch erschwerten Kommunikation wäre es sehr zeitaufwendig gewesen, wenn der Softwarearchitekt alle diese Änderungen vor ihrer Umsetzung abgenommen hätte. Daher konnten Onshore- und Offshore-Entwickler Änderungen an der Architektur ohne vorherige Rücksprache mit dem Softwarearchitekten umsetzen. Um eine sinnvolle Evolution der Architektur sicherzustellen, wurde die Architektur durch den Softwarearchitekten regelmäßig mit automatisierten Werkzeugen überprüft (vgl. Abschnitt 5.1.2). Er arbeitete die durch die Entwickler vorgenommenen Änderungen ein und korrigierte sie bei Bedarf. Die aktualisierte Architekturdokumentation wurde allen Entwicklern zur Verfügung gestellt. Im Projekt kam es selten vor, dass der Softwarearchitekt architektonische Änderungen der Entwickler zurücknehmen musste.

5.4.3 Qualitätssicherung und Tests

Der Qualitätssicherung sollte in global verteilten Entwicklungsprojekten besondere Aufmerksamkeit gewidmet werden. Eine konsistente, klar kommunizierte Anwendungsarchitektur und Programmierrichtlinien sollten möglichst automatisch kontinuierlich überprüft werden. Für eine gute Qualität ist auch eine ausreichende Absicherung mit Komponententests wesentlich. Vorhandene Forschungsarbeiten in diesem Bereich empfehlen eine testgetriebene Entwicklung gerade für Offshoring-Projekte und globale Softwareentwicklung [Sengupta et al. 2004].

Martin Fowler beschreibt in [Fowler 2006b] die Verwendung von Akzeptanztests, um Anforderungen vor der Implementierung zu klären. Das Onshore-Team schreibt dabei die Tests in Prosaform. Das Offshore-Team setzt sie in automatisch oder manuell ausführbare Testskripte um. Dadurch wird die Interpretation der Anforderungen überprüfbar. Als Hauptproblem dieser Vorgehensweise beschreibt Fowler, dass das Offshore-Team motiviert werden muss, die Testskripte zu schreiben.

Im Empirieprojekt Alpha wurde versucht, Komponententests von den Offshore-Entwicklern einzufordern. Dort ergaben sich Schwierigkeiten durch das ausgeprägte Hierarchie-Bewusstsein und die stark arbeitsteilige Arbeitsweise des Offshore-Teams. So wurde z. B. gleichzeitig mit einer neu implementierten Funktion ein Fehlerbericht für ebendiese Funktion mitgeschickt. In einem anderen Beispiel wies die Implementierung von Rückzahlungsraten bei einer Rate einen durch Kopieren und Einfügen entstandenen Fehler auf. Der Test zu genau dieser Rate war auskommentiert. Die Fehlerbehebung wäre in beiden Fällen sehr einfach möglich gewesen. In einem agil arbeitenden Team hätte ein Entwickler den Fehler sofort behoben, aber anscheinend waren verschiedene

Entwickler mit den Tätigkeiten Entwicklung und Test betraut. Dies scheint typisch für Dienstleister mit höherem CMMI Level zu sein, vgl. Abschnitt 3.2.4.

Zwischen den Teams war eine Abdeckung von 85% des gesamten Quelltextes mit Komponententests informell vereinbart. Dieser Wert wurde vom Offshore-Team bei Weitem nicht erreicht. Der folgende Ausschnitt aus einem E-Mail-Verkehr zeigt die unterschiedlichen Auffassungen der Teams bezüglich Komponententests (das vollständige Review ist im Anhang B.2 zu finden):

Onshore-Softwarearchitekt: "Not every attribute in Lieferkonditionen is initialised, the test Lieferkonditionen_Test is insufficient."

Offshore-Projektleiter: "Can you please clarify what else is expected in the Test. According to us, the test will become too large if we test for all the fields."

Onshore-Softwarearchitekt: "Check for correct initialisation of all values after calling the constructor. Then check for correct setting of values by calling and testing the getters after the setters."

Auf Druck des Onshore-Teams konnten die Testabdeckung erhöht und einige agile Praktiken zu einem gewissen Teil umgesetzt werden, z. B. Programmierstandards, Refactorings, kurze Releasezyklen und einfaches Design.

Für zukünftige Offshoring-Projekte sollten nach diesen Erfahrungen weitere Vorgehensweisen umgesetzt werden, die in der Feldstudie nicht mehr untersucht werden könnten:

- Ostroff et al. schlagen eine Kombination von testgetriebener Entwicklung und dem Vertragsmodell vor. Mit beiden Techniken kann die Qualität des Quelltextes erhöht werden. Sie sollten komplementär verwendet werden [Ostroff et al. 2004]. Damit werden Schnittstellen verständlicher und die Fehlerzahl kann reduziert werden.

- Durch moderne Versionsverwaltungssysteme mit Kontrollfunktionen, welche z. B. die Testabdeckung messen und ungenügende Quelltexte zurückweisen, kann sichergestellt werden, dass sich immer ein lauffähiger Stand im Archiv befindet [Duvall et al. 2007]. So müssen viele Fehlertypen nicht mühsam gesucht werden.

- Eine weitere Möglichkeit besteht darin, Akzeptanztests durch die Anwender schreiben zu lassen. Dafür könnte das FIT-Rahmenwerk genutzt werden [Melnik et al. 2004]. FIT steht für „Framework for Integrated Test". Mit FIT können die Akzeptanztests automatisiert werden. Es ist zu untersuchen, inwieweit diese Vorgehensweise einsetzbar ist, wenn Anwender und Entwickler aus anderen Kulturkreisen stammen und nicht dieselbe Sprache sprechen.

5.5 Zusammenfassung

In diesem Kapitel wurde ein Ansatz auf Basis der architekturzentrierten Entwicklung entwickelt, mit dem die Zusammenarbeit zwischen verteilten Teams in globalen Entwicklungsprojekten gestärkt wird.

Für diesen Ansatz wurde das historische Verständnis von architekturzentrierter Entwicklung, das seine Wurzeln im RUP/UP und am SEI hat, durch Einbeziehung von Konzepten und Methoden aus STEPS und WAM erweitert. Eine Analyse der vorhandenen wissenschaftlichen Erkenntnisse hat ergeben, dass architekturzentriertes Arbeiten in global verteilten Entwicklungsprojekten noch nicht in ausreichendem Maße und zu wenig unter Einbeziehung von empirischen Erfahrungen aus professionellen Softwareentwicklungsprojekten erforscht wurde.

Das architekturzentrierte Vorgehen wurde daher in ein professionelles Offshoring-Entwicklungsprojekt nach der Forschungsmethode Action Research eingebracht und verfeinert. Dort hat sich gezeigt, dass ein Projekterfolg durch flexibles Reagieren auf die sich stellenden Probleme erreichbar ist. So wurde im betrachteten Empirieprojekt mit den „Component Tasks“ ein besseres Mittel zur Anforderungsweitergabe gefunden, als sich herausstellte, dass die aus der Methode Extreme Programming stammenden Story-Cards für das Offshore-Team nicht einfach umsetzbar waren.

In der Analyse konnte auch begründet werden, wie architekturzentrierte Softwareentwicklung und unterstützende agile Techniken geholfen haben, Probleme bei der Projektsteuerung, der Verteilung von Aufgaben, der Qualitätssicherung und anderen Aufgaben der Softwareentwicklung besser zu beherrschen. Die Auswertung der Feldstudie hat zu wichtigen Problemfeldern der architekturzentrierten Entwicklung bei global verteilten Entwicklungsprojekten geführt und Bezüge zu den Aufgaben eines Architekten aufgezeigt. Für viele Aspekte des Umgangs mit Softwarearchitektur konnten in diesem Kapitel Empfehlungen erarbeitet und begründet werden.

Im nächsten Kapitel werden die zentralen Ergebnisse dieses Kapitels in Musterform dokumentiert und in zwei weiteren Feldstudien evaluiert.

Teil III

Evaluierung der Ergebnisse

6 Überprüfung in der Praxis

Dieses Kapitel beschäftigt sich damit, die gewonnenen Erkenntnisse in weiteren Forschungsprojekten zu evaluieren und sie für den praktischen Einsatz nutzbar zu machen. Dazu werden die Erkenntnisse in Musterform dokumentiert. Muster sind in der Softwaretechnik seit mehreren Jahren etabliert und haben sich in vielen Bereichen bewährt, um Wissen festzuhalten und weiterzugeben.

Im nächsten Abschnitt werden auf Basis der in den letzten beiden Kapiteln erarbeiteten Ergebnisse Muster für die global verteilte Entwicklungsarbeit vorgestellt. Die neuen Muster werden im darauf folgenden Abschnitt auf zwei weitere Feldstudien angewendet. Diese Feldstudien basieren auf Interviews mit Projektbeteiligten und der Analyse von Projektdokumenten. Die in beiden Feldstudien gewonnenen Erkenntnisse werden anschließend einer zusammenfassenden Auswertung unterzogen.

Mit diesem Vorgehen werden die gewonnenen Erkenntnisse abgesichert und mit weiterer Empirie konsolidiert.

6.1 Muster für global verteilte Softwareentwicklung

Mitte der 90er Jahre wurde das erste Buch über Entwurfsmuster veröffentlicht, das mittlerweile weit verbreitet ist. Seitdem sind Muster in der Softwaretechnik als Dokumentationsform fest etabliert.

Im nächsten Abschnitt wird der Einsatz von Mustern in der Softwaretechnik beschrieben. Im darauf folgenden Abschnitt werden wichtige Muster für agile global verteilte Entwicklungsarbeit zusammengefasst, bevor daran anschließend wichtige Erkenntnisse dieser Arbeit als Muster dokumentiert werden.

6.1.1 Der Musterbegriff in der Softwaretechnik

In der Informatik, insbesondere in der Softwaretechnik, werden mit Mustern Lösungsschablonen für relevante, wiederkehrende Probleme beschrieben. Dabei handelt es sich um in der Praxis bewährte Lösungen, nicht um neue Ideen. Mit Mustern können gewonnene Erfahrungen in einheitlicher Form dokumentiert und damit in vergleichbaren Kontexten anwendbar gemacht werden. Davon profitieren vor allem der Wissenstransfer und die Softwaretechnik-Ausbildung.

In der Softwareentwicklung wurden als erstes Entwurfsmuster bekannt. Im Jahr 1995 wurde das Buch „Design Patterns“ [Gamma et al. 1995] veröffentlicht, das mittlerweile zu einem Standardwerk zu objektorientierter Softwareentwicklung geworden ist. Es beruht hauptsächlich auf der Doktorarbeit von Erich Gamma, der Arbeiten von Christopher Alexander et al. zu Mustern zur Architektur von Gebäuden, Konstruktionen und Städten [Alexander et al. 1977] auf die Informatik übertragen hat.

Der grundlegende Aufbau, der sich in allen Arten von Mustern findet, besteht aus der Beschreibung eines Problems und daran anschließend der Beschreibung einer Lösung. Der vollständige Aufbau enthält in der Regel zumindest die Punkte „Name“, „Problemstellung“, „Kontext“, „Kräfte“ und „Lösung“, die im Folgenden kurz charakterisiert werden:

Name: Sollte treffend gewählt werden, um das entsprechende Muster klar zu benennen.

Problemstellung: Die Frage oder Aufgabe eines Musters.

Kontext: Das Umfeld, in dem das Muster anwendbar ist. Ermöglicht die Prüfung, ob das Muster in einem bestimmten Fall eingesetzt werden kann, und bestimmt Anforderungen an die Lösung.

Kräfte: Die verschiedenen Einflüsse auf die Problemstellung. Kräfte können in unterschiedliche Richtungen wirken und unter Umständen in einem Konflikt untereinander stehen. Dann stellt die Lösung einen Kompromiss zwischen diesen Kräften dar.

Lösung: Diskutiert die Lösungsstruktur für das Problem in generischer Form im Hinblick auf den Kontext und die darin wirkenden Kräfte.

Muster können auf verschiedenen Abstraktionsebenen angesiedelt sein. Es gibt Muster, die allgemeine Konzepte beschreiben, und Muster für Implementierungsdetails. Muster zu ähnlichen Themenbereichen sollten in Musterkatalogen zusammengefasst werden, in denen sie nach bestimmten Kriterien erfasst und in derselben Form beschrieben werden. Es ist in der Regel möglich und sogar erwünscht, verschiedene Muster für einen Einsatzzweck zu kombinieren.

6.1.2 Muster für agile global verteilte Entwicklungsarbeit

Existierende Muster

In den letzten Jahren wurden viele Erkenntnisse der Softwaretechnik in Musterform dokumentiert. In diesem Abschnitt werden wichtige existierende Muster untersucht, die in agilen global verteilten Entwicklungsprojekten anwendbar sind. Dies sind Entwurfsmuster, Architekturmuster und Muster für agile Entwicklung.

Entwurfsmuster: Entwurfsmuster wurden oben schon als erste weitverbreitete Musterart in der Softwareentwicklung eingeführt. Neben dem „Design Patterns“-Buch [Gamma et al. 1995] wurden weitere Sammlungen von Entwurfsmustern bekannt, auch wenn sie nicht dessen Verbreitungsgrad erreichten. Erwähnenswert sind beispielsweise die Arbeiten von Wolfgang Pree, der u. a. Meta-Pattern identifiziert und beschrieben hat, die den Entwurfsmustern zugrunde liegen [Pree 1995].

Entwurfsmuster beschäftigen sich mit häufig anzutreffenden Entwurfsproblemen auf der Ebene von mehreren Objekten und ihren Beziehungen untereinander. Entwurfsmuster sollten heutzutage zum Handwerkszeug jedes guten Entwicklers gehören. Dass dem nicht immer so ist, hat beispielsweise das Empirieprojekt Alpha gezeigt, in dem die Offshore-Entwickler auch im Umgang mit grundlegenden Entwurfsmustern nicht vertraut waren. Daher sollte bei global verteilter Entwicklung immer geprüft werden, auf welche Wissensbasis aufgesetzt werden kann.

Architekturmuster: Architekturmuster beschreiben die grundlegende Organisation und Interaktion zwischen den Komponenten einer Anwendung. Damit befinden sie sich auf einer höheren Ebene als die Entwurfsmuster. Viele Konzepte, die in Kapitel 5 dieser Arbeit beschrieben sind, wurden von diversen Autoren als Architekturmuster formuliert, z. B. verschiedene Architekturstile wie die Schichtenarchitektur. Eine recht weite Verbreitung haben die Architekturmuster aus der Buchreihe „Pattern-Oriented Software Architecture“ von Buschmann et al. gefunden, die bereits 1996 gestartet wurde [Buschmann et al. 1996].

Architekturmuster können bei global verteilter Softwareentwicklung gut eingesetzt werden, um die Grundzüge der Architektur des zu entwickelnden Systems festzulegen.

Muster für agile Entwicklung: Recht bekannt geworden sind die Organisationsmuster von Coplien und Harrison für agile Softwareentwicklung [Coplien u. Harrison 2004]. Mit diesen Organisationsmustern werden Strukturen oder Vereinbarungen innerhalb von Unternehmen oder Projekten beschrieben, die für agile Softwareentwicklung förderlich sind.

In dieser Arbeit wurden die Organisationsmuster schon an einigen Stellen eingebracht: „Organization Follows Location“ in Abschnitt 3.4.2, „Architect Controls Product“ in Abschnitt 5.3.2 und „Standards Linking Locations“ in Abschnitt 5.3.5. Sie eignen sich sehr gut, um die Vorzüge von agiler Entwicklung effektiv nutzen zu können.

Ein Ansatz für Muster für agile verteilte Entwicklungsarbeit findet sich bei Braithwaite und Joyce. Sie haben im Jahr 2005 in zwei thematisch eng verwandten Publikationen eine Reihe von Mustern zusammengestellt, die sie in den Projekten ihres global aktiven Arbeitgebers identifiziert haben.

In einem Beitrag zur EuroPLoP, einer bekannten europäischen Konferenz für Muster in diversen Bereichen der Softwareentwicklung, sind sechs Muster enthalten [Braithwaite u. Joyce 2005b]:

- Ambassador,
- Kickoff,
- Multiple Communication Modes,
- Remote Pair,
- Virtual Shared Location,
- Visits Build Trust.

Die Namen der Muster sind gut gewählt, sodass sie auch ohne genaue Erläuterung mit etwas Hintergrundwissen verständlich sind. Die dahinter steckenden Ideen wurden auch in dieser Arbeit schon diskutiert, vgl. Kapitel 3. In der zweiten Publikation auf der XP 2005 finden sich diese Muster noch einmal [Braithwaite u. Joyce 2005a]. „Virtual Shared Location“ wird zu „Wiki as Shared Location“; ansonsten stimmen die Namen überein. Zusätzlich sind acht weitere Muster beschrieben:

- Balanced Sites,
- Code is Communication,
- Distributed Standup,
- Functional Tests Capture Requirements,
- One Team,
- One Team, One Build,
- One Team, One Codebase,
- Tests Announce Intention.

Braithwaite und Joyce bezeichnen alle enthaltenen Muster als Muster*kandidaten* oder Praktiken. Leider wurde dieser erste Musterkatalog bisher nicht von anderen Autoren bestätigt oder um weitere Muster ergänzt.

Beschreibung der Erkenntnisse als Muster

Die wesentlichen Erkenntnisse der vorangegangenen Kapitel werden im Folgenden als Muster beschrieben. Die Musterform bietet sich dafür aus zwei Gründen an:

- Mit Mustern lassen sich Erfahrungen, Anleitungen und Empfehlungen verbinden. Sie sind damit gut geeignet für die Dokumentation von Ergebnissen aus qualitativer Forschung.
- Mit Mustern lassen sich wissenschaftliche Ergebnisse für die weitere Forschung dokumentieren und in die Praxis übertragen, denn sie sind sowohl offen als auch geschlossen. Offen in dem Sinne, dass sie von anderen Autoren aufgegriffen, diskutiert und verändert werden können. Muster können zu Musterkatalogen zusammengefasst werden. Geschlossen in dem Sinne, dass Praktiker sie sofort an ihre Zwecke anpassen und einsetzen können.

Aus den Erkenntnissen dieser Arbeit wurden drei Muster abgeleitet. Sie werden hier kurz und prägnant in der oben angegebenen Form beschrieben. Details zu den jeweils dahinter stehenden Konzepten und Techniken werden hier nicht wiederholt. Sie lassen sich in den Kapiteln 4 und 5 dieser Arbeit nachlesen.

Dual-Shore-Modell

Problemstellung: Bei globaler Verteilung sind die Kommunikationskanäle zwischen verteilten Entwicklerteams oft schmal, was die Kooperation der Teams beeinträchtigt. Wie kann trotzdem eine gute, agile Zusammenarbeit der Teams erleichtert werden?

Kontext: Das Muster kann immer angewendet werden, wenn Teams global verteilt sind und sie nicht unabhängig voneinander arbeiten können. Erprobt wurde das Muster für die Zusammenarbeit von einem Onshore- und einem Offshore-Team beim Agilen Offshoring. Es sollte jedoch auch für ähnliche Konstellationen geeignet sein (vgl. Abschnitt 4.4.6).

Kräfte: Die wesentlichen Probleme der global verteilten Zusammenarbeit ergeben sich durch die geografische, zeitliche und organisatorische Verteilung der Akteure. Die Akteure haben unterschiedliche Rollen inne und verteilen sich auf die Teams. Ihnen stehen verschiedene Kommunikationsmedien und -mittel zur Verfügung.

Lösung: Beim Dual-Shore-Modell unterstützt ein externer Dienstleister den Kunden und seinen Offshoring-Dienstleister. Zusätzlich zum Personal des externen Dienstleisters arbeiten auch Entwickler des Offshoring-Dienstleisters für eine gewisse Zeit onshore. Optional arbeitet auch Onshore-Personal temporär offshore. Mit diesem Modell werden die Kommunikation zwischen den Teams und das gegenseitige Verständnis erleichtert. Die Zusammenarbeit der Teams wird verbessert.

Detaillierte Diskussion: Siehe Abschnitt 4.4.

Architekturzentrierte Softwareentwicklung

Problemstellung: Bei global verteilter Entwicklung kommt es oft zu Missverständnissen und Problemen zwischen den Entwicklerteams. Mehrfachentwicklungen, inkompatible Implementierungen und Qualitätseinbußen sind mögliche Folgen. Wie kann die agile Entwicklung mit verteilten Teams unterstützt werden, damit diese Effekte nicht oder seltener auftreten?

Kontext: Das Muster ist immer dann anwendbar, wenn an verschiedenen Standorten an einem gemeinsamen Produkt entwickelt wird.

Kräfte: Bei global verteilter Entwicklung arbeiten Entwickler aus unterschiedlichen Kulturkreisen an einem gemeinsamen Produkt. Sie verfügen über unterschiedliche Ausbildungen und haben oft andere Wertvorstellungen und persönliche Ziele. Bei der Entwicklung im Offshoring sind verschieden Organisationen beteiligt, die ebenfalls andere Kulturen und Vorgehensweisen haben können.

Lösung: Mit architekturzentrierter Softwareentwicklung steht die Softwarearchitektur im Mittelpunkt aller Aktivitäten. Die Fokussierung auf ein gemeinsames Artefakt stellt u. a. eine einheitliche Sprache zur Verfügung, erleichtert eine Aufteilung in Komponenten, die parallel implementiert werden können, ermöglicht eine übergreifende Steuerung der Teams und eine sinnvolle Qualitätssicherung und führt somit zu einer Beherrschung der Komplexität von global verteilter Entwicklung. Softwarearchitekten unterstützen die global verteilte Arbeit, indem sie 1. fachliches und technisches Wissen verbinden und für einen Wissenstransfer sorgen, 2. Vorgaben aufstellen können, welche die Entwicklung steuern und trotzdem die Eigenverantwortung der Entwickler stärken, und 3. zwischen den Teams vermitteln und moderieren können.

Detaillierte Diskussion: Siehe Kapitel 5.

Component Tasks

Problemstellung: Bei agiler global verteilter Entwicklung bestehen häufig Schwierigkeiten bei der Dokumentation von Anforderungen für entfernt arbeitende Teams und bei der Integration der von diesen Teams implementierten Teile aufgrund missverstandener Anforderungen oder schlechter innerer Qualität. Wie können diese Probleme angegangen werden?

Kontext: Das Muster ist immer dann anwendbar, wenn an verschiedenen Standorten an einem gemeinsamen Produkt entwickelt wird und die Anforderungen für ein Team von einem anderen Team erstellt werden. Architekturzentrierte Entwicklung (siehe vorhergehendes Muster) oder zumindest eine ausgeprägte und immer aktuell gehaltene Softwarearchitektur ist ebenso eine Voraussetzung für den Einsatz des Musters.

Kräfte: Neben der Heterogenität der Entwickler (siehe Kräfte beim vorhergehenden Muster) spielen hier unterschiedliche Aufgaben eine Rolle. Die Erhebung von Anforderungen in enger Zusammenarbeit mit dem Kunden ist bei verteilter Entwicklung in der Regel getrennt von der Implementierung der Anforderung durch offshore arbeitende Teams.

Lösung: Component Tasks sind um Kontext- und Detailinformationen angereicherte Story-Cards. Das können z. B. eine ausführlichere Beschreibung der Funktionen oder eine Menge von funktionalen Tests sein. Die Besonderheit von Component Tasks besteht in ihrer Beziehung zur Softwarearchitektur. Die Anforderungen werden nicht nach fachlichen Aspekten, sondern nach Komponenten der Architektur geschnitten. Dadurch verfügen die zu entwickelnden Teile über klare Schnittstellen. Das Entwicklerteam hat weitgehende Freiheiten bei der Implementierung der Komponente, solange die Schnittstelle eingehalten wird.

Detaillierte Diskussion: Siehe Abschnitt 5.3.6.

Streng genommen handelt es sich bei den hier vorgestellten Mustern um Muster*kandidaten*. Erst, wenn sie von anderen Autoren in der Praxis überprüft wurden, werden sie zu vollwertigen Mustern. Die Muster „Dual-Shore-Modell“ und „Architekturzentrierte Softwareentwicklung“ beschreiben grundlegende Organisationsformen und Vorgehensweisen. Das Muster „Component Tasks“ ist auf einer anderen Ebene angesiedelt. Es beschreibt ein spezifisches Artefakt und den Umgang damit.

Die Muster sind thematisch am ehesten mit denen von Braithwaite und Joyce vergleichbar, siehe oben. Sie sollten gut mit diesen kombinierbar sein.

6.2 Empirische Evaluierung der Muster

Mit zwei Feldstudien sollen die Ergebnisse der letzten Kapitel und die daraus extrahierten Muster in weiteren Empirieprojekten überprüft werden.

6.2.1 Zielsetzung und Methodik

Im Folgenden werden zwei größere Empirieprojekte vorgestellt. Wie schon im letzten Kapitel beim Empirieprojekt Alpha wurden auch diese Empirieprojekte mit den Mechanismen einer Feldstudie ausgewertet. Bei den hier beschriebenen Empirieprojekten erfolgte keine direkte Einflussnahme auf das Projekt. Es kam aber möglicherweise zu indirekten Rückkopplungen, indem Projektbeteiligte aus den durchgeführten Untersuchungen Schlüsse zogen und ihr Verhalten im Projekt änderten.

Es wird jeweils betrachtet, ob die gefundenen Muster einsetzbar wären. Dazu werden jeweils der Kontext und die zu erwartenden positiven Auswirkungen betrachtet. Um

die Validität der Ergebnisse sicherzustellen, schließt sich eine feldstudienübergreifende Analyse (Cross Case Analysis) an.

Als wesentliches Mittel der Analyse wurden Interviews eingesetzt (siehe Abschnitt 1.3.3). Alle Interviews wurden durch den Autor als direkte (d. h. von Angesicht zu Angesicht) Einzelinterviews durchgeführt. Die Interviewpartner wurden von den jeweiligen Projektverantwortlichen bestimmt. Allen Beteiligten wurde Anonymität zugesichert. Ein ausgewählter Interviewleitfaden und Beispielfragen sind im Anhang zu finden.

Neben der Auswertung der Interviews wird Material aus den Projekten untersucht, das von den jeweiligen Projektbeteiligten zur Verfügung gestellt wurde: diverse Projektdokumente, E-Mails, Quelltexte u. Ä.

6.2.2 Auswahl und Beschreibung der Feldstudien

Die Auswahl der Empirieprojekte für die Feldstudien erfolgte hauptsächlich anhand von zwei Kriterien:

1. *Nach der Art der Projekte:* Bei beiden Projekten handelte es sich um kleine bis mittelgroße, kommerzielle Projekte zur Entwicklung interaktiver Anwendungssoftware. Agile Methoden wurden in beiden Projekten mehr oder weniger ausgeprägt eingesetzt. Ein Großteil der Entwicklung fand offshore statt. Die Entwicklung erfolgte nicht explizit nach dem Dual-Shore-Modell. Einige Mitarbeiter arbeiteten für kürzere Zeitspannen beim entfernten Team. Diese Form der Zusammenarbeit hätte jedoch in beiden Projekten deutlich ausgebaut werden können. Architekturzentrierte Softwareentwicklung wurde nicht explizit praktiziert. Jedoch arbeiteten in beiden Projekten zumindest zeitweise Softwarearchitekten und es wurde eine Soll-Architektur definiert.

2. *Nach der Möglichkeit, einen guten Einblick in die Projekte zu gewinnen:* Bei kommerziellen Projekten ist es für Forscher in der Regel nicht einfach, aussagekräftige empirische Daten zu erheben. Dies gilt aus offensichtlichen Gründen insbesondere für global verteilte Projekte. Daher wurden Projekte ausgewählt, bei denen persönliche, offene Interviews mit den Beteiligten und ein möglichst wenig eingeschränkter Zugang zu Entwicklungs- und Prozessdokumenten möglich waren. Die Ergebnisse dürfen frei veröffentlicht werden, wenn auch in anonymisierter Form.

In beiden Fällen handelte es sich um Nearshore-Projekte (Onshore-Standort ist jeweils Deutschland, Offshore-Standort Irland bzw. Russland). Dies ermöglichte in einem Fall die Befragung von Mitgliedern des Offshore-Teams. Vergleiche die Diskussion des Begriffs Nearshoring auf Seite 19. Beide Projekte stammten aus demselben Konzern. Sie wurden jedoch mit verschiedenen Teams unter deutlich unterschiedlichen Bedingungen durchgeführt.

Die Beschreibung der beiden Feldstudien folgt einem einheitlichen Muster. Im Abschnitt **Kurzvorstellung des Projekts** werden wichtige Daten der Projekte genannt. Dies ermöglicht einen schnellen Überblick über das jeweilige Projekt. Besonderen Wert wird auf eine Darstellung der eingesetzten agilen Methoden und den Umgang mit Architektur gelegt. Auf die Kurzvorstellung folgt eine **Einordnung in die Übersichtskarte des IT-Sourcings**.

Der **Beschreibung der Untersuchung** ist zu entnehmen, wie die oben diskutierte Methodik der Untersuchung auf die jeweilige Feldstudie angewendet wurde. Hier wird beschrieben, welche Akteure interviewt wurden und welche Projektdokumente zur Auswertung zur Verfügung standen.

Der Gliederungspunkt **Vorfälle, Folgen und Auswertung** stellt den Hauptteil der Untersuchung dar. Exemplarische Vorfälle aus den Projekten, die einen Bezug zu den Mustern haben, werden mit Belegen (Zitate aus den Interviews), einer Beschreibung des Vorfalls und einer Reflexion unter besonderer Beachtung der Dimensionen der Verteilung und kultureller Aspekte diskutiert. Bei der Diskussion orientieren wir uns am Rahmenwerk von Wiredu für die Untersuchung von Koordination bei GSD [Wiredu 2006]. Das Rahmenwerk baut auf der klassischen „Leavitt-Raute“ [Leavitt 1965] auf. Diese wurde von Leavitt für die Untersuchung von Veränderungsprozessen in Organisationen konzipiert. Leavitt unterscheidet die Dimensionen Aufgaben, Akteure, Technik und Organisationsstruktur. Im Rahmenwerk von Wiredu wurden die vier Dimensionen an die verteilte Entwicklung angepasst, siehe Abbildung 6.1:

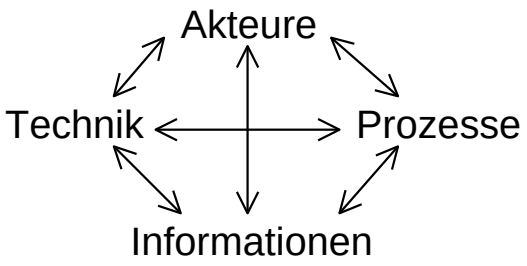


Abbildung 6.1: *Leavitt-Raute mit GSD-Dimensionen*

Akteure (engl. people): Die Dimension Akteure umfasst alle Personen, die an der Softwareentwicklung beteiligt sind, mit ihren Rollen, Fähigkeiten und Eigenschaften. In einer verteilten Umgebung werden die Akteure durch das kulturelle Umfeld ihres Arbeitsorts beeinflusst. Dies kann sich in ihrer Einstellung und ihrem Handeln ausdrücken.

Prozesse (processes): Alle Aufgaben, die Akteure bei der Softwareentwicklung wahrnehmen, sind Bestandteil der Dimension Prozesse. Dazu gehören u. a. auch Abstimmungen zwischen den Akteuren an unterschiedlichen, entfernten Arbeitsorten.

Informationen (information): Die Dimension Informationen berücksichtigt Informationen als zentralem Bestandteil der Softwareentwicklung. Informationen müssen in Form von Daten erhoben, verarbeitet, zwischen Akteuren ausgetauscht und in von allen Arbeitsorten zugänglichen Archiven gespeichert werden. Es muss sichergestellt werden, dass Informationen an allen Arbeitsorten gleich interpretiert werden.

Technik (technology): Die Technikdimension umfasst alle technischen Systeme, die bei der Entwicklung eingesetzt werden, u. a. Programmiersprachen, Entwicklungsumgebungen und Archive.

Alle Dimensionen beeinflussen sich gegenseitig. Für ein Gesamtverständnis ist daher die Betrachtung von Vorfällen aus allen Dimensionen nötig. Vgl. auch das Information Systems Research Framework von Hevner et al. [Hevner et al. 2004].

6.2.3 Das Empirieprojekt Beta

Kurzvorstellung des Projekts

Projektkontext: In diesem Projekt wurde ein neues Front-Office-System für den Verkauf von Produkten eines großen Transportunternehmens entwickelt. Hauptanforderungen waren Übersichtlichkeit und Performanz. Der Kunde befand sich in Deutschland. Das Onshore-Team mit Sitz in Hamburg war verantwortlich für die Anforderungsanalyse, das Projektmanagement und das Marketing. Es bestand aus dem Produktmanager, einem Analysten für die Anforderungsermittlung und für den Kontakt mit den späteren Anwendern, einem externen Tester, zwei Entwicklern, die von Zeit zu Zeit im Projekt eingesetzt waren, sowie – nach der Produktivsetzung – einem Helpdesk für die Unterstützung der Pilotanwender.

Entwickelt wurde hauptsächlich bei einem offshore arbeitenden Team in Dublin, Irland (hier und im Folgenden vereinfachend als Offshore-Team bezeichnet). Dieses gehörte zum Konzern des Kunden. Der Auslandsstandort hatte sich in einer internen Ausschreibung durchgesetzt. Das Team bestand aus der Entwicklungsmanagerin und – über die gesamte Projektlaufzeit gesehen – zwischen drei und sechs Entwicklern. Einige kleinere Entwicklungsarbeiten erledigte auch das Onshore-Team. Akzeptanztests wurden von einer Testerin in den USA durchgeführt. Das sich aus dieser Konstellation ergebende Kooperationsmodell des Projekts ist in Abbildung 6.2 dargestellt.

Das Projekt war durch sich oft ändernde Anforderungen geprägt, die innovative Lösungsansätze erforderten. Dies war hauptsächlich in der komplexen fachlichen Domäne begründet. Das Transportunternehmen hatte eine große Anzahl an Produkten mit teilweise komplizierten Buchungsschnittstellen und Abhängigkeiten im Angebot, die häufigen Änderungen unterlagen.

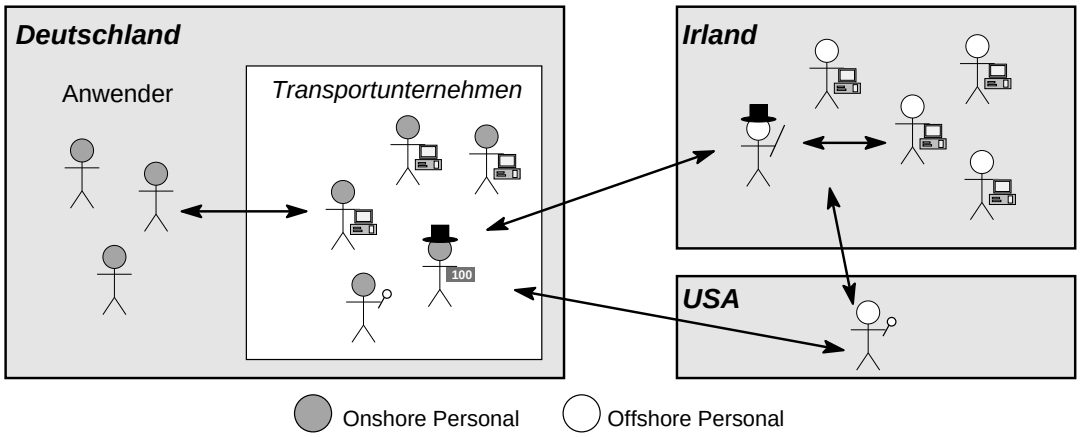


Abbildung 6.2: *Kooperationsmodell des Empirieprojekts Beta*

Im Laufe der Entwicklung kam es zu mehreren problematischen Situationen und Konflikten zwischen den Teams. Das Finanzbudget wurde deutlich überschritten. Nach mehreren Entwicklungsjahren wurde das Projekt schließlich gestoppt. Die Anwendung war so komplex und instabil geworden, dass neue Anforderungen nur mit großen Änderungen an der Anwendungsstruktur hätten umgesetzt werden können. Anschließend wurde die Anwendung auf Basis der gewonnenen technischen und fachlichen Erkenntnisse von einem in Deutschland an einem Standort arbeitenden Team von Grund auf neu entwickelt.

Zeitrahmen: Das Kickoff-Treffen des Projekts fand im August 2002 statt; Anfang 2003 wurde mit der Entwicklung begonnen. Im 3. Quartal 2005 wurde eine erste Version der Anwendung für etwa 60 Pilotkunden produktiv gestellt. Im April 2006 folgte die zweite Version. Im Dezember 2006 wurde das Projekt eingestellt. Mitte Januar 2007 wurde mit der kompletten Neuentwicklung mit einem nur onshore arbeitendem Team begonnen.

Die Interviews mit dem Onshore-Team fanden im Oktober und November 2006, die Interviews mit dem Offshore-Team Ende November 2006 statt.

Umsetzung agiler Methoden: In Bezug auf die Nutzung agiler Methoden im Projekt lassen sich zwei Phasen unterscheiden. In der ersten Phase wurde jede Anforderung vom Onshore-Team in einem umfangreichen Pflichtenheft beschrieben, von denen einige einen Umfang von bis zu dreißig Seiten hatten. Diese Dokumente enthielten genaue Vorgaben und Richtlinien für die Implementierung. Ihre Erstellung dauerte so lange, dass die Informationen schnell veralteten. Der Entwicklungsprozess wurde von den Beteiligten als unstrukturiert wahrgenommen. Es gab keine Iterationen von

vorgegebener Dauer; ausgeliefert wurde bei Fertigstellung. Der Projektfortschritt war schlecht messbar.

Diese Probleme wurden erkannt. Als Folge wurden ab März 2006 Praktiken aus dem von Ambler entwickelten Agile Unified Process (AUP, siehe [Ambler 2006]) eingeführt: Der Entwicklungsprozess wurde in zweiwöchige Iterationen unterteilt. Die Anforderungen wurden als Storys aufgeschrieben und im gesamten Team in Telefonkonferenzen diskutiert. Es wurden tägliche Standup-Meetings mittels Telefon etabliert, bei denen Neuigkeiten besprochen und auch Fragen geklärt wurden. Die gemessene Produktivität stieg in der zweiten Phase beträchtlich.

Softwarearchitektur: Das neu entwickelte System war in die bestehende, sehr komplexe, historisch gewachsene Systemlandschaft eingebettet. Es baute auf Infrastrukturkomponenten auf, die teilweise auch von anderen Systemen genutzt wurden. Die wichtigste dieser Komponenten war ein onshore von einem nicht direkt an dem Projekt beteiligten Team entwickeltes Mid-Office-System. Es wurde während der Projektlaufzeit weiterentwickelt, befand sich damit im Stadium „Erhalten der Architektur“ (siehe die Definition der Phasen in Abschnitt 5.2.2).

Im Projekt wurde ein Adapter neu entwickelt, der das Mid Office mit dem Buchungssystem des Transportunternehmens verbindet. Dessen Architektur wurde von Grund auf neu erstellt (Stadium „Entwerfen der Architektur“). Zu Beginn des Projekts arbeitete ein Softwarearchitekt im Offshore-Team. Aus Kostengründen wurde er aus dem Projekt abgezogen. Als ein Onshore-Analyst ins Team kam, führte er am Standort des Offshore-Teams eine Analyse der statischen Architektur durch. Dies blieb jedoch eine einmalige Aktion. Architekturzentrierte Entwicklung in der zu Beginn dieses Kapitels diskutierten Bedeutung wurde nicht eingesetzt.

In Anhang C.1 wird die statische Architektur des Systems grafisch dargestellt und erläutert.

Einordnung in die Übersichtskarte des IT-Sourcings

Abbildung 6.3 zeigt, wie das Projekt in die Übersichtskarte des IT-Sourcings einzuordnen ist.

Wie bei der Feldstudie im vorhergehenden Kapitel handelt es sich um Application Outsourcing, bei dem eine neue Anwendung entwickelt wurde *(Grad Geschäftsorientierung)*. Mit dem Onshore-Standort Deutschland und dem Offshore-Standort Irland fällt das Projekt in die Kategorie Nearshore Sourcing *(Standort)*. Da das Offshore-Team konzernintern ausgewählt wurde, spricht man von Captive Outsourcing *(Finanzielle Abhängigkeit)*. Das Sourcing ist selektiv. Teile der Anwendungsentwicklung (insbesondere das zugrunde liegende Rahmenwerk) verblieben im Kernunternehmen *(Grad externer Leistungsbezug)*. Da das Offshore-Team zum Konzern gehörte, ist die Leistungserstellung

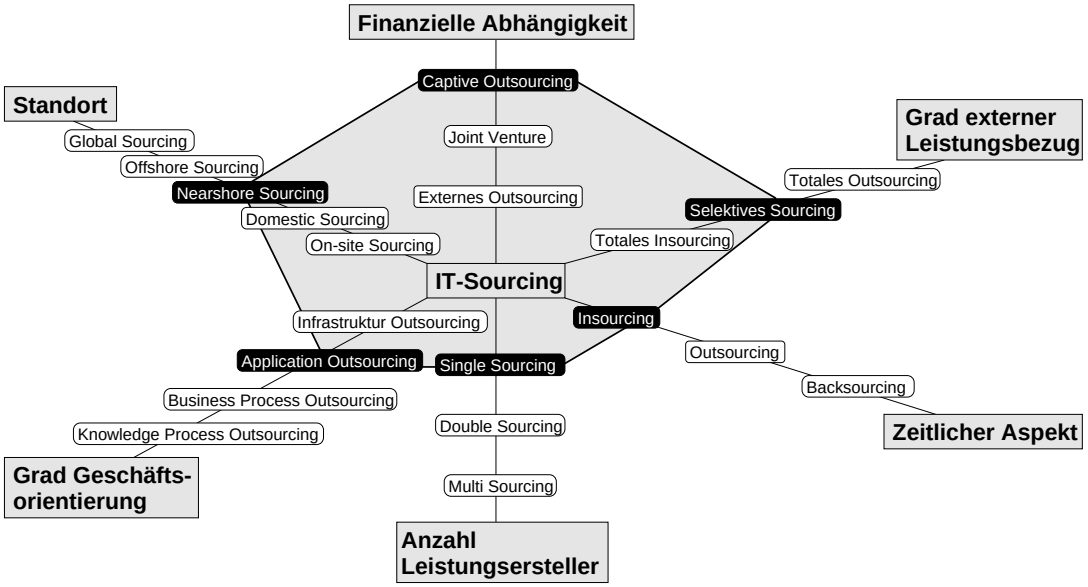


Abbildung 6.3: *Einordnung von Projekt Beta in die IT-Sourcing-Map*

formal als Insourcing zu bezeichnen. Praktisch handelte es sich jedoch aufgrund der Größe des Konzerns und der weitgehend unabhängigen Arbeit der Teams eher um Outsourcing *(Zeitlicher Aspekt)*. Auch wenn das Onshore-Team und die Akzeptanztesterin zu einem gewissen Teil an der Entwicklung beteiligt waren, gab es hauptsächlich einen Leistungsersteller, somit geht es um Single Sourcing *(Anzahl Leistungsersteller)*.

Beschreibung der Untersuchung

In diesem Projekt konnten die wichtigsten Akteure aus beiden Teams in Einzelinterviews befragt und zur Verfügung gestellte Materialien ausgewertet werden. Für die Interviews wurde jeweils ein Leitfaden für ein semistrukturiertes Gespräch erstellt. Als Akteure des Onshore-Teams wurden vom Autor der Projektmanager, der Analyst und ein Entwickler ausgewählt. Die Interviews wurden in deutscher Sprache an einem neutralen Ort (Entwickler) bzw. an den Arbeitsplätzen (Projektmanager, Analyst) durchgeführt. Daran anschließend folgten Interviews mit dem entfernten Team am Offshore-Standort in englischer Sprache. Für die Interviewleitfäden für das Offshore-Team wurden die Erkenntnisse der Interviews mit den Onshore-Akteuren berücksichtigt. Interviewt wurden die Entwicklungsmanagerin sowie zwei von ihr ausgewählte Teammitglieder: eine Entwicklerin und ein Entwickler. Die Interviews dauerten zwischen einer Dreiviertelstunde und zwei Stunden. Die Erkenntnisse aus den Interviews wurden mit dem Onshore-Team durchgesprochen.

Als Materialien aus dem Projekt wurden mehrere Präsentationen, eine detaillierte schriftliche AUP-Selbsteinschätzung des Offshore-Teams zur Nutzung agiler Methoden[1], ein Projektprotokoll des Onshore-Teams sowie eine Beschreibung der Softwarearchitektur für die Untersuchung zur Verfügung gestellt.

Vorfälle, Folgen und Auswertung

In den folgenden Abschnitten werden fünf Vorfälle diskutiert, die in der Auswertung der Interviews und des weiteren Materials als besonders wichtig und typisch für die verteilte Zusammenarbeit in diesem Projekt identifiziert wurden: 1. Trennung von Rahmenwerks- und Anwendungsentwicklung, 2. Innere Qualität und Softwarearchitektur, 3. Gemeinsame Arbeit am Quelltext, 4. Inhalt der Storys und 5. Fehlen von Komponententests.

Vorfall 1: Trennung von Rahmenwerks- und Anwendungsentwicklung

Die Anwendung bestand hauptsächlich aus dem Mid-Office-Rahmenwerk und einem Adapter. Das Rahmenwerk wurde onshore von einem externen Team weiterentwickelt, der Adapter im Projekt. Die Abstimmung zwischen beiden Entwicklungslinien funktionierte nicht gut. Es kam zu Verzögerungen und Missverständnissen.

Belege:

„Wenn da [im Mid Office] Änderungen eingeführt werden, müssen sie [das Offshore-Team] Anpassungen vornehmen. [...] Das ganze Mid-Office-Entwicklungsteam hat wöchentlich ein Meeting, bei dem sie sagen, was sich verändert hat, aber die Dubliner nehmen nicht dran teil. Sie hören nicht einmal per Telefon zu.“

Aus dem Interview mit dem Onshore-Entwickler

“Most of the mid office application server is being developed in Hamburg and we have to slot into that. The big issue for us was that we always had a very short window to integrate our application. [...] The mid office developers are having their meetings and their documentation in German. So I suppose being remote from Hamburg we wouldn't get all the updates. Out test server is only updated with the major releases. That could be an issue for us.”

Aus dem Interview mit dem Offshore-Entwickler

[1] Mit Fragebögen zur Selbsteinschätzung kann ein Entwicklungsteam feststellen, wie es agile Techniken nutzt und welche es ggf. noch besser umsetzen könnte. Beispiele für entsprechende Fragebögen finden sich in [Leffingwell 2007] und [Warden u. Shore 2007].

Beschreibung:

Um die Produkte des Transportunternehmens über die vorhandene Infrastruktur buchen zu können, wurde ein Adapter geschrieben. Diese Schnittstelle stellte zusammen mit der neuen Anwendung für Endbenutzer die Hauptleistung des hier beschriebenen Projekts dar. Der Adapter griff auf mehrere vorhandene Infrastrukturkomponenten zurück. Die wichtigste davon war ein Mid-Office-Rahmenwerk. Es wurde von einem Hamburger Team betreut und dort parallel zu diesem Projekt weiterentwickelt.

Bei der Erweiterung des Rahmenwerks ließen sich zwei Anlässe unterscheiden. Zum einen wurde es aufgrund von Anforderungen aus anderen Projekten weiterentwickelt. Diese Änderungen hatten teilweise Einfluss auf den Adapter, z. B. durch geänderte Schnittstellen. Zum anderen kamen Änderungswünsche vom Offshore-Team. Diese wurden an das Mid-Office-Team weitergeleitet und dort eingeplant. In regelmäßigen Abständen gab es neue Releases des Rahmenwerks.

Für das Projekt erwiesen sich insbesondere Änderungen der ersten Kategorie als problematisch, da sie nicht ausreichend mit dem Offshore-Team abgestimmt wurden. Für die deutschen Entwickler gab es ein wöchentliches Treffen in Hamburg, bei dem die Änderungen besprochen wurden. Das Offshore-Team wurde nicht beteiligt. Da das Rahmenwerk nur auf Deutsch dokumentiert wurde, wurde das Offshore-Team über viele Aspekte nicht oder nur ungenau informiert. Es gab zwar einen E-Mail-Austausch, doch wurden darin keine Details diskutiert, z. B. anzupassende Passagen im Quelltext.

Jedes neue Release des Rahmenwerks musste das Offshore-Team unter Zeitdruck integrieren. Da den Offshore-Mitarbeitern Details der Änderungen nicht bekannt waren, war das zeitaufwendig und stressig und eine potenzielle Fehlerquelle. Das Onshore-Team verstand nicht, warum sich die Offshore-Entwickler nicht aktiver um Informationen über die Releases bemühten. Sie sahen es jedoch auch nicht als ihre Aufgabe an, zu vermitteln.

Im späteren Projektverlauf besserte sich die Situation etwas, als die Rahmenwerksentwickler sensibler für die Nöte des Offshore-Teams wurden und Änderungen schneller per E-Mail oder Telefon kommunizierten. Insgesamt blieb die Situation aber unbefriedigend.

Reflexion:

Die Trennung von Rahmenwerksentwicklung und Adapterentwicklung bei gegenseitigen Abhängigkeiten zwischen den Komponenten schuf eine schwierige Situation. Die Akteure kannten sich nicht persönlich und konnten sich schlecht verständigen. Da das beschriebene Projekt nur einer von vielen Klienten des Rahmenwerks war, blieb die Dokumentation auf Deutsch und damit schwer verständlich für das Offshore-Team. Diesem Team fehlten wichtige Informationen über die Änderungen. Prozesse zum Austausch dieser Informationen waren nicht etabliert und das Offshore-Team kümmerte sich auch

nicht aktiv darum. Technische Lösungen des Problems waren schwierig, da die Sprache ein Haupthindernis darstellte, das auch Videokonferenzen o. Ä. entgegenstand.

Einsatzmöglichkeiten der Muster:

Mit dem Einsatz des Dual-Shore-Modells hätte ein Offshore-Entwickler in das Entwicklerteam des Rahmenwerks integriert werden können, um die Kommunikation zu unterstützen. Dies wäre aber mit beträchtlichen Kosten verbunden gewesen.

Die Vermittlung durch das Onshore-Team wäre eine weitere Lösungsmöglichkeit gewesen. Dieses sah sich nicht für die Kommunikation mit den Entwicklern des Mid-Office-Rahmenwerks verantwortlich, da das Offshore-Team technisch eigenständig arbeiten wollte. Wenn im Onshore-Team architekturzentrierte Entwicklung eingeführt worden wäre, hätte ein Softwarearchitekt das Offshore-Team bei den Treffen der Rahmenwerksentwickler vertreten können. Dadurch wären die Auswirkungen von Änderungen auf die Schnittstellen zwischen beiden Komponenten transparenter geworden. Die Mitsprachemöglichkeiten der Projektmitarbeiter bei der Rahmenwerksentwicklung wäre deutlich gestärkt worden.

Rahmenwerk-Releases hätten leichter integriert werden können, wenn das Rahmenwerk in der Architektur des Adapters besser gekapselt gewesen wäre (siehe auch Vorfall 2) oder die Funktionalität des Adapters mit Komponententests abgesichert worden wäre (siehe auch Vorfall 5).

Vorfall 2: Innere Qualität und Softwarearchitektur

Die Architekturevolution des Systems wurde nicht ausreichend gesteuert, sodass es nach einiger Zeit stark monolithische Züge und deutliche softwaretechnische Mängel aufwies. Diese Probleme führten zum Stopp des Projekts, nachdem sich neue Anforderungen nur mit großem Aufwand umsetzen ließen.

Belege:

„Das ist ein typisches Beispiel einer Fehlentwicklung. [...] Fachlich ist das System monolithisch, da gab es keine nennenswerte Strukturierung. [...] Es macht einfach keinen Sinn, das so zu bauen. Das ist viel zu kompliziert für das, was es tun soll. Das ist der blanke Overkill.“

Aus dem Interview mit dem Onshore-Analysten

“The architecture analysis was very, very interesting, but I'm not sure if it was actually that useful in bringing about change. Some things that the automated tool finds that aren't great are just necessary evils. But I think the comment were taken aboard and there might have been minor changes, but in general it wasn't something that changed our attitude.”

Aus dem Interview mit der Offshore-Entwicklerin

"We are missing an architect. We are missing a senior software person that can settle arguments, that can say how to implement this feature. The drawback is that the design can get a little more out of control when it is not so disciplined. [...] What we are missing actually are code reviews. So there is no overview from somebody saying we are doing something correctly or incorrectly."

Aus dem Interview mit dem Offshore-Entwickler

Beschreibung:

Als offshore noch ein Softwarearchitekt arbeitete, wurden die Grundlagen der Architektur der Anwendung gelegt. Unter Architektur wurde dabei hauptsächlich die Verteilungssicht (siehe Abschnitt 5.1.1) verstanden, da das Kundenunternehmen und das Projekt durch eine verteilte Infrastruktur mit mehreren Komponenten geprägt sind. Es wurde jedoch auch eine Vorstellung von der statischen Soll-Architektur entwickelt.

Nachdem der Architekt aus dem Projekt abgezogen worden war, um Kosten zu sparen, zeigten sich bald Probleme. Zum einen sahen es das Onshore- und das Offshore-Team als einen wesentlichen Fehler an, dass die Architektur nicht ausreichend dokumentiert worden war. Dadurch war die Einstiegshürde für neue Entwickler groß. Aus Unkenntnis über die ursprünglich vorgesehene Architektur wurde diese inkonsistent erweitert. Änderungen und Erweiterungen wurden an unpassenden Stellen vorgenommen und verschlechterten die Architektur.

Zum anderen war der Adapter sehr monolithisch aufgebaut. Die einmalig durchgeführte Architekturanalyse durch den Onshore-Analysten ergab eine ganze Reihe von Schwachstellen: sehr viele Klassen mit vielen zyklischen Beziehungen, unnötige Abstraktionen, sehr komplexe Implementierungen, auskommentierter oder unbenutzter Quelltext und praktisch keine Abdeckung mit Komponententests. Aus Sicht des Onshore-Analysten war das gewählte Konzept nicht nur schwer wart- und erweiterbar, sondern auch für zukünftige Projektphasen schlecht geeignet.

In den Interviews zeigte sich, dass die Probleme mit der inneren Qualität von einem Teil der Offshore-Entwickler gesehen wurden. Auch aus der AUP-Selbsteinschätzung geht hervor, dass dem Offshore-Team Schwachstellen wie sehr viel unbenutzter oder auskommentierter Quelltext und zu lange und mit Funktionalität überladene Methoden bekannt waren (siehe Anhang C.2). Sie konnten die Probleme mit der inneren Softwarequalität aber nicht selbst beheben. Gegenüber dem Onshore-Team wurden die softwaretechnischen Mängel verharmlost. Letztendlich wurden diese aber zum Hauptgrund für das Scheitern des Projekts, da Änderungen am Quelltext zu aufwendig wurden, das System immer monolithischer wurde und die Architektur nicht ausreichend auf neue Anforderungen ausgerichtet war.

Reflexion:

Dieser Vorfall macht deutlich, welche Folgen drohen, wenn die technische Weiterentwicklung einer Anwendung nicht ausreichend gesteuert wird. Einzelne Entwickler vermissten einen Architekten, der koordinierte und strukturierte. Sie hätten gerne Architektur- und Quelltextanalysen gehabt. Der Onshore-Analyst hätte aufgrund seiner Qualifikation diese Rolle ausüben können, wie auch die Architekturanalyse zeigte. Doch aus politischen Gründen war das nicht erwünscht.

Die Informationen über die Mängel waren also sowohl beim Onshore- als auch beim Offshore-Team vorhanden, doch wurden daraus nicht die richtigen Schlüsse gezogen. Die leitenden Akteure des Offshore-Teams sahen zwar, dass die Umsetzung der Anforderungen immer länger dauerte. Sie lehnten jedoch eine Einflussnahme des Onshore-Teams in technischen Fragen strikt ab. Da es keine weiteren Architekturanalysen gab, war der genaue Stand der inneren Qualität unbekannt. Die nötigen Prüfprozesse wurden nicht rechtzeitig etabliert.

Einsatzmöglichkeiten der Muster:

Im Projekt wurden zwei wesentliche Fehler im Umgang mit Architektur gemacht, die hätten vermieden werden können, wenn dem Thema eine größere Rolle zugekommen wäre. Zum einen hätten für die Architekturerstellung und -evolution mehr Ressourcen zur Verfügung stehen müssen. So hätte ein erfahrener Softwarearchitekt verhindern können, dass eine schwer wartbare, monolithische Struktur entstand. Hier hätte insbesondere eine Zerlegung der Anwendung in Subsysteme und Komponenten mit explizit definierten und regelmäßig überprüften Schnittstellen und Beziehungen geholfen. Diese Struktur hätte so ausführlich wie nötig in einer statischen Architektursicht dokumentiert und mit Architekturregeln u. Ä. abgesichert werden sollen.

Zum anderen hätte der Softwarearchitekt nicht abgezogen werden dürfen. Die sehr dynamische Produktpalette des Transportunternehmens hatte immer wieder Anpassungen der Architektur zur Folge. So wurde z. B. neue Druckhardware unterstützt, die ganz anders anzusteuern war. Bei diesen Anpassungen wäre eine Übersicht über die Gesamtarchitektur der Anwendungen die Grundlage für eine schlüssige und zukunftsfähige Weiterentwicklung des Systems gewesen. Aufgrund von fehlender Kenntnis der zukünftigen fachlichen Entwicklung konnte man offshore die Architekturevolution nicht sinnvoll gestalten. Der Onshore-Produktmanager konnte seine steuernde Rolle in der bestehenden Konstellation nicht erfüllen, da er nur Einfluss auf die Anforderungen, nicht jedoch auf die Tätigkeit der Offshore-Entwickler hatte. Man hätte jedoch Prüfungen der inneren Qualität zum Bestandteil der Abnahme neuer Releases machen können.

Vorfall 3: Gemeinsame Arbeit am Quelltext

Den Großteil der Anwendungsentwicklung übernahm das Offshore-Team. Onshore wurden kleinere Komponenten entwickelt. Bei der Integration dieser Komponenten gab

es häufig Probleme. Aus Sicht des Onshore-Teams verzögerten die Offshore-Entwickler die Aufnahme in die gemeinsame Quelltextbasis oder behinderten sie sogar.

Belege:

„Das ist wirklich wie ein Geschwür in deren Code, wie die unseren Code auffassen. Das ist unglaublich. [...] Wir haben so zwei Entwickler im Dublin-Team, die würden das [den vom Onshore-Team geschriebenen Quelltext] auch integrieren, ohne das an die große Glocke zu hängen, aber das ist unschön.“

Aus dem Interview mit dem Onshore-Analysten

“The distribution of the team made it more difficult for development than for working on a story or working on requirements. [...] He [the onshore developer] gave me a fix and it was up to me to integrate it into our codebase and that was definitely difficult. We were both out of sync. [...] Even today there are features that he has implemented that are not in our codebase. We have other tasks with higher priority.”

Aus dem Interview mit dem Offshore-Entwickler

Beschreibung:

Der Hauptteil der Entwicklungsarbeiten fand offshore statt. In unregelmäßigen Abständen wurden jedoch kleinere Teile vom Onshore-Team entwickelt. Da dieses keinen direkten Zugriff auf das Versionsverwaltungssystem hatte und auch nicht am Offshore-Team vorbei arbeiten wollte, wurden die Änderungen von Offshore-Entwicklern integriert. Hierbei kam es häufig zu Verzögerungen. Das Onshore-Team hatte zudem den Eindruck, dass die onshore entwickelten Funktionen vom Offshore-Team nicht richtig angenommen wurden. So wurden Weiterentwicklungen nicht wie eigentlich vorgesehen übernommen. Das Offshore-Team begründete die Verzögerungen meist damit, dass andere Aufgaben höhere Priorität hätten. Aus Sicht des Onshore-Teams wurden Integrationen häufig auch so lange verzögert, bis das Offshore-Team mit dem durch die Verzögerung entstandenen Zusatzaufwand für eine Neuentwicklung argumentierte.

In einem Fall wurde eine Komponente des Onshore-Teams nicht übernommen. Stattdessen implementierte das Offshore-Team die Komponente neu. Der Onshore-Entwickler hatte den Eindruck, dass dabei Quelltext verwendet wurde, den er selbst geschrieben hatte. Als Grund für die Neuimplementierung gab das Offshore-Team Performanzprobleme der onshore entwickelten Komponente an. Das Onshore-Team konnte das nicht nachvollziehen.

Bei der gemeinsamen Arbeit am Quelltext kam es immer wieder zu Konflikten zwischen den Teams, die sich nicht zufriedenstellend lösen ließen.

Reflexion:

In den Interviews bezeichneten beide Teams die Kommunikation untereinander als zufriedenstellend. Die Sprache oder kulturelle Unterschiede würden keine Hindernisse bei der Verständigung darstellen. Aus Sicht des Onshore-Analysten konnte das Offshore-Team mit Andeutungen von Kritik jedoch überhaupt nicht umgehen. Auch auf konstruktiv gemeinte Vorschläge gingen sie nicht ein und verhinderten dadurch eine Verbesserung der Situation.

Ein Blick auf die Vorgeschichte offenbart einen unterschwelligen unternehmenspolitischen Konflikt. Der Standort in Irland hatte das Projekt in einer internen Ausschreibung gewonnen. Man hatte eine Aufwandsschätzung abgegeben, die deutlich unter derjenigen der deutschen Konkurrenz lag. Der Auftrag wurde dann durch eine Managemententscheidung nach Irland vergeben. Das Onshore-Team sah die offshore geleistete Arbeit immer besonders kritisch, vor allem, als sich im Verlaufe des Projekts zeigte, dass die anfängliche Aufwandsschätzung deutlich zu niedrig war. Das Onshore-Team ging davon aus, dass es selber die Entwicklung schneller und mit weniger Fehlern geschafft hätte.

Wie schon in Abschnitt 2.2.1 diskutiert wurde, zieht Offshoring oft Auswirkungen auf Arbeitsplätze nach sich. Der Wettstreit um Aufträge führte in diesem Projekt dazu, dass das Offshore-Team verhindern wollte, dass sich das Onshore-Team stärker an der Entwicklung beteiligte. Aus diesem Grund ist es den Akteuren dieses Projekts schwer möglich gewesen, die Situation selber zu lösen. Es fehlte eine Person, die aus möglichst objektiver Sicht technische Entscheidungen hätte treffen können.

Einsatzmöglichkeiten der Muster:

Die Anforderungen an die Komponente hätten in einem Component Task beschrieben werden können. So wären die Beziehungen zu anderen Komponenten und die Anforderungen an die Performanz der Komponente deutlich geworden.

Ein Softwarearchitekt hätte eine vermittelnde Rolle ausfüllen können. Gegenüber einem externen Moderator, der sich auf die Konfliktlösung konzentriert, bringt ein Softwarearchitekt projektbezogenes und technisches Wissen mit. Er hätte technisch fundierte Entscheidungen für oder gegen entwickelte Komponenten treffen, Maßnahmen mit beiden Teams absprechen und dadurch die Konkurrenzsituation entschärfen können. Auf dieser Basis hätten die agilen Techniken „Fortlaufende Integration“ und „Gemeinsame Verantwortlichkeit“ Möglichkeiten geboten, das Onshore-Team direkt an der Entwicklung zu beteiligen.

Vorfall 4: Inhalt der Storys

Das Onshore-Team war unzufrieden mit der Umsetzung der Storys durch das Offshore-Team. Das Offshore-Team beklagte sich hingegen darüber, dass ihnen die Anforderungen zu eng vorgegeben würden und sie zu wenig Zeit für Refactorings und andere interne

Storys hätten. Dies war zu einem großen Teil darin begründet, dass Onshore- und Offshore-Teams unterschiedliche Auffassungen über die offshore vorhandene fachliche und technische Kompetenz hatten.

Belege:

„Sie [die Offshore-Entwickler] hatten vor meiner Zeit den Zugang zum Test-Client bekommen. Das war aber alles auf Deutsch und sie haben es nie hingekriegt, den Test-Client zu installieren oder zum Laufen zu bringen. [...] Und als ich dann endlich erkannt habe, wofür der Test-Client eigentlich gedacht war und was man damit machen sollte, war es für mich erstaunlich, dass das vorher nicht bekannt war und eingesetzt wurde, denn man hat hier wirklich blind versucht, irgendwas umzusetzen. [...] Ich habe eine Zeit lang E-Mails bekommen: Kannst Du mir das mal übersetzen. Und dann war das etwas Banales wie ‚Das System steht zurzeit nicht zur Verfügung'."

Aus dem Interview mit dem Onshore-Produktmanager

„Es klappt mal phasenweise und dann eskaliert wieder etwas und man hat den Eindruck, man ist eigentlich keinen Schritt weitergekommen."

„Das Dublin-Team liefert auch kein Feedback zwischendurch, z. B. ‚da gibt es technische Probleme'. Man sieht dann, was sie fertiggemacht haben und stellt dann fest ‚ok, damit können wir eigentlich nicht leben'. Wir leben auf einer permanenten Baustelle dort. Das ist katastrophal, würde ich sagen. [...] Der Abnahmeprozess funktioniert nicht."

Aus dem Interview mit dem Onshore-Analysten

"The story is a user story. It tells the description from a user's perspective. The implementation is behind the scenes."

"Refactoring is not as customer-facing. The product manager isn't seeing the value of refactoring as much. Everytime we are planning a refactoring story, we have to sell the business value to him, e. g. if we refactor this particular feature, it will be twice as fast."

Aus dem Interview mit der Offshore-Entwicklungsmanagerin

Beschreibung:

Wie schon in der Kurzvorstellung des Projekts beschrieben wurde, wurden bei der Einführung agiler Methoden die Pflichtenhefte durch Storys ersetzt. Diese Maßnahme führte zu einer deutlichen Verbesserung. Sie brachte jedoch auch Verstimmungen zwischen den Teams mit sich.

Der Onshore-Analyst hatte den Eindruck, dass die Offshore-Entwickler die Anforderungsdokumente nicht richtig verstanden, „gerne im Brain-Off-Modus“ arbeiteten, sich bei Fragen nicht meldeten und daher „teilweise unglaublich viel Mist“ implementierten. Als Folge dieser Einschätzung gab der Onshore-Produktmanager die Anforderungen in den Storys sehr detailliert und kleinschrittig vor. Zwar nicht mehr so ausführlich wie bei den Pflichtenheften, aber doch viel umfangreicher als ursprünglich vorgesehen. Oft enthielten die Storys auch konkrete Empfehlungen für die technische Umsetzung. Der Onshore-Analyst hielt eine selbstständige Abbildung der Anforderungen vom fachlichen auf das technische Modell durch die Offshore-Entwickler für erstrebenswert, doch traute er es ihnen bei komplexen Aufgaben nicht zu.

Das Onshore-Team vermutete einen Grund für das aus ihrer Sicht mangelnde fachliche Verständnis des Offshore-Teams in fehlenden Sprachkenntnissen. Fachliche Dokumente des Transportunternehmens waren auf Deutsch und enthielten viele Fachausdrücke. Auch im System wurde Deutsch als Hauptsprache genutzt, nicht nur für in der Benutzungsoberfläche sichtbare Texte, sondern auch für XML-Nachrichten, interne Fehlermeldungen u. Ä. Offizielle Übersetzungen erschienen nur für einige Teile, kamen spät und waren recht schlecht. Das Onshore-Team unterstellte den Offshore-Entwicklern auch fehlendes Verständnis der fachlichen Domäne und mangelhafte Kundennähe. Das Offshore-Team konnte nicht direkt mit dem Kunden kommunizieren und war auf Unterstützung beim Verständnis der Anforderungen und anderer fachlicher Artefakte angewiesen. Ein Beispiel war ein Test-Client, der nur auf Deutsch verfügbar war. Sein Einsatz wurde kurz vom Offshore-Team geprüft. Anscheinend verstanden sie ihn nicht. Er wurde daher nicht eingesetzt, obwohl er für die Entwicklung sehr hilfreich gewesen wäre. Erst als sich der Onshore-Produktmanager damit beschäftigte und den Wert erkannte, wurde er verwendet.

Die Selbsteinschätzung des Offshore-Teams war eine andere. Sie sahen die Ursache für die Probleme hauptsächlich in der komplexen Produktpalette des Transportunternehmens. Die fremde Sprache wäre hingegen kein großes Problem, die würden sie mittlerweile verstehen. In den Kunden und seine Anforderungen könnten sie sich dank der guten Arbeit des Onshore-Produktmanagers als Kunden-Stellvertreter gut hineinversetzen.

Sie bemängelten, dass ihnen die Anforderungen zu detailliert vorgegeben wurden. Dadurch, dass die Storys nicht nur die fachlichen Anforderungen, sondern auch technische Details enthielten, hätten sie bei der Umsetzung wenig Raum für eigene Ideen. Die Offshore-Entwicklungsmanagerin gab an, dass der Onshore-Analyst sich auf seine Rolle als Analyst konzentrieren und sich nicht in die Technik einmischen sollte. Storys sollten

rein fachlich sein und keine technischen Vorgaben beinhalten. Des Weiteren fehlte dem Offshore-Team die Zeit für notwendige Refactorings, da der Onshore-Produktmanager die Entwicklung allein über die erledigten fachlichen Aufgaben steuerte. Er sah daher nicht direkt die positiven Auswirkungen von Refactorings. Das Offshore-Team musste ständig um Zeit für Refactorings und andere interne Verbesserungen ringen.

Reflexion:

Durch die getrennten Interviews mit beiden Seiten sind deutliche Unterschiede in der Einschätzung der Fähigkeiten des Offshore-Teams bezüglich technischem Können, fachlichem Verständnis und Sprachkenntnissen sichtbar geworden. Diese unterschiedliche Wahrnehmung ist typisch für verteilt arbeitende Teams, bei denen sich die Akteure wenig kennen und nur begrenzt austauschen können. Sie führte zu abweichenden Erwartungen an die Art und Weise der Anforderungsübermittlung.

Das Onshore-Team fühlte eine gewisse Hilflosigkeit angesichts der vermuteten technischen Inkompetenz des Offshore-Teams. Onshore hatte man wenig Einflussmöglichkeiten auf die technische Umsetzung und versuchte, sich durch technische Vorgaben in den Storys etwas abzusichern. Aus Sicht des Offshore-Teams scheint es verständlich, dass man sich Freiheiten bei der Umsetzung der Anforderungen wünschte (vgl. die Diskussion in Abschnitt 3.4.5). Offshore sah man die Implementierung als eigene Aufgabe an, die nicht in der Kompetenz des Onshore-Teams lag.

Für einzelne Probleme hätten sich konkrete Lösungsmöglichkeiten angeboten, wenn sie denn von beiden Teams erkannt worden wären. So hätte für Refactorings explizit Zeit eingeplant werden müssen, da aufgeschobene Umstrukturierungen schnell zu schlecht wartbarem Quelltext führen können und dadurch spätere Änderungen viel aufwendiger werden. Dies hätte z. B. in Form von Konsolidierungsiterationen geschehen können, in denen keine neuen fachlichen Storys umgesetzt werden. Die Sprachprobleme hätten durch Hinzunahme eines deutschsprachigen Mitarbeiters im Offshore-Team entschärft werden können. Speziell für das Problem mit dem Test-Client hätte sich eine softwaretechnische Möglichkeit angeboten, bei der die XML-Nachrichten durch eine spezielle Komponente übersetzt werden. Die Übersetzungstabelle hätte onshore gepflegt werden können.

Einsatzmöglichkeiten der Muster:

Dem Vorfall liegt die Fragestellung zugrunde, wo technisches und wo fachliches Wissen vorhanden sein muss. Die Hauptverantwortung des Onshore-Produktmanagers war es, für eine angemessene Unterstützung der fachlichen Prozesse der Anwender zu sorgen. Da es in diesem Projekt Probleme mit der technischen Umsetzung der Anforderungen gab, musste er sich mehr in die Technik einarbeiten als in vergleichbaren Projekten in derselben Rolle. Dies entsprach aber eigentlich nicht seiner Rolle und führte zu Frustration.

Es wird deutlich, dass im Projekt keine übergreifende Kompetenz zu fachlichen Anforderungen und technischer Umsetzung vorgesehen war. Die Vermittlung zwischen diesen Gebieten gehört zu den klassischen Aufgaben eines Softwarearchitekten. Im Gegensatz zum Produktmanager hätte ein Architekt als technischer Experte den Umsetzungsaufwand von Anforderungen und die Fähigkeiten der Entwickler beurteilen können. Er hätte mit einem architekturzentrierten Vorgehen die Entwicklung besser steuern und auch Refactorings u. Ä. anordnen können. So wären interne Storys im Projektplan sichtbar geworden und hätten eingeplant werden können.

Es scheint keine gute Idee gewesen zu sein, die fachlichen Storys durch konkrete technische Vorgaben anzureichern, ohne eine technische Steuerung auf anderer Ebene zu etablieren. Dadurch wurde die gestalterische Freiheit der Offshore-Entwickler zu stark beschnitten. Component Tasks hätten ein Mittel geboten, fachliche Anforderungen mit technischen Details zu verbinden, ohne die Implementierung zu stark vorzugeben.

Vorfall 5: Fehlen von Komponententests

Nach der ersten Produktivstellung wurden viele Fehler durch die Pilotanwender gemeldet. Aufgrund einer mangelhaften Absicherung des Quelltextes mit Komponententests wurden bei der Fehlerbehebung oft neue Fehler ins System eingebracht. Es gelang nicht, die Testabdeckung deutlich zu erhöhen. Ein Grund bestand in der komplexen inneren Struktur, die nicht auf leichte Testbarkeit hin ausgelegt war.

Belege:

„Es gibt keine Unit-Tests. Das liegt zu einem großen Teil daran, dass es mit der aktuellen Architektur auch gar nicht möglich ist, Unit-Tests zu schreiben. Dazu ist es viel zu monolithisch. [...] Das Debuggen verschlingt Aufwände bis zum Abwinken. Wenn man da Unit-Tests einführen könnte, würde man diese Debugging-Aufwände sehr gut reduzieren können.“

Aus dem Interview mit dem Onshore-Analysten

“We also write JUnits [Komponententests mit JUnit]. [...] As you are developing a feature or fixing a bug-fix, you write a JUnit or a couple of JUnits, depending on the size of it. And then that is run as part of our daily-builds.”

Aus dem Interview mit der Offshore-Entwicklungsmanagerin

“We do not really use test-driven development. Not as much as we should do. [...] I suppose the closest we get is with acceptance tests. Before we write code we have a story with acceptance tests. They are a kind of substitute for our software tests.”

Aus dem Interview mit dem Offshore-Entwickler

Beschreibung:

Insbesondere in den ersten Phasen des Projekts wurden sehr wenige Komponententests geschrieben. Sie wurden nach der Funktionalität implementiert und trieben nicht im Sinne des Test-First-Ansatzes (vgl. [Beck 2002]) die Entwicklung. Die Abdeckung des Quelltextes mit Tests wurde nicht gemessen, sodass die Entwickler nicht unter Druck standen, Tests zu schreiben. Sie konnten Komponententests schreiben und wurden dazu animiert, doch mussten sie es nicht.

Bei der Analyse durch den Onshore-Analysten wurde festgestellt, dass „praktisch keine Testabdeckung“ vorhanden war. Als das Projekt komplexer wurde und das erste Release ausgeliefert worden war, meldeten die Pilotanwender viele Fehler. Ohne durch Tests abgesichert zu sein, führten die Entwickler beim Beheben der gemeldeten Fehler häufig neue ein. Nach Ansicht des Onshore-Analysten, der mit den Anwendern in dieser Phase intensiv kommunizierte, blieb die Anzahl der Fehler dadurch konstant oder stieg sogar. Für die behobenen Fehler wurden auch meist keine überprüfenden Tests geschrieben, sodass einige erneut auftraten.

Das Onshore-Team sah die Unzulänglichkeiten und forderte das Offshore-Team auf, mehr Komponententests zu schreiben. Aufgrund der Arbeitsteilung im Projekt konnten sie ihnen das jedoch nicht vorschreiben. Die Offshore-Entwicklungsmanagerin vertrat die Auffassung, dass das Team ausreichend Tests schrieb. Die Offshore-Entwickler sahen jedoch das Verbesserungspotenzial und bewerteten die Testabdeckung in ihrer AUP-Selbsteinschätzung als schlecht. Trotzdem änderte sich an der Situation wenig.

Nach Einschätzung des Onshore-Analysten hat nur die recht umfangreiche Abdeckung mit Akzeptanztests, die aufwendig manuell abgearbeitet wurden, verhindert, dass es zu größeren Problemen kam.

Reflexion:

Tests sind gerade bei so komplexen Domänen wie der dieses Projekts wichtig. Durch die sich oft ändernde Produktpalette des Transportunternehmens waren fortdauernd Anpassungen notwendig. Akzeptanztests waren sehr hilfreich. Sie konnten aber kein Ersatz für Komponententests sein.

Es scheint unverständlich, warum nicht mehr Tests geschrieben wurden. Die notwendige Technik stand zur Verfügung und die Akteure beider Teams sahen den Wert von Tests. Ein Grund lag darin, dass die Abnahmeprozesse zwar die Erfüllung von Akzeptanztests, nicht aber die Absicherung durch Komponententests forderten. Der Onshore-Analyst vermutete den Hauptgrund jedoch in der komplexen Architektur. Durch den monolithischen Aufbau der Anwendung war es schwierig, testbare Teile zu extrahieren. Dadurch war der Aufwand für die Entwickler so groß, dass sie sich mit den Tests sehr schwer taten, obwohl sie den Nutzen sahen. Durch den Zeitdruck im Projekt wurde dann die Implementierung neuer Funktionen dem Testen vorgezogen.

Einsatzmöglichkeiten der Muster:

Softwarearchitektur und Komponententests weisen Wechselwirkungen auf. Eine gut entkoppelte Architektur ist leicht testbar. Durch Tests können auch architektonische Probleme aufgezeigt werden. Architektur und Tests sollten daher parallel weiterentwickelt werden. In diesem Projekt wurde die Architektur vernachlässigt. Darunter litten auch die Tests.

Eine architekturzentrierte Entwicklung hätte nicht nur die Erstellung von Komponententests erleichtern können. Ein Softwarearchitekt hätte außerdem die Tests bei der Abnahme neuer Funktionen einfordern und deren Qualität und Quelltextabdeckung prüfen können. Bei Component Tasks sind die Abnahmekriterien für jede Story explizit vorgegeben. Wenn ein darauf basierender Abnahmeprozess von Anfang an etabliert gewesen wäre, wären vermutlich viele Probleme nicht entstanden.

6.2.4 Das Empirieprojekt Gamma

Kurzvorstellung des Projekts

Projektkontext: Gegenstand von Projekt Gamma war die Entwicklung einer innovativen Anwendung für Mid-Office-Arbeitsplätze zum Einsatz in Russland. Sie wurde von einem Onshore-Team in Deutschland und einem Offshore-Team in Russland durchgeführt. Das Onshore-Team bestand aus dem Projektleiter, einem Entwickler, der bei der Ermittlung der Anforderungen und dem Entwurf der Benutzungsoberfläche unterstützte und anfangs auch mitentwickelte, und einem Softwarearchitekten, der dem Projekt in Teilzeit als Berater zur Verfügung stand. Alle Onshore-Mitarbeiter wurden parallel in weiteren Projekten eingesetzt. Das Offshore-Team bestand aus einem Entwicklungsleiter und vier, zeitweise fünf Entwicklern. Zusätzlich war ein Manager des Kunden vor Ort in Russland, der den Kontakt zu den späteren Anwendern des Systems hielt. Das formale Projektmanagement und die Kostenkontrolle erfolgten durch die Firmenzentrale des Kunden in den USA. Abbildung 6.4 zeigt das Kooperationsmodell.

Zeitrahmen: Das Projekt startete im Mai 2006 mit einem Proof of Concept[2]. Die anfängliche Entwicklung eines Prototyps fand offshore unter Beteiligung eines Entwicklers des Onshore-Teams statt. Anschließend entwickelte das Offshore-Team den Prototyp weiter. Nach drei Monaten wurde entschieden, diesen zu einer produktiv einsetzbaren Anwendung auszubauen. Für November 2006 war das Ende der ersten Projektphase mit anschließendem Betatest geplant. Die Interviews fanden vorher, im September 2006, statt.

[2]Ein Proof of Concept, auf Deutsch am ehesten mit „Machbarkeitsstudie“ übersetzbar, soll die prinzipielle Durchführbarkeit eines Vorhabens zeigen. Dazu wird ein Prototyp entwickelt und evaluiert.

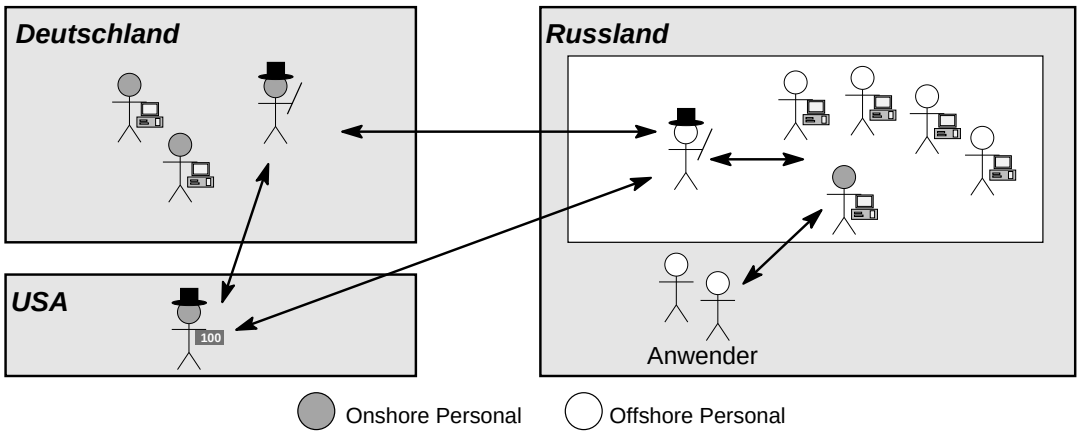


Abbildung 6.4: *Kooperationsmodell des Empirieprojekts Gamma*

Umsetzung agiler Methoden: Mithilfe des Onshore-Beraters wurde ein agiler Prozess etabliert. Die Entwicklung wurde in mehrere Iterationen aufgeteilt. Am Ende jeder Iteration stand ein Release, das intern getestet wurde. Die Anforderungen wurden individuell für jede Iteration festgelegt. Sie wurden als User Storys beschrieben und mit dem Werkzeug XPlanner verwaltet, das von beiden Standorten aus zugreifbar war. Das Concurrent Versions System (CVS) diente als Versionsverwaltungssystem. Dieses System war ebenso von beiden Standorten aus erreichbar. Für die Kommunikation wurde hauptsächlich synchrone Kommunikation per Telefon sowie asynchrone Kommunikation per E-Mail genutzt.

Softwarearchitektur: Da die Anwendung nicht auf Basis einer bestehenden Anwendung oder eines vorhandenen Rahmenwerks entwickelt wurde, befand sich die Architektur im Stadium der Neuentwicklung. Die grundlegende Architektur wurde in der Prototypphase gemeinsam durch Onshore- und Offshore-Entwickler festgelegt. Sie basierte auf einzelnen Komponenten. Für deren Zuschnitt wurden Metaphern benutzt, die in einem Whitepaper beschrieben wurden. Implizite Beziehungen zwischen den Metaphern wurden als explizite Beziehungen zwischen den Komponenten umgesetzt. Die Komponenten und ihre Beziehungen wurden mit einem proprietären Plug-In-Rahmenwerk auf Basis von XML-Dateien definiert. Das Rahmenwerk ermöglichte die Prüfung, ob die Beziehungen zwischen den Komponenten eingehalten wurden. Weitere Architekturregeln wurden nicht regelmäßig überprüft.

Der Entwicklungsleiter des Offshore-Teams war zuständig für die Planung des Ressourceneinsatzes auf der Offshore-Seite und der Hauptkontakt für das Onshore-Team. Er war der erfahrenste Entwickler und wurde bei komplexeren Entwurfsproblemen zurate gezogen. Er agierte jedoch nicht als Softwarearchitekt, der generell architektonische Entscheidungen trifft und die Softwarearchitektur dokumentiert und überprüft. Die

Einflussmöglichkeiten des Onshore-Softwarearchitekten waren gering. Er hatte nur eine beratende Tätigkeit und konnte Hinweise auf Probleme und Lösungsvorschläge geben. Im späteren Verlauf des Projekts war er nur wenige Tage pro Monat für das Projekt aktiv.

Einordnung in die Übersichtskarte des IT-Sourcings

In Abbildung 6.5 ist dargestellt, wie sich das Projekt in die Übersichtskarte des IT-Sourcings einordnen lässt.

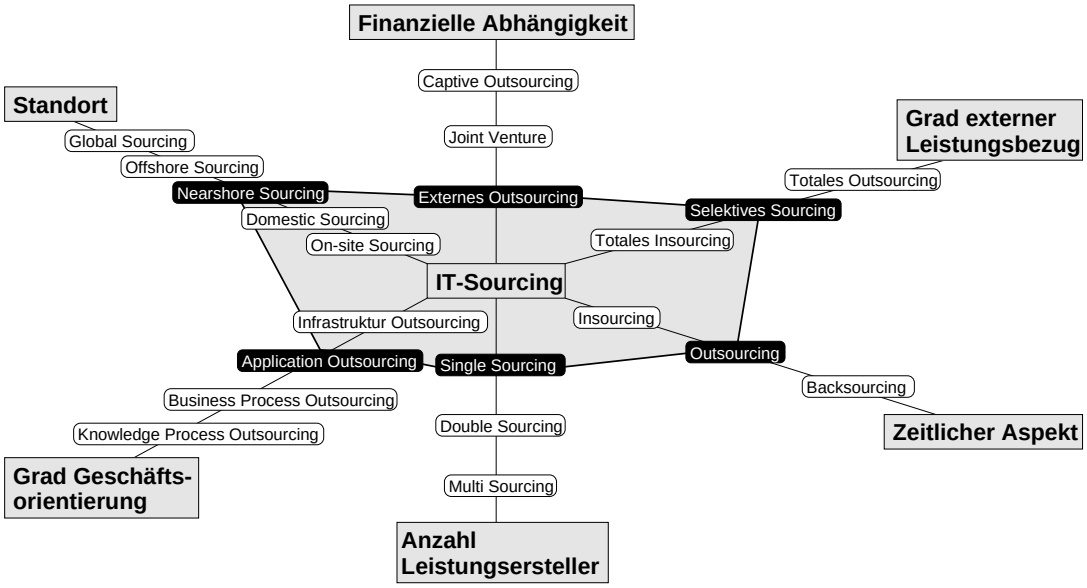


Abbildung 6.5: *Einordnung von Projekt Gamma in die IT-Sourcing-Map*

Das Projekt ist als Application Outsourcing einzustufen, da eine neue Anwendung entwickelt wurde *(Grad Geschäftsorientierung)*. Für Deutschland ist Russland Nearshore-Standort *(Standort)*. Bemerkenswert ist die Tatsache, dass die Anwendung für Russland, dem Heimatmarkt des Offshore-Teams, vorgesehen war. Dieses Team hatte von Anfang an Wissen über die Anwendungsdomäne und keine sprachlich und kulturell bedingten Kommunikationsprobleme mit den Endanwendern. Da das Projekt von Deutschland aus gesteuert wurde und dort auch die Anforderungen festgelegt wurden, kann es trotzdem als Offshore-Entwicklungsprojekt gesehen werden. Der Offshoring-Dienstleister war rechtlich selbstständig, daher war das Outsourcing extern *(Finanzielle Abhängigkeit)*. Es handelte sich um Selektives Sourcing, da Teile der Anwendung (hauptsächlich Host-Programme) weiterhin im Kernunternehmen entwickelt wurden. Auch bei der eigentlichen Anwendung waren Onshore-Entwickler teilweise beteiligt *(Grad externer Leistungsbezug)*. Die Leistung wurde zum ersten Mal im Outsourcing fremdvergeben

(Zeitlicher Aspekt) und mit Single Sourcing von einem einzelnen Dienstleister erbracht *(Anzahl Leistungsersteller)*.

Beschreibung der Untersuchung

Die Datenerhebung erfolgte durch Einzelinterviews und Auswertung der zur Verfügung gestellten Materialien. Die Interviews wurden durch den Autor in den Räumen des Onshore-Teams in deutscher Sprache durchgeführt. Interviewt wurden der Projektleiter (Interview am 22. September 2006, ca. 1 Stunde), der Softwarearchitekt (Interview am 22. September 2006, ca. 1,5 Stunden) und der Entwickler (Interview am 11. September 2006, ca. 2,5 Stunden). Die Interviews wurden jeweils vorher vorbereitet und anhand eines Interviewleitfadens als semistrukturierte Interviews durchgeführt (siehe Anhang D.1). Interviews mit dem Offshore-Team waren nicht möglich.

Von den Interviewten wurden Dokumente aus dem Projekt ausgewählt und dem Autor zur Verfügung gestellt. Hauptsächlich waren dies das initiale Whitepaper, in dem die Ziele und die grundsätzliche Herangehensweise des Projekts beschrieben wurden, mehrere Powerpoint-Präsentationen zum Projekt, Mockups der Benutzungsoberfläche auf Powerpoint-Basis, interne Notizen des Onshore-Teams und eine lauffähige Version der Anwendung.

Vorfälle, Folgen und Auswertung

Aus dem erhobenen Material wurden drei wesentliche Vorfälle zur Diskussion ausgewählt: 1. Management der Anforderungen und Fortschrittskontrolle, 2. Innere Qualität und Softwarearchitektur und 3. Komponententests und Zusammenarbeit.

Vorfall 1: Management der Anforderungen und Fortschrittskontrolle

Im Projektverlauf wurden mit zunehmender Häufigkeit Anforderungen durch das Offshore-Team nicht zur Zufriedenheit des Onshore-Teams umgesetzt. Es wurde für den Onshore-Projektleiter zusehends schwerer, den Projektfortschritt festzustellen.

Belege:

„[Wenn wir Anforderungen stellen,] kommt oft so ein ‚Ja, ist ok, haben wir verstanden', es passiert aber dann trotzdem nichts. [...] Das ist so eine Art Kommunikationsproblem, was wir da haben."

„Richtig organisiert ist die Abnahme umgesetzter Anforderungen nicht. Wir haben die Erfahrung gemacht, dass es ganz gut ohne Kontrolle im Proof of Concept gelaufen ist. [...] Richtig überprüfen tun wir jetzt [bei der Entwicklung des Prototyps] nichts mehr. [...] Das Schwierige ist, dass wir keinen richtigen Überblick haben, wie weit die sind.

Das ist schlecht nachzuvollziehen, wenn man nicht mit im Entwicklungsprozess drin sitzt.“

Aus dem Interview mit dem Onshore-Entwickler

Beschreibung:

Die Anforderungen wurden generell vom Onshore-Team aufgestellt und dem Offshore-Team in Prosaform per E-Mail oder Telefon übermittelt. Bei Änderungen, die in der Benutzungsoberfläche sichtbar waren, lieferte der Onshore-Entwickler häufig Mockups der Benutzungsoberfläche mit, die er selbst erstellt hatte. Das Offshore-Team verwaltete die Anforderungen mit dem Werkzeug XPlanner, auf das auch das Onshore-Team Zugriff hatte.

Im anfänglichen Verlauf des Projekts, beim Proof of Concept, hatte das Onshore-Team die Offshore-Entwickler als technisch gut, verlässlich und zu eigenständiger Arbeit fähig kennengelernt. Daher wurde ihnen viel Freiheit zugestanden. Dadurch und durch den zunehmenden Kostendruck kontrollierte das Onshore-Team die Arbeitsergebnisse immer seltener. Der Onshore-Softwarearchitekt reduzierte die Nachverfolgung der Anforderungen und ihrer Umsetzung im XPlanner. Der Projektfortschritt wurde fast nur noch über die akkumulierten Kosten und in der Benutzungsoberfläche sichtbare Funktionen überwacht.

Nach einiger Zeit bemerkte das Onshore-Team, dass Anforderungen durch das Offshore-Team nicht richtig oder überhaupt nicht umgesetzt wurden. Stattdessen wurden andere, nicht verlangte Änderungen an der Funktionalität vorgenommen. Durch die besondere Projektsituation, bei der die Anwendung für den Heimatmarkt des Offshore-Teams entwickelt wurde, besaß dieses ein gutes Verständnis der Bedürfnisse der Anwender. Es war daher organisatorisch vorgesehen, dass das Offshore-Team eigene Ideen zur Weiterentwicklung der Anwendung einbringen konnte. Es wurde vom Onshore-Team jedoch sehr kritisch gesehen, dass sich das Offshore-Team ohne Abstimmung über gemeinsam getroffene Entscheidungen bezüglich der Priorisierung der Anforderungen hinwegsetzte. Obwohl diese Thematik bei Abstimmungstreffen angesprochen wurde, änderte sich nicht viel.

Ein verwandtes Problem entstand bei der Fortschrittskontrolle. Im Laufe des Projekts fiel es dem Projektleiter immer schwerer, den Projektstatus zu bestimmen. Die Umsetzung der Anforderungen durch das Offshore-Team war ihm nicht transparent. Auf Nachfragen beim Offshore-Team wurde ihm oft ein hoher Fertigstellungsgrad („95% sind fertig!“) genannt, den er nicht überprüfen konnte. In der Praxis zogen sich die verbleibenden Arbeiten jedoch in der Regel lange hin. Im Laufe des Projekts nahmen die Verzögerungen zu und der Zeitplan geriet in Gefahr.

Aufgrund dieser Probleme hatte der Projektleiter das Gefühl, dass ihm die Steuerung des Offshore-Teams und damit des Gesamtprojekts entglitt.

Reflexion:

Der Zustand des Projekts zum Zeitpunkt der Untersuchung zeigte mehrere Probleme in der Kooperation der Teams. Diese Probleme begannen, als man aus Kostengründen den Arbeitsumfang des Onshore-Teams verringerte. Dadurch litt das Management der Anforderungen durch den Projektleiter. Der Entwickler beteiligte sich nur noch sporadisch an der Weiterentwicklung der Anwendung. Der Softwarearchitekt konnte die innere Qualität der Anwendung nicht mehr untersuchen. Solange die Anforderungen durch das Offshore-Team korrekt und in der vorgegebenen Zeit umgesetzt wurden, sah man keine Notwendigkeit zum Handeln. Beim Proof of Concept gab es keine Probleme. Als die Entwicklung des Prototyps nicht mehr gut funktionierte, wurde deutlich, dass die Abkopplung des Onshore-Teams vom Entwicklungsprozess nicht sinnvoll war.

Die Entkopplung der Akteure durch die geografische und organisatorische Entfernung beeinflusste auch die anderen GSD-Dimensionen. Der Informationsfluss zwischen den Akteuren flaute ab. Wichtige Informationen wurden dem anderen Team nicht oder erst auf Nachfrage mitgeteilt. Es zeigte sich, dass der Entwicklungsprozess keine ausreichenden Kontroll- und Kommunikationsmechanismen umfasste. Es war sehr schwierig, solche Mechanismen im Nachhinein einzuführen. Die Technik stellte bei diesem Vorfall kein Problem dar. Es waren Zugriffsmöglichkeiten auf die Dokumentation der Anforderungen und ihrer Umsetzung über den XPlanner vorhanden. Auch standen den Teams ausreichende Kommunikationsmittel zur Verfügung.

Die eingesetzten agilen Methoden haben geholfen, die Probleme frühzeitig aufzudecken. So wurde das Problem der nicht oder inkorrekt umgesetzten Anforderungen bei den regelmäßigen Releases entdeckt. Dabei wurde auch deutlich, dass die Fortschrittskontrolle auf ungenauen Daten fußte und den tatsächlichen Stand der Entwicklung nicht korrekt widerspiegelte.

Einsatzmöglichkeiten der Muster:

Mit einer aktiveren Einbeziehung des Softwarearchitekten hätte man den Problemen begegnen können. Da er sich mit technischem Wissen den Respekt der Entwickler verschaffen und als Experte für die Umsetzung fachlicher Anforderungen dem Projektleiter zur Seite stehen kann, kann er zwischen beiden Seiten vermitteln. Ein Softwarearchitekt spielt damit bei der Steuerung von Entwicklungsprojekten eine wichtige Rolle, gerade wenn Entwickler und Projektleiter durch die Verteilung Probleme bei der Kommunikation haben.

Im Interview mit dem Softwarearchitekten äußerte dieser die Befürchtung, das Offshore-Team durch zu starre Architekturvorgaben einzuschränken und ihnen damit ihren Expertenstatus zu entziehen. In so einem Fall kann ein Architekturrahmen vorgegeben werden, bei dem die Details der Umsetzung mit damit einhergehenden Anpassungen der Architektur weiterhin dem Entwicklerteam überlassen sind. Für die Übermittlung von Anforderungen können gut Component Tasks eingesetzt werden. Der Softwarearchitekt

sollte die Qualität der Umsetzung prüfen und die Richtung der Architekturentwicklung bestimmen. Damit ist er in einer Position, in der er den Stand und die innere Qualität der Umsetzung der Anforderungen beurteilen und dem Projektleiter genaue Fortschrittsberichte geben kann.

Vorfall 2: Innere Qualität und Softwarearchitektur

Anfangs hatte der Onshore-Entwickler noch aktiv an der Anwendung mitentwickelt. Später arbeitete das Offshore-Team zunehmend eigenständiger und entwickelte die Softwarearchitektur in eigener Verantwortung weiter. Das Onshore-Team gewann im Projektverlauf den Eindruck, dass die innere Qualität der Anwendung abnahm, und war deswegen besorgt.

Belege:

„Im Moment glaube ich, wenn ich weiterhin nur an ein paar Calls teilnehme und sonst nicht viel mache, würde ich es [konkrete Probleme mit der inneren Qualität, wie z. B. Zyklen] wahrscheinlich nur mitkriegen, wenn irgendjemand eine Bemerkung macht, die mir verdächtig erscheint.“

„Wir haben tatsächlich keine gemeinsame Architekturvorstellung. [. . .] Ich fürchte, dass das ein echter Motivationsblocker für die wäre, wenn ich ihnen da [in die Softwarearchitektur] zu sehr reinreden würde. [. . .] Wenn ich mich stärker involvieren würde, wäre bald ein Code-Review fällig, auf alle Fälle!“

Aus dem Interview mit dem Onshore-Softwarearchitekten

Beschreibung:

Die ersten Releases der Anwendung in der Proof of Concept-Phase wurden von dem Onshore-Entwickler gemeinsam mit dem Offshore-Team implementiert. Dabei wurde die grundlegende Softwarearchitektur definiert. Bei einer Architekturanalyse einer frühen Version der Anwendung durch den Softwarearchitekten wurden Zyklen gefunden. Diese konnten leicht aufgelöst werden. Die Analyse blieb jedoch ein Einzelfall. Während der Implementierung des Prototyps arbeitete nur das Offshore-Team an der Entwicklung der Anwendung. Das Onshore-Team beschränkte sich auf das Projektmanagement, die Ermittlung und Verfolgung der Anforderungen und die Akzeptanztests der Releases.

Der Onshore-Softwarearchitekt verzichtete bewusst auf eine Mitsprache bei der weiteren Anpassung der Architektur der Anwendung. Zum einen sprachen finanzielle Gründe gegen eine stärkere Beteiligung. Zum anderen hatte er während des Proof of Concept das Offshore-Team als fähige Entwickler kennengelernt und vertraute ihnen. Er fürchtete, das Offshore-Team durch Diskussion und Kontrolle der Architektur zu demotivieren und ihnen den Expertenstatus streitig zu machen. Durch die einseitige Weiterentwicklung

der Anwendung verlor das Onshore-Team schnell den Überblick über den inneren Zustand der Anwendung. Die anfangs geschaffene gemeinsame Architekturvorstellung ging verloren.

Als es bei der Umsetzung der Anforderungen zu Problemen kam (siehe Vorfall 1), vermutete der Onshore-Softwarearchitekt aufgrund des Verhaltens des Offshore-Teams, dass es Probleme mit der inneren Qualität der Anwendung gab. Das Offshore-Team gab auf Nachfragen keine befriedigende Auskunft zum Zustand der Anwendung und zu durchgeführten Qualitätssicherungsmaßnahmen. Dies führte beim Onshore-Projektleiter zu Besorgnis, da die Anwendung als Basis für weitreichende Weiterentwicklungen dienen sollte, insbesondere auch durch andere Teams. Daher waren eine qualitativ hochwertige, gut dokumentierte Softwarearchitektur und eine gute innere Qualität kritisch.

Dem Offshore-Team fehlten die Einflussmöglichkeiten, um das Onshore-Team zur Erhöhung der inneren Qualität der Anwendung anzuhalten. Zum Zeitpunkt der Interviews hatte man noch keine Lösung des Konflikts gefunden.

Reflexion:

Wie beim ersten Vorfall ist eine Hauptursache für die Probleme in der Abkopplung der Onshore-Teams vom Entwicklungsprozess zu sehen. Das Onshore-Team konzentrierte sich im Projektverlauf aus Zeit- und Kostengründen auf die Erhebung der Anforderungen und die Umsetzungskontrolle anhand der entwickelten Anwendung. Nicht nur, dass der Projektleiter dadurch die Steuerung des Projekts vernachlässigte, auch der Softwarearchitekt zog sich zu sehr zurück. Dies war rückblickend keine gute Entscheidung. Die beim Proof of Concept einmalig durchgeführte Architekturanalyse des Prototyps hatte gezeigt, dass eine Auflösung von Zyklen und illegalen Aufrufbeziehungen recht einfach möglich war. Eine Fortführung der Analysen hätte daher wohl verhindern können, dass die innere Qualität der Anwendung sank.

Das Onshore-Team hatte im späteren Projektverlauf nur wenig Möglichkeiten, die innere Qualität der Anwendung zu beurteilen. Unabhängig davon, ob wirklich ein Qualitätsproblem bestand, führte dieser Umstand zu einem Vertrauensverlust gegenüber dem Offshore-Team. Dies zeigt wieder, wie leicht es durch die besondere Konstellation von Offshoring-Projekten zu Problemen kommen kann. Onshore- und Offshore-Akteure arbeiteten geografisch entfernt in verschiedenen Organisationen, wodurch die Aktivitäten im jeweils anderen Team nicht transparent waren. Die Koordinationsprozesse waren darauf ausgerichtet, onshore ermittelte Anforderungen möglichst kosteneffizient offshore umzusetzen. Informationen über Schwierigkeiten bei der Entwicklung, die innere Qualität der Anwendung und Details der Anforderungsumsetzung wurden nicht ausgetauscht. Vorhandene technische Mittel zur Dokumentation und Überprüfung der Softwarearchitektur wurden nicht ausgenutzt.

Einsatzmöglichkeiten der Muster:

Der beschriebene Vorfall zeigt, dass es bei der Entwicklung innovativer interaktiver Anwendungssysteme nicht ausreichend ist, die Softwarearchitektur nur auf recht abstrakter Ebene am Anfang zu definieren. Anforderungen, die implementierte Anwendung und damit auch ihre Softwarearchitektur ändern sich im Entwicklungsverlauf stark.

Die Erfahrung des Autors ist, dass eine von allen Beteiligten geteilte Architekturvorstellung sehr hilfreich ist. So wird ein Austausch über den Implementierungsaufwand von Anforderungen und innere technische Details der Anwendung erleichtert. Dadurch, dass Details der Softwarearchitektur explizit gemacht werden, wird ein Verständnis und eine mögliche Weiterentwicklung durch andere Entwickler erleichtert. Im betrachteten Projekt wurde verpasst, diese gemeinsame Architekturvorstellung zu etablieren und zu pflegen. Da der Onshore-Softwarearchitekt dem Offshore-Team die volle Verantwortung über die Architektur überlassen hatte, wäre es schwierig, eine gemeinsame Architekturvorstellung zu einem recht weit fortgeschrittenen Stand des Projekts durchzusetzen.

Um die innere Qualität zu verbessern und dem Onshore-Team wieder mehr Vertrauen in die Arbeit des Offshore-Teams zu geben, würde sich eine grundlegende Architekturanalyse anbieten. Durch die beim Proof of Concept durchgeführte Analyse konnte auch in diesem Projekt die Qualität verbessert werden, wenn auch nur einmalig. In einer Architekturanalyse können sinnvolle Refactorings identifiziert werden. Auf dieser Grundlage lässt sich ihre Notwendigkeit beurteilen und sie können eingeplant werden. Refactorings, die fortlaufende Überprüfung von Architekturregeln und -metriken und andere qualitätssichernde Maßnahmen erhalten durch regelmäßige Architekturanalysen eine größere Bedeutung. Dies spielt insbesondere beim Offshoring eine Rolle, da solche internen Storys mangels Einsicht in Interna des Projekts vom Onshore-Team selten eingeplant werden. So auch in diesem Projekt, wo die Planung rein anforderungsgetrieben erfolgte. Hier ist ein erfahrener Softwarearchitekt gefragt, der zum einen den inneren Zustand der Anwendung beurteilen kann, zum anderen aber auch Einblick in aktuelle und zukünftige Anforderungen hat. Mit diesem Wissen lässt sich abschätzen, wie viel Aufwand in Refactorings und andere qualitätssichernde Maßnahmen fließen sollte, um die Weiterentwicklung und spätere Wartung zu erleichtern.

Das Dual-Shore-Modell hätte eine gute Möglichkeit geboten, durch zeitweise Arbeit beim jeweils anderen Team mehr Transparenz in das Projekt zu bringen und das gegenseitige Verständnis zu verbessern. Die dafür aufzubringenden Mehrkosten hätten durch eine verbesserte innere Qualität der Software und damit weniger Aufwand für die Fehlerbehebung zumindest teilweise ausgeglichen werden können.

Vorfall 3: Komponententests und Zusammenarbeit

Es kam zu einem Konflikt, als das Onshore-Team darauf bestand, dass die Offshore-Entwickler mehr Komponententests schreiben sollten. Dieser Konflikt konnte nur mit viel Kommunikationsaufwand gelöst werden.

Belege:

„Wir haben von der ersten Iteration an gesagt: ‚Unit-Tests wollen wir haben'. In der zweiten Iteration haben wir gesehen, sie [das Offshore-Team] schreiben keine. [...] In der dritten Iteration haben wir gesagt, das soll mal eine Konsolidierungsiteration sein, da schreibt mal nur Unit-Tests. Sie haben dann aber nur herumgedruckst."

Aus dem Interview mit dem Onshore-Softwarearchitekten

„Fehlende Komponententests sind wirklich ein großes Problem. Ich glaube, das kommt daher, dass sie [das Offshore-Team] nicht den Sinn dieser Testfälle sehen. [...] Es kommen von denen [dem Offshore-Team] immer Argumente wie z. B. ‚Wir wollen doch gerne noch folgendes Feature implementieren, und wenn wir jetzt Testfälle machen, dann muss dieses Feature eben rausbleiben.'"

Aus dem Interview mit dem Onshore-Entwickler

Beschreibung:

Das Onshore-Team forderte seit Beginn des Projekts, dass das Offshore-Team Komponententests schreibt. Sowohl der Onshore-Entwickler als auch der Onshore-Softwarearchitekt wiesen bei mehreren Gelegenheiten darauf hin. Das Offshore-Team schrieb aus Sicht des Onshore-Teams nur widerwillig Tests. Demzufolge war die Abdeckung des Quelltextes mit Tests sehr gering (nach einer einmaligen Messung des Onshore-Softwarearchitekten nur knapp über 10%). Daran änderte auch eine sogenannte Konsolidierungsiteration nichts, bei der nur Tests geschrieben werden sollten.

Als Argumente gegen Tests wurden vom Offshore-Team Zeitprobleme angeführt. Andere Aufgaben waren immer wichtiger als das Schreiben von Tests. Außerdem machten Schnittstellen, die sich noch häufig änderten, Tests aus Sicht des Offshore-Teams sehr aufwendig, da sie oft anzupassen wären. Nachdem Probleme mit der inneren Qualität auftraten (siehe Vorfall 2), drängte das Onshore-Team erneut auf Tests. Der Projektleiter, der anfangs auf die Einsicht des Offshore-Teams vertraut hatte, nahm sich der Angelegenheit verstärkt an. Nach einigen Diskussionen per E-Mail und Instant Messenger begannen die Offshore-Entwickler damit, mehr Tests zu schreiben. Sie taten das jedoch aus Sicht des Onshore-Teams ohne große Motivation und nicht in dem Umfang, wie es sich das Onshore-Team vorstellte.

Reflexion:

Obwohl dieser Vorfall auf den ersten Blick viele Gemeinsamkeiten mit Vorfall 5 des Empirieprojekts Beta aufweist, erklären sich die Probleme in diesem Projekt anscheinend weniger durch eine schlecht testbare Anwendung als vielmehr durch unterschiedliche Wertvorstellungen der Teams und unzureichende Kommunikation miteinander.

Für das Onshore-Team waren Komponententests sehr wichtig und ein Grunderfordernis zur Absicherung der implementierten Funktionen einer Anwendung. Der Onshore-Entwickler war intensives Testen gewohnt. In vielen seiner Projekte wurde sogar testgetrieben entwickelt. Die Erfahrung des Softwarearchitekten war, dass man in jedem Projekt darauf achten muss, dass die Testabdeckung nicht absinkt. Ansonsten entstehen leichter Fehler und deren Behebung kostet viel Zeit, da die fehlerhafte Stelle im Quelltext erst gefunden werden muss. Für das Offshore-Team stellte sich die Situation anders dar. Sie sahen das Schreiben von Tests als zeitaufwendige Tätigkeit, die sie von der Umsetzung der fachlichen Anforderungen abhielt. Da sie an Letzterem hauptsächlich gemessen wurden, war ihnen das wichtiger.

Interessant ist die Tatsache, dass es selbst dem Onshore-Projektleiter lange Zeit nicht gelang, das Offshore-Team zum Schreiben von mehr Tests zu bewegen. Hier zeigt sich, dass durch die räumliche und organisatorische Trennung der Akteure auch die Autorität von Akteuren in leitenden Positionen leiden kann und Sanktionsmechanismen nicht greifen. Sicherlich spielte auch die spezielle Projektsituation eine Rolle.

Komponententests wurden nicht von Anfang an als wichtiger Bestandteil des Entwicklungsprozesses etabliert. Im späteren Projektverlauf, als das Onshore-Team Informationen über die anscheinend mangelhafte Abdeckung des Quelltextes mit Tests erhielt, war eine Lösung der Qualitätsprobleme kaum noch möglich. Denn wenn eine Anwendung längere Zeit ohne oder mit wenig Tests implementiert wird, ist sie schlecht nachträglich testbar. Daher sollten Tests möglichst früh eingeführt werden. Die Technik stellte auch in diesem Fall kein Hindernis dar. Das Testrahmenwerk (JUnit) war vorhanden, es wurde jedoch nicht ausreichend genutzt. Möglichkeiten zur Überwachung der Testabdeckung hätten kurzfristig durch entsprechende Systeme integriert werden können.

Bei den spät unter Druck geschriebenen Tests stellt sich die Frage, wie effektiv sie waren. Es ist recht einfach, schlechte Tests zu schreiben, die den Quelltext nur schlecht absichern. Auch wenn die Tests den Quelltext gut abdecken, können sie zu wenige oder zu simple Prüfungen beinhalten. Dies ist nicht auf den ersten Blick erkennbar, sondern nur nach genauerer Analyse [Westphal 2006]. Daher sollten auch die Testklassen regelmäßig überprüft und ggf. verbessert werden.

Einsatzmöglichkeiten der Muster:

Das Offshore-Team sträubte sich gegen Vorgaben des Onshore-Teams, das diese nicht durchsetzen konnte. Ein im Projektteam anerkannter und etablierter Softwarearchitekt hätte Möglichkeiten, Komponententests einzuführen. Er kennt die Wichtigkeit von Tests und kann anhand der konkreten Anwendung aufzeigen, wo Tests helfen. Er hat damit eine gute Grundlage, um zwischen der Umsetzung neuer Anforderungen und dem Schreiben von Tests abzuwägen und einen Plan für die Entwicklung zu erstellen.

Bei Component Tasks sind Abnahmetests zu jeder Komponente vorgesehen. Nach der Etablierung von Komponententest im Projekt kann der Softwarearchitekt deren Kontrolle übernehmen. Er kann die Effektivität von Tests beurteilen und die Testabdeckung messen. Damit stehen ihm die benötigten Kontrollinstrumente zur Verfügung.

6.2.5 Zusammenfassende Auswertung der Feldstudien

Im Folgenden wird betrachtet, worin die generellen Probleme der betrachteten Empirieprojekte bestanden und wie gut die Muster einsetzbar gewesen wären.

Gemeinsamkeiten der Projekte

Obwohl beide Projekte recht unterschiedlich verliefen und eins (Beta) scheiterte, während das andere (Gamma) zu einem erfolgreichen Produkt führte, weisen sie eine Reihe von für diese Arbeit interessanten Gemeinsamkeiten auf.

In beiden Empirieprojekten sind ähnlich gelagerte Probleme aufgetreten. Dies liegt zum einen an der Auswahl der Projekte und dem Fokus der Befragung, macht zum anderen aber auch die projektübergreifende Bedeutung dieser Problemfelder deutlich.

Die Zusammenarbeit von Onshore- und Offshore-Teams brachte Probleme mit sich, die bei nicht-verteilten Projekten voraussichtlich nicht aufgetreten wäre. Dabei erwies sich weniger die Technik als Ursache von Problemen. Die Infrastruktur, die gemeinsamen Versionsverwaltungssysteme und Kommunikationsanwendungen konnten gut eingesetzt werden. Dagegen haben die Informationsweitergabe und die Zusammenarbeit der Akteure teilweise nicht gut funktioniert. Im Laufe der Projekte zeigten sich Schwächen der Kooperationsprozesse, insbesondere bei der Abnahme und der Qualitätssicherung.

In beiden Projekten hatte die Anwendung agiler Methoden positive Auswirkungen. Besonders hilfreich war, dass die Anforderungen in kleine Einheiten, hauptsächlich User Storys, aufgeteilt wurden und mehrere Releases erfolgten. In beiden Projekten wurde die Softwarearchitektur vernachlässigt. Die Softwarearchitekten wurden aus dem Projekt abgezogen (Beta) oder in ihrer Tätigkeit aus Kostengründen eingeschränkt (Beta und Gamma).

Einsatzmöglichkeiten der Muster

Die in den Vorfällen diskutierten Probleme scheinen für den Projekterfolg sehr relevant gewesen zu sein. Sie handelten von Fragen, die nach den Untersuchungen der vorhergehenden Kapitel auch in anderen global verteilten Entwicklungsprojekten vielfach aufgetreten sind. Die Probleme betreffen insbesondere die Kommunikation, Abstimmung der Teams, Definition von Anforderungen und die Qualitätssicherung ihrer Umsetzung.

In der Betrachtung der Probleme hat sich gezeigt, dass der Kontext aller Probleme den Einsatz eines oder mehrerer Muster erlaubt hätte. Die Reflexion der zu erwartenden positiven Auswirkungen beim Einsatz der Muster lässt erwarten, dass sie in vielen Fällen eine Verbesserung der Situation zur Folge gehabt hätten. Die im letzten Kapitel festgestellten Erkenntnisse lassen sich damit begründen.

Das **Dual-Shore-Modell** hätte zu einer in den meisten Fällen benötigten besseren Kooperation zwischen allen Beteiligten beitragen können. Abstimmungsprobleme hätten wahrscheinlich durch häufigere gegenseitige Besuche vermieden werden können.

Bezüglich der **architekturzentrierten Entwicklung** lassen sich die im letzten Kapitel identifizierten Problemfelder im Umgang mit Architektur (vgl. Abschnitt 5.3.7) auch in diesen Feldstudien nachvollziehen. Der folgenden Aufstellung ist zu entnehmen, in welchen Vorfällen die Problemfelder berührt wurden. Der griechische Buchstabe bezieht sich jeweils auf die Bezeichnung des Projekts, die Ziffer auf die Nummer des Vorfalls.

Evolution und Dokumentation der Architektur: $\beta 2$, $\gamma 2$

Innere Qualität und Komponententests: $\beta 2$, $\beta 5$, $\gamma 2$, $\gamma 3$

Anforderungsdefinition: $\beta 4$, $\gamma 1$

Kontrolle und Steuerung der Entwicklung: $\beta 3$, $\gamma 1$

Aufgabenverteilung auf die Teams: $\beta 1$, $\beta 3$

Die sich aus den Problemfeldern ergebenden Aufgaben eines Softwarearchitekten (vgl. Abschnitt 5.3.8) wurden zu einem großen Teil schon in den Empirieprojekten praktiziert, wenn auch nicht in dem Umfang, in dem es wohl nötig gewesen wäre. Bei den Hauptnutzen (vgl. Abschnitt 5.3.9) ist erkennbar, dass die dort beschriebenen Funktionen von Softwarearchitekten bei den in den Empirieprojekten beschriebenen Vorfällen benötigt worden wären. Dies trift sowohl auf „Verbinden von fachlichem mit technischem Wissen“ als auch auf „Finden einer Balance zwischen starren Vorgaben und Eigenverantwortung der Entwickler“ als auch auf „Vermitteln und Moderieren zwischen den Teams“ zu.

Die verständliche Definition von Anforderungen und die Qualitätssicherung der Implementierung spielten in beiden Empirieprojekten eine große Rolle. **Component Tasks** bieten eine Alternative zu den dort eingesetzten Beschreibungsmitteln. In Kombination mit einem architekturzentrierten Vorgehen hätten Component Tasks die Zusammenarbeit in den Empirieprojekten erleichtern können.

Realisierungsempfehlungen und -alternativen

Aus der Untersuchung der beiden Feldstudien lassen sich weitere Empfehlungen für die Realisierung eines architekturzentrierten Entwicklungsprozesses bei global verteilter Entwicklung ableiten.

Es erscheint wichtig, einen Softwarearchitekten frühzeitig im Projekt zu etablieren und über die gesamte Projektlaufzeit zu involvieren. Der richtige Umgang mit Architektur sollte schon zu Projektbeginn festgelegt und später angepasst werden. Sonst können später leicht Akzeptanzprobleme entstehen, die zu Nachlässigkeiten bezüglich der inneren Qualität führen können. Auch bei einer Zusammenarbeit nach dem Dual-Shore-Modell macht sich in diesem Aspekt die Entfernung zwischen den Teams bemerkbar. Die Architektenrolle sollte bis zum Projektende besetzt sein. Gerade bei agil durchgeführten Projekten spielt die Weiterentwicklung der Architektur eine wichtige Rolle. In den Feldstudien kam es schnell zu Problemen, nachdem die Beteiligung der Softwarearchitekten eingeschränkt wurde.

Aufgrund der Verteilung der Teams wären zwei Softwarearchitekten sinnvoll, einer in jedem Team. Diese Architekten könnten sich regelmäßig abstimmen. Da in dieser Arbeit KMU im Mittelpunkt stehen, scheint diese Lösung aus Kostengründen häufig nicht realisierbar zu sein, wie auch die Feldstudien gezeigt haben.

Bei Agilem Offshoring stellt sich die Frage, ob die Rolle des Architekten onshore oder offshore anzusiedeln ist. Es ist mit Blick auf die diskutierten Punkte sinnvoll, Architekturwissen auch onshore vorzuhalten. Dafür spricht auch, dass ein Architekt vorzugsweise auf der Seite verfügbar sein sollte, die mit der Anforderungsermittlung und der Projektsteuerung betraut ist. Mit Blick auf die Diskussion um Arbeitsplätze (siehe Abschnitt 2.2.1) ist die Architektenrolle als anspruchsvolle Tätigkeit zu beurteilen, die onshore erhalten bleiben sollte. Andererseits besteht die Gefahr, dass bei überwiegender Offshore-Entwicklung der Architekt zu weit von dem Standort der eigentlichen Entwicklung entfernt ist. Die Vorteile eines Onshore-Architekten, so wie er auch im Empirieprojekt Alpha umgesetzt worden war, scheinen zu überwiegen.

6.3 Zusammenfassung

In diesem Kapitel wurden die in den vorhergehenden Kapiteln gewonnenen wissenschaftlichen Erkenntnisse in Form von Mustern dokumentiert. Es wurden drei Muster für global verteilte Entwicklungsprojekte mit agilen Methoden identifiziert: Dual-Shore-Modell, Architekturzentrierte Entwicklung und Component Tasks.

Die Muster wurden auf zwei größere Empirieprojekte angewendet. Die Empirieprojekte wurden als Feldstudien betrachtet. Die Empirie wurde durch persönlich geführte Interviews und die Analyse relevanter Projektdokumente erhoben. In allen Projekten konnte ein Nutzen der Muster festgestellt werden, um die Vorgänge im Projekt besser

zu verstehen und Lösungsmöglichkeiten für relevante Probleme zu eröffnen. Durch die Auswertung konnten weitere Einsatzmöglichkeiten der Muster sowie Realisierungsempfehlungen und -alternativen erarbeitet werden.

Im folgenden letzten Kapitel werden alle Ergebnisse dieser Arbeit zusammengefasst und bewertet.

7 Resümee und Ausblick

In diesem Abschlusskapitel wird die Arbeit zusammengefasst. Die wichtigsten Forschungsergebnisse werden zusammengestellt und vor dem Hintergrund des Forschungsfeldes kritisch gewürdigt. Das Kapitel endet mit einem Ausblick auf weiteren Forschungsbedarf und mögliche Folgearbeiten.

7.1 Zusammenfassung der Arbeit

In dieser Arbeit wurde untersucht, welches die charakteristischen Probleme der global verteilten Softwareentwicklung sind und wie sie sich mit agilen Methoden besser bewältigen lassen. Dazu wurde qualitative Forschung mit empirischen und konzeptionell-konstruktiven Forschungsmethoden angewendet. Der wissenschaftliche Forschungsstand wurde einbezogen und es wurde eine themenbezogene Konzeptarbeit geleistet. Es wurden eigene praktische Erfahrungen und Erkenntnisse aus drei Feldstudien analysiert. Insgesamt hat der Autor vier große Projekte direkt untersucht und zehn Interviews geführt.

Die Forschungsergebnisse dieser Arbeit wurden in drei Teilen beschrieben. In **Teil I: Praxis der global verteilten Softwareentwicklung**, bestehend aus den Kapiteln zwei und drei, wurden die Grundlagen der Arbeit gelegt, das zu bearbeitende Forschungsfeld festgelegt und die zu lösenden Probleme benannt. In **Teil II: Erarbeitung von Lösungskonzepten**, bestehend aus den Kapiteln vier und fünf, wurden die Probleme nach wissenschaftlichen Methoden analysiert. Mit dem Dual-Shore-Kooperationsmodell und architekturzentrierter Entwicklung wurden Lösungskonzepte in einer Feldstudie erarbeitet. Diese Lösungskonzepte wurden in **Teil III: Evaluierung der Ergebnisse** in Musterform dokumentiert und in zwei weiteren Feldstudien in der Praxis evaluiert. Dadurch konnte die Plausibilität der Ergebnisse nachgewiesen werden.

In den einzelnen Kapiteln wurden folgende Themen behandelt:

In **Kapitel 2** wurden verschiedene Ausprägungen der global verteilten Softwareentwicklung untersucht. Globale Softwareentwicklung zeichnet sich dadurch aus, dass sich die an der Entwicklung Beteiligten an verschiedenen Standorten befinden, die durch erhebliche räumliche Entfernungen getrennt sind. IT-Offshoring wurde als eine besondere Organisationsform der global verteilten Entwicklung identifiziert, bei der IT-Funktionen oder Geschäftsprozesse mit hohem IT-Anteil an Unternehmen in geografisch entfernte Niedriglohnländer vergeben werden.

Anhand einer Untersuchung von Erwartungen und Ergebnissen aus der Praxis konnten Besonderheiten sowie allgemeine Herausforderungen und Risiken der global verteilten Entwicklung aufgezeigt werden. Zu beachten sind insbesondere die geografische und zeitliche Verteilung der Teams sowie kulturelle Unterschiede bei den beteiligten Akteuren. Dadurch leidet oft die Kommunikation zwischen den Teams und es entstehen weitere Kosten. Ds führt dazu, dass Offshore-Projekte schwieriger zu meistern sind als nicht-verteilte Projekte. Viele Projekte sind gescheitert.

Aus dieser Untersuchung wurden wissenschaftliche Fragestellungen für verschiedene Forschungsbereiche und für die Softwaretechnik im Speziellen abgeleitet. Fast alle Fragen der Softwaretechnik sind für global verteilte Projekte wichtig, z. B. bezüglich der Arbeitsorganisation, der Abstimmung der Teams und des Wissenstransfers.

In **Kapitel 3** wurde betrachtet, wie agile Methoden in global verteilten Entwicklungsprojekten genutzt werden können, um die typischen Herausforderungen und Risiken verteilter Projekte anzugehen. Dazu wurde zunächst die Vereinbarkeit von Agilität und Verteilung diskutiert. Obwohl sich mehrere Grundannahmen von agilen Methoden und verteilten Entwicklungsprojekten unterscheiden, zeigte die konzeptionelle Untersuchung, dass agile global verteilte Entwicklung Chancen bietet, die Probleme von verteilten Projekten zu lösen.

Im Mittelpunkt der weiteren Untersuchungen stand eine systematische Untersuchung von veröffentlichten Erfahrungsberichten, um die Vereinbarkeit von Agilität und verteilter Entwicklung in der Praxis zu überprüfen. Dabei wurden relevante Themen der Praxis und dabei auftretende Probleme und Lösungsansätzen identifiziert. Aus den Ergebnissen wurde ersichtlich, dass für eine erfolgreiche Umsetzung von agiler global verteilter Entwicklung hauptsächlich zwei Fragen schlüssig zu beantworten sind: Zum einen muss geklärt werden, wie die verteilten Teams effizient miteinander kommunizieren können. Dies führte zur Untersuchung von Kooperationsmodellen in Kapitel 4. Zum anderen muss die Entwicklung so gesteuert werden können, dass die Flexibilität der Entwickler weitgehend erhalten bleibt, ihre Arbeiten aber trotzdem abgestimmt und kontrolliert werden können. Dies führte zur Untersuchung von architekturzentrierter Entwicklung in Kapitel 5.

Mit den Kapiteln 2 und 3 konnte somit die erste Forschungsfrage („Welche Themen sind in der Praxis besonders problematisch?“) für global verteilte Entwicklung allgemein und für agile global verteilte Entwicklung im Speziellen beantwortet werden.

Das Zusammenspiel zwischen Kommunikation, Koordination und Kooperation wurde in **Kapitel 4** betrachtet. Es wurde untersucht, wie die Zusammenarbeit zwischen Teams bei verteilter Entwicklung im Offshoring durch die Dimensionen der Verteilung und kulturelle Unterschiede beeinflusst wird. Zur genaueren Auswertung wurden drei Kooperationsmodelle betrachtet: Direktes Offshoring, Indirektes Offshoring und das Dual-Shore-Modell. Zum schnellen Vergleich von unterschiedlichen Kooperationsmodellen wurde eine grafische Notation entwickelt und eingeführt.

Für jedes Kooperationsmodell wurde die Kommunikation der Teams über verschiedene Kanäle detailliert untersucht. Eine Bewertung der Modelle anhand dieser Untersuchung und der in Kapitel 3 gewonnenen Erkenntnissen zu agiler global verteilter Entwicklung führte zu dem Ergebnis, dass die Zusammenarbeit nach dem Dual-Shore-Modell die besten Voraussetzungen für eine erfolgreiche Zusammenarbeit der verteilten Teams beim Agilen Offshoring bietet. Abschließend wurden Realisierungsalternativen des Dual-Shore-Modells diskutiert, z. B. die Einbeziehung eines Entwicklerteams des Kunden.

In **Kapitel 5** wurde untersucht, wie architekturzentrierte Entwicklung bei global verteilter Entwicklung eingesetzt werden kann, um die Softwareentwicklung zu steuern und qualitativ hochwertige Anwendungen zu produzieren. Dazu wurden zunächst unterschiedliche Aspekte von Softwarearchitektur betrachtet. Je nach Blickwinkel kann Softwarearchitektur unter den Metaphern „Blaupause“, „Literatur“, „Sprache“ oder „Entscheidung“ betrachtet werden. Es wurden auch weiterführende Konzepte im Umgang mit Softwarearchitektur wie Architektursichten und -stile sowie Architekturanalyse und -evolution eingeführt und diskutiert.

Bei einer architekturzentrierten Entwicklung steht die Softwarearchitektur im Mittelpunkt aller Aktivitäten. Architekturzentrierte Entwicklung wurde zunächst aus historischer Perspektive betrachtet, mit den zugrunde liegenden Arbeiten im Rahmen des RUP/UP und am SEI der Carnegie Mellon University. Anschließend wurde ein erweitertes Verständnis von architekturzentrierter Entwicklung unter Einbeziehung von Erkenntnissen aus den beiden Methoden STEPS und WAM formuliert. Bei diesem Verständnis wird ein flexibler Umgang mit Architektur gepflegt, bei dem diese permanent weiterentwickelt wird. Die Entwicklung der Architektur wird auch auf detaillierter Ebene unterstützt, u. a. mit konkreten Metaphern, Architekturvorstellungen und -regeln.

Die Anwendung von architekturzentrierter Entwicklung bei global verteilter Softwareentwicklung stand im Mittelpunkt der darauf folgenden Ausführungen. Zunächst wurde der Forschungsstand untersucht. Die Untersuchung zeigte, dass nur wenige wissenschaftliche Erkenntnisse vorliegen und dass insbesondere weitere empirische Forschung vonnöten ist. Daher wurde eine durch Action Research gesteuerte Feldstudie durchgeführt. In dieser wurden typische Aufgaben von Softwarearchitekten bei global verteilter Entwicklung identifiziert. Es wurde festgestellt, dass die wichtigsten Funktionen von Softwarearchitekten darin bestehen, fachliches und technisches Wissen zu verbinden, eine Balance zwischen starren Vorgaben und einer Eigenverantwortung der Entwickler zu finden und zwischen allen Beteiligten zu moderieren. Aus den Untersuchungen wurden Realisierungsempfehlungen für architekturzentriertes Arbeiten bei global verteilter Entwicklung abgeleitet.

Somit konnten mit den Kapiteln 4 und 5 Antworten auf die zweite Forschungsfrage („Mit welchen Mitteln kann der Einsatz von agilen Methoden unterstützt werden?“) gefunden werden.

In **Kapitel 6** wurden die erarbeiteten Ansätze in der Praxis evaluiert. Um die Konzepte handhabbar zu machen, wurden sie in Musterform dokumentiert. Es wurden die Muster „Dual-Shore-Modell“, „Architekturzentrierte Softwareentwicklung“ und „Component Tasks“ beschrieben.

Diese Muster wurden auf zwei weitere Feldstudien angewendet, die auf Interviews und der Auswertung von Projektdokumenten basierten. In beiden Feldstudien wurden wichtige Vorfälle identifiziert, die mit Zitaten aus den Interviews und Ausschnitten aus den Projektdokumenten belegt wurden. In diesen Vorfällen wurden die Einsatzmöglichkeiten der Muster untersucht. Dabei wurden die Dimensionen Akteure, Prozesse, Informationen und Technik berücksichtigt. In dieser Untersuchung konnte zum einen gezeigt werden, dass die Ansätze plausibel sind. Zum anderen konnten weitere konkrete Empfehlungen und Realisierungsalternativen für den Einsatz von agiler verteilter Entwicklung erarbeitet werden.

Dies beantwortet die dritte Forschungsfrage („Wie können Empfehlungen für eine erfolgreiche praktische Umsetzung dokumentiert und evaluiert werden?“).

7.2 Kritische Würdigung des Erreichten

In Abschnitt 2.4.2 wurde Betrand Meyer mit der Aussage zitiert, dass die Auswirkungen des Offshoring der Softwareentwicklung auf die Softwaretechnik noch zu wenig erforscht wurden. Mit den Untersuchungen in dieser Arbeit konnte diese Forschungslücke verringert und dadurch die Softwaretechnik-Forschung in einem wichtigen Feld vorangebracht werden.

Als wesentliche Erkenntnis dieser Arbeit ist festzuhalten, dass der Einsatz von agilen Methoden in global verteilten Projekten zur Entwicklung von Anwendungssoftware die Erfolgschancen solcher Vorhaben gegenüber konventionellen Vorgehensweisen bei global verteilten Entwicklungsprojekten erhöhen kann. Dies wurde durch einen stringenten Gang der Untersuchung begründet: Auf Basis einer Untersuchung der Praxis wurden allgemeine Herausforderungen der global verteilten Entwicklung und des Offshoring und wichtige Fragestellungen für die Softwaretechnik benannt. Die Untersuchung des Einsatzes von agilen Methoden bei global verteilter Entwicklung führte zu einem Katalog von Themen mit zugeordneten Problemen und Lösungsansätzen. Als entscheidende Faktoren für den Erfolg von global verteilten Entwicklungsprojekten mit agilen Methoden wurden daraufhin das Dual-Shore-Modell und architekturzentrierte Entwicklung identifiziert und in mehreren Feldstudien untersucht.

In dieser Arbeit wurden mehrere empirische Quellen zusammengeführt und mit qualitativen Forschungsmethoden ausgewertet. In den konstruktiven Teilen wurden Lösungsansätze für wichtige Probleme erarbeitet. Mit den eingesetzten qualitativen Forschungsmethoden sind prinzipiell keine Validierung und keine allzu weitreichende Verallgemeinerung der Ergebnisse möglich. Die vorgestellten Ansätze können jedoch

für Projekte mit ähnlichen Voraussetzungen empfohlen werden. Durch Konzeptarbeit und empirisch-konstruktive Untersuchungen konnten für solche Vorhaben konkrete Empfehlungen in Form von Mustern formuliert werden.

Als wichtige innovative Einzelergebnisse dieser Arbeit sollen die folgenden besonders hervorgehoben werden:

1. Es wurde ein umfassendes, konsistentes Begriffsmodell für den Themenkomplex global verteilte Entwicklung, IT-Offshoring sowie dem Einsatz von agilen Methoden in diesen Feldern erstellt. Darin wurden Begriffsbildungen aus der wissenschaftlichen Literatur und aus der Praxis integriert. Dies stellt einen wichtigen Schritt dar, da viele Begriffe in der Literatur bisher noch uneinheitlich verwendet werden.

2. Der Einsatz von agilen Methoden bei global verteilter Entwicklung wurde gründlich untersucht. Es konnte ein übersichtlicher Katalog mit Themen, Problemen und Lösungsansätzen aus der Praxis entwickelt werden, der über exemplarische Erfahrungsberichte hinausgeht. Dieser Katalog wurde unter Einbeziehung von weiteren wissenschaftlichen Erkenntnissen konsolidiert.

3. Das bestehende Konzept des Kooperationsmodells wurde für den Bereich der global verteilten Entwicklung eingeführt. So konnten in der Praxis anzutreffende Offshoring-Konstellationen und konkrete Projektkonstellationen systematisch untersucht werden. Zur Veranschaulichung wurde eine eigene Darstellungsform für Kooperationsmodelle entwickelt und eingesetzt.

4. Mit der architekturzentrierten Entwicklung wurde ein neues Thema in die Diskussion von global verteilter Entwicklung eingebracht. Der Umgang mit Softwarearchitektur im Entwicklungsprozess von global verteilten Projekten wurde ausführlich untersucht. Diese Arbeit geht damit über die bestehenden Forschungsergebnisse zum Nutzen von Softwarearchitektur bei verteilter Entwicklung hinaus und bietet zusätzlich Muster und Empfehlungen für den praktischen Einsatz.

7.3 Offene Fragen: ein Blick nach vorne

Um agile global verteilte Softwareentwicklungsprojekte noch besser zu verstehen, sollte die empirische Basis mit weiterführenden Forschungsarbeiten erweitert werden. Mit einer erweiterten Empiriebasis kann auch evaluiert werden, inwiefern die in dieser Arbeit vorgestellten Konzepte auf andere Projekte übertragen werden können und in welchem Typ von Projekten sie noch eingesetzt werden können. Interessante Fragen sind beispielsweise, wie gut die Ansätze die Skalierung von Projekten mit steigender Projektgröße und -komplexität unterstützen, und wie viele Mitarbeiter onshore nötig sind, um größere Teams mit Offshore-Entwicklern zu steuern.

Die Dokumentation der Ergebnisse in Form von Mustern bietet eine Basis für weitere Forschungstätigkeiten. Zum einen sollten die gefundenen Muster in weiteren Projekten eingesetzt und evaluiert werden. Zum anderen können sie, zusammen mit den Mustern von Braithwaite und Joyce [Braithwaite u. Joyce 2005b], den Grundstock für einen umfassenden Musterkatalog für agile global verteilte Entwicklung bilden.

In Teilbereichen kann auch weitere quantitative Forschung zu einem Erkenntnisgewinn beitragen, so aufwendig diese Art von Forschung bei global verteilten Entwicklungsprojekten auch sein mag. So erscheinen kontrollierte Experimente zu einzelnen Einflussfaktoren (z. B. zeitliche und geografische Entfernung, Anzahl und Größe der Teams) möglich und wünschenswert. So kann untersucht werden, wie der Projekterfolg am besten erreicht werden kann.

Ausgehend von einer geeigneten Softwarearchitektur und einem integrierten Komponentenmodell ergibt sich eine Vision, in der Komponenten eines Softwaresystems so spezifiziert und isoliert betrachtet werden können, dass es keine Rolle mehr spielt, wo auf der Welt und durch wen sie implementiert werden. Dieses Ziel hat der Autor dieser Arbeit schon in einem Konferenzbeitrag skizziert [Sauer 2006a]. Es ist nun detailliert zu untersuchen, welche organisatorischen und technischen Rahmenbedingungen für diese flexible Art der Zusammenarbeit geschaffen werden müssen und ob dies in der Praxis ein gangbarer Weg für die Entwicklung von fachlich komplexen Anwendungssystemen ist. Diese Arbeit bietet dazu eine gute Grundlage.

Aktuelle und zukünftige geopolitische Entwicklungen werfen weitere Fragen für die Forschung auf. In Abschnitt 4.4.5 wurde schon beschrieben, dass indische und andere Offshore-Dienstleister verstärkt Niederlassungen in Ländern typischer Offshoring-Kunden gründen und ihre Mitarbeiter mit Branchenwissen ausstatten. Wenn dieser Trend anhält, könnten Offshore-Dienstleister mit großen Entwicklungskapazitäten in ihren Heimatländern noch stärker in direkte Konkurrenz mit den einheimischen Dienstleistern treten (vgl. [Hamm 2006]). Es wäre lohnenswert zu untersuchen, welche Auswirkungen eine solche Entwicklung auf das Offshoring von Softwareentwicklungsprojekten hat. Der Blick auf Prognosen für die nächsten Jahrzehnte zeigt, dass eine Verschiebung der wirtschaftlichen Kräfteverhältnisse in absehbarer Zeit wahrscheinlich ist (vgl. [Wilson u. Purushothaman 2003]). Es ist zu erwarten, dass durch diese Entwicklung die globale Softwareentwicklung noch weiter zunehmen wird. Damit stellen sich neue Herausforderungen, gerade auch für die Softwaretechnik.

Die beschriebenen Entwicklungsperspektiven haben auch Einfluss auf die zukünftigen Aufgaben von deutschen Softwareentwicklern, -architekten und Projektmanagern. Schon heute sollten daher nach Ansicht des Autors dieser Arbeit und weiterer Forscher (vgl. z. B. [Hawthorne u. Perry 2005]) Studierende im Informatik- und Wirtschaftsinformatikstudium auf die verteilte Softwareentwicklung vorbereitet werden. Der in dieser Arbeit vorgestellte Ansatz kann eine Grundlage für diese Ausbildung sein. An der Universität Hamburg wurden Studierende im Haupt- bzw. Masterstudium in Kooperation mit der Universität von Alexandria (Ägypten) bereits mit den Schwierigkeiten von agilen Offshoring-Projekten vertraut gemacht. Weitere Lehrpläne und Aktivitäten in diesen

Bereich sollten angestoßen werden, um die Wettbewerbsfähigkeit des IT-Standorts Deutschland so weit wie möglich langfristig zu sichern.

Teil IV

Anhang

A Agile Methoden für die verteilte Entwicklung

In diesem Anhang werden die drei agilen Methoden Extreme Programming, Scrum und DSDM mit ihren Anpassungen für die verteilte Entwicklung vorgestellt. Bei allen Methoden wird zunächst die ursprüngliche Variante betrachtet und anschließend die angepasste Methode für die verteilte Entwicklung.

A.1 Extreme Programming

Mit der Publikation des ersten Buches zu Extreme Programming (XP) durch Kent Beck 1999: „Extreme Programming Explained: Embrace Change“ [Beck 1999] wurden agile Methoden einer breiten Öffentlichkeit bekannt gemacht. In der Folge entwickelte sich XP zu einer der wichtigsten Softwareentwicklungsmethoden. Im Jahr 2004 veröffentlichte Beck die zweite Auflage seines Buches mit einer grundlegenden Überarbeitung des Ansatzes [Beck u. Andres 2004].

A.1.1 Ursprüngliche Methode

Die erste Fassung von XP beruht auf vier Werten:

- offene Kommunikation zwischen allen Beteiligten,
- stetiges Feedback und konstruktive Kritik,
- Einfachheit von Prozessen und entwickelter Software und
- Mut, neue Wege zu gehen und Kritik anzunehmen.

In der zweiten Fassung kam als neuer Wert Respekt gegenüber allen Beteiligten hinzu, insbesondere zwischen Anwendern und Entwicklern und unter den Entwicklern.

Zur Umsetzung der Werte werden Prinzipien und Techniken eingesetzt. Die Prinzipien werden hier nicht behandelt. In der ersten Fassung von XP gibt es die folgenden zwölf Techniken (deutsche Übersetzung der englischen Originalbegriffe nach [Wolf et al. 2005]):

Planungsspiel: Kunden und Entwickler sitzen am Anfang jeder Iteration zusammen und erstellen (User) Storys, in denen funktionale Anforderungen an die Anwendung aus Akteurssicht festgehalten werden. Die Entwickler schätzen jeweils den Iterationsaufwand; die Kunden priorisieren die Anforderungen. Anschließend einigt man sich auf eine Menge von Storys, die in der nächsten Iteration umgesetzt werden sollen.

Kurze Releasezyklen: Die Entwickler stellen den Kunden nach jeder Iteration ein lauffähiges Release zur Verfügung. Die Kunden testen dieses und geben Feedback.

Kunde vor Ort: Ein Vertreter des Kunden sitzt bei den Entwicklern, beantwortet fachliche Fragen und trifft kleinere Entscheidungen.

Gemeinsame Verantwortlichkeit: Das Entwicklerteam ist für den gesamten Quelltext gemeinschaftlich verantwortlich. Jeder Entwickler kann jeden Quelltext ändern.

Fortlaufende Integration: Die Entwickler arbeiten auf einer gemeinsamen Quelltextbasis und testen und integrieren ihre Änderungen fortlaufend, spätestens nach einem Entwicklungstag.

Programmieren in Paaren: Die Entwickler arbeiten paarweise an einem Rechner, um die Qualität der erstellten Systeme zu verbessern. Die Paare wechseln regelmäßig.

Refactoring: Durch regelmäßige Vereinfachungen der Software ohne wahrnehmbare Änderung der Funktionalität werden Änderungen und Erweiterungen erleichtert.

Programmierstandards: Für alle Entwickler verbindliche Konventionen erleichtern den Paarwechsel und das Refactoring von fremdem Quelltext.

Einfaches Design: Anstelle komplizierter Entwürfe, die auf zukünftige Anforderungen ausgerichtet sind, soll besser eine schlichte, leicht verständliche Lösung gewählt werden.

Testen: Automatisierte Komponententests zu allen wichtigen Programmfunktionen schützen zu einem großen Teil davor, dass Änderungen bestehende Funktionalitäten beeinträchtigen.

Metapher: Die Anwendung wird auf eine Metapher aufgebaut, welche die grundlegenden Systemeinheiten und ihre Beziehungen verständlicher macht. Aus der Metapher sollte sich eine gemeinsame Sprache für das Projekt ableiten, auch für technische Elemente. Oft lässt sich die Anwendung auch sinnvoll durch eine *Menge* von Metaphern beschreiben.

Nachhaltiges Tempo: Das Projekt sollte so geplant und durchgeführt werden, dass ausreichend Zeit für Erholungspausen bleibt. Es ist ein schlechtes Zeichen, wenn regelmäßig Überstunden anfallen.

In der zweiten Fassung von XP wurden die Techniken nach den gesammelten Erfahrungen überarbeitet, präzisiert und vereinfacht. Sie sind jetzt in dreizehn Primär- und elf Folgetechniken aufgeteilt. Die Folgetechniken bauen auf den Primärtechniken auf; Kent Beck hält es für gefährlich, sie ohne die Vorarbeit der Primärtechniken einzusetzen [Beck u. Andres 2004].

Besonders interessant für diese Arbeit sind die Primärtechniken

Beieinander sitzen: Das gesamte Team sollte zusammensitzen, möglichst in einem Raum.

Echte Kundenbeteiligung: Die Entwickler sollten mit den tatsächlichen späteren Anwendern zusammenarbeiten, um deren Anforderungen möglichst gut umsetzen zu können.

Inkrementeller Entwurf: Die Anwendung sollte schrittweise anhand der nächsten Anforderungen entworfen werden, nicht im Voraus als großer Gesamtentwurf.

Testgetriebene Entwicklung: Die Komponententests sollten vor dem getesteten Quelltext geschrieben werden. Keine Funktionalität darf ohne zugehörigen Test eingecheckt werden.

sowie die Folgetechnik **Team-Kontinuität:** Entwickler sollten nur an einem Projekt arbeiten. Personalwechsel sollten vermieden werden.

A.1.2 Angepasste Methode

Distributed Extreme Programming (DXP) überträgt die Grundsätze von XP auf verteilte und nicht ortsgebundene Teams. Der Begriff geht zurück auf einen Artikel von Kircher, Jain, Corsaro und Levine aus dem Jahr 2002 und wird dort definiert als “Extreme Programming with certain relaxations on the requirements of close physical proximity of the team members” [Kircher et al. 2002, S. 555]. DXP ist nicht nur auf Offshoring, sondern auf jede Art der verteilten Arbeit ausgerichtet. Offshoring und damit eine Verteilung des Teams auf zwei oder mehr Kontinente wird jedoch als eines der Hauptanwendungsgebiete genannt. Die XP zugrunde liegenden Werte werden durch DXP nicht verändert. Genauso sieht es auch Kent Beck [Beck u. Andres 2004]. Er empfiehlt, der Feedback- und Kommunikationskultur besondere Aufmerksamkeit zu widmen, um fehlenden direkten Kontakt auszugleichen. Man sollte sich mehr anstrengen, einfache Lösungen umzusetzen, da es weniger Gelegenheiten gäbe, übermäßige Komplexität zu entdecken. Gegenseitiger Respekt sei bei unterschiedlichen Kulturen und Lebensweisen in verteilten Teams noch wichtiger als sonst.

Kircher et al. stellen fest, dass die Verteilung des Teams, das nicht mehr in einem Raum zusammensitzen kann, die größten Veränderungen bei DXP gegenüber dem Original-XP erforderlich macht. Davon betroffen sind die Techniken Planungsspiel,

Programmieren in Paaren, fortlaufende Integration und Kunde vor Ort. Die Autoren empfehlen folgende Anpassungen:

Planungsspiel: Das Treffen für das Planungsspiel findet virtuell statt. Die räumliche Distanz wird durch Videokonferenzsysteme oder gemeinsam genutzte Anwendungen, beispielsweise zum Schreiben von Story-Cards, unterstützt. Die Autoren geben keine Hinweise, wie mit unterschiedlichen Zeitzonen und Sprachen umgegangen werden soll.

Programmieren in Paaren: Ähnliche technische Maßnahmen können eingesetzt werden, um zusammen am Quelltext zu arbeiten, was als Remote Pair Programming (RPP) bezeichnet wird. RPP hat bereits viel Aufmerksamkeit gefunden, sodass dafür maßgeschneiderte Anwendungen entwickelt und evaluiert wurden. Umfangreiche empirische Untersuchungen zur Effektivität von RPP stehen noch aus. Erste Ergebnisse sind jedoch positiv (vgl. z. B. [Flor 2006], [Hanks 2005]).

Fortlaufende Integration: Die fortlaufende Integration ist nur dann ein Problem, wenn ein eigenständiger Integrationsrechner verwendet werden soll, um die neu entwickelten Quelldateien in das gemeinsame Archiv hochzuladen (vgl. [Fowler 2006a]). Alternativ können die Quelldateien direkt von den Entwicklungsrechnern synchronisiert werden. Die Arbeit mit einem gemeinsamen Archiv ist dann mit entsprechend entworfenen Versionskontrollsystemen möglich.

Kunde vor Ort: Es ist in der Regel nicht möglich, einen Vertreter des Kunden für längere Zeit zum Offshore-Team zu entsenden (vgl. auch [Martin et al. 2004], [Wallace et al. 2002]). Als Alternative sieht DXP Videokonferenzen mit dem Kunden vor, der dann zum „virtuellen Kunden vor Ort“ wird.

Die Autoren sind sich jedoch im Klaren darüber, dass alle diese Maßnahmen physische Nähe und direkte menschliche Kommunikation nicht vollständig ersetzen können. Dennoch führen Paasivaara und Lassenius in [Paasivaara u. Lassenius 2006] Projekte an, in denen gerade mit DXP Kunde und Offshore-Entwickler enger miteinander arbeiten und somit Verzögerungen vermeiden und die Güte der Kommunikation steigern konnten.

Braithwaite und Joyce haben die Arbeit von Kircher et al. aufgegriffen und Erweiterungen vorgeschlagen [Braithwaite u. Joyce 2005a]. Hierzu zählen Botschafter, die für längere Zeit beim anderen Team arbeiten und somit Wissen verbreiten und gegenseitiges Verständnis unter den Teams wecken; wechselseitige Besuche, um Vorurteile abzubauen und sich gegenseitig kennenzulernen; verteilte Standup-Meetings per Videokonferenz zur Abstimmung; und Wikis als gemeinsame virtuelle Räume, mit denen Informationen schnell und für alle asynchron zugreifbar erfasst werden können.

A.2 Scrum

Der Name Scrum kommt von der englischen Bezeichnung für das Gedränge im Rugby-Sport. Genauso wie das Team beim Rugby den nächsten Spielzug bespricht, treffen sich die Projektbeteiligten in täglichen Standup-Meetings („Daily Scrum“) und besprechen die nächsten Entwicklungsschritte. In diesen etwa 15-minütigen Treffen geht es um die Fragen „Was habe ich seit dem letzten Treffen geschafft?“, „Was möchte ich bis zum nächsten Treffen erledigen?“ und „Auf welche Hindernisse stoße ich dabei?“ 1993 wurden durch Jeff Sutherland erste Scrum-Projekte in der Easel Corporation durchgeführt.

A.2.1 Ursprüngliche Methode

Scrum kennt drei Rollen [Schwaber u. Beedle 2002], [Schwaber 2004]: Das **Team** entwickelt die Funktionalität. Neben Entwicklern gehören dazu ggf. auch Tester und andere Spezialisten. Das Team organisiert sich selbst. Der **Scrum Master** ist für die Umsetzung des Scrum-Prozesses verantwortlich. Er sorgt dafür, dass das Team ungestört arbeiten kann und die Grundregeln von Scrum beachtet. Er ist damit der Methodenfachmann, kein Projektleiter. Der **Product Owner** vertritt den Auftraggeber. Er legt fest, welche fachlichen Anforderungen umgesetzt werden.

Die Entwicklung erfolgt in Sprints von jeweils 30 Tagen. In einigen Projekten werden auch kürzere Sprintdauern gewählt. Die Dauer soll über die Laufzeit des Projekts konstant bleiben. Der Product Owner verwaltet die Anforderungen an die Software im Product Backlog. Sie können von allen Beteiligten aufgrund gesammelter Erfahrungen jederzeit geändert werden. Zu Beginn jedes Sprints wird im Sprint Planning Meeting festgelegt, welche Anforderungen umgesetzt werden sollen. Der Product Owner priorisiert sie und das Team schätzt den Aufwand. Die Anforderungen für den jeweiligen Sprint werden im Sprint Backlog in Form einzelner Tasks festgehalten. Das konkrete Vorgehen bei der Implementierung wird durch Scrum nicht vorgegeben. Das Team ist selbstverantwortlich für die Aufgabenteilung und die Entwurfs- und Programmiermethode.

Täglich aktualisierte Feature-Burndown-Charts dokumentieren den Entwicklungsverlauf und ermöglichen eine Schätzung des Fortschrittsgrads des Sprints. Am Ende jedes Sprints steht ein Sprint Review, bei dem die Ergebnisse anhand laufender Software präsentiert und diskutiert werden. Es folgt jeweils ein Sprint Retrospective Meeting zur Diskussion über den Verlauf des Sprints, um die nächsten Sprints zu verbessern. Umgesetzte Anforderungen werden aus dem Product Backlog entfernt.

Es ist möglich, Techniken aus Scrum und XP zu kombinieren. Beispielsweise können Standup-Meetings auch ergänzend bei XP genutzt werden. XP-Techniken können bei der Entwicklung in Scrum-Sprints eingesetzt werden.

A.2.2 Angepasste Methode

Ken Schwaber geht in [Schwaber 2004] auf große Projekte mit mehreren Scrum-Teams ein, die auch global verteilt sein können. Die einzelnen Teams haben dabei eine Stärke von etwa sieben bis neun Personen. Die Abstimmung erfolgt in einem gemeinsamen täglichen Scrum Meeting mittels Telefon- oder Videokonferenz. Wenn dies aufgrund der Zeitdifferenz oder anderer Faktoren (z. B. der Sprache) nicht umsetzbar ist, können sich alternativ die Scrum Master der einzelnen Scrum-Teams in einem Scrum of Scrums treffen. Ein Chief Scrum Master koordiniert alle Teams. Genauso sinnvoll ist ein Chief Product Owner und ein tägliches, ggf. nur virtuelles Abstimmungstreffen der Product Owner aller Teams.

Eine wichtige Voraussetzung von Distributed Scrum ist, dass alle an derselben Programmversion arbeiten. Zumindest täglich sollte das System neu erstellt werden. Wie beim herkömmlichen Scrum soll jedes Teammitglied über alle Informationen zum Projektfortschritt und zu Problemen verfügen. Für vollständige Transparenz ist gesorgt, wenn die Sprint Backlogs, Projektmetriken und andere Dokumente aller Teams für jeden einsehbar sind.

Jeff Sutherland et al. haben in [Sutherland et al. 2007b] aus der Praxis drei Modelle des verteilten Scrum für Agiles Offshoring identifiziert. Das soeben beschriebene Vorgehen bezeichnen sie als **Distributed Scrum of Scrums**. Wenn nur das Onshore- und nicht das Offshore-Team Scrum nutzt, spricht man von **Isolated Scrums**. Dies kann beispielsweise dann der Fall sein, wenn das Management des Offshore-Dienstleisters keine agilen Prozesse einsetzen möchte. Beim **Totally Integrated Scrum**-Modell setzen alle Teams Scrum ein. Im Gegensatz zum Distributed Scrum of Scrums sind dabei die Mitglieder eines Teams auf viele Orte verteilt. Dadurch soll die Transparenz und Produktivität der Teams und des Gesamtprojekts steigen.

Nach [Paasivaara u. Lassenius 2006] gibt es momentan noch wenig Erfahrung mit Distributed Scrum. Erste Erfahrungen zeigten jedoch, dass die Scrum-Prinzipien erfolgreich eingesetzt werden können, wenn es gelingt, eine häufige und effiziente Kommunikation zwischen den Teams zu etablieren.

A.3 DSDM

A.3.1 Ursprüngliche Methode

Die Dynamic Systems Development Method wurde im Februar 1995 veröffentlicht. Sie wird vom DSDM Consortium gepflegt, einer gemeinnützigen, herstellerunabhängigen Organisation. DSDM spielt vor allem in Großbritannien eine große Rolle, wo es auch hauptsächlich entwickelt wurde [Stapleton 2003]. In Deutschland ist es kaum verbreitet, daher werden in dieser Arbeit auch die englischen Begriffe verwendet. Die aktuelle Version 4.2 des DSDM-Rahmenwerks ist öffentlich verfügbar [DSDM Consortium 2010].

Eine kostenpflichtige Mitgliedschaft im Konsortium ist nur erforderlich, wenn Produkte und Dienstleistungen zu DSDM kommerziell vertrieben werden sollen.

DSDM verfolgt einen flexiblen, iterativ-inkrementellen Ansatz, mit dem die Akteure in ihrer Zusammenarbeit unterstützt werden sollen, um ihre Geschäftsziele zu erreichen. DSDM ist so technik- und werkzeugunabhängig, dass es nicht nur für Softwareentwicklungsprojekte eingesetzt werden kann, obwohl darauf das Hauptaugenmerk liegt. DSDM beruht auf neun Prinzipien:

1. Benutzer sollen sich aktiv beteiligen.
2. Das Team muss befähigt werden, Entscheidungen zu treffen.
3. Es werden regelmäßige Auslieferungen angestrebt.
4. Das Hauptkriterium für die Akzeptanz liegt darin, Funktionalität auszuliefern, die für die Anwender einen Geschäftswert darstellt.
5. Iterative, inkrementelle Entwicklung ist unverzichtbar.
6. Alle im Projektlebenszyklus vorgenommenen Änderungen können zurückgenommen werden.
7. Anforderungen werden auf einer hohen Ebene beschrieben.
8. Testen ist ein integraler Bestandteil des gesamten Projektlebenszyklus'.
9. Kollaboration und Kooperation zwischen allen Beteiligten sind zwingend erforderlich.

DSDM besteht aus den drei Phasen: Pre-Project, Project Lifecycle und Post Project. Die mittlere Hauptphase ist fünfstufig: Feasibility Study (Klärung der Erfolgsaussichten), Business Study (Fokus des Projekts, wesentliche Anforderungen und Beteiligte), Functional Model Iteration (detaillierte Spezifikation, Prototypen), Design and Build Iteration (Entwicklung und Test), Implementation (Übergang in den operativen Betrieb). Die Stufen drei bis fünf werden iterativ durchlaufen. DSDM besitzt ein ausgefeiltes Rollenkonzept. So sind z. B. der Ambassador User (Kunde mit Entscheidungsbefugnis) und der Advisor User (späterer Anwender der Software) mit dem Kunden vor Ort von XP vergleichbar. Im Gegensatz zu Scrum gibt es explizite Team Leader. Der Project Manager ist für das gesamte Projekt verantwortlich.

Zum Management der Entwicklung gibt es zwei Kerntechniken. Mit **Timeboxing** werden Dauer und Ziel jeder Iteration festgelegt. Der Endtermin muss eingehalten werden. Notfalls müssen weniger wichtige Anforderungen in die nächste Iteration verschoben werden. Zur Priorisierung von Anforderungen wird das **MoSCow-Prinzip** verwendet, bei dem Anforderungen in die Kategorien „must have“, „should have“, „could have“ und „Want to have but Won't have this time“ eingeteilt werden.

A.3.2 Angepasste Methode

Die DSDM and Offshore Software Development Task Group hat Richtlinien für den Einsatz von DSDM in Offshoring-Projekten entwickelt, was als DSDM-O bezeichnet wird [DSDM Consortium 2005]. Darin werden Hinweise für kritische Bereiche gegeben, die bei verteilter Entwicklung besondere Sorgfalt und Beachtung erfordern. Diese umfassen

- einen partnerschaftlichen Umgang mit Verträgen, der eine kooperative und konstruktive Beziehung fördert,
- die Etablierung einer offenen Kommunikation und von gegenseitigem Verständnis und
- eine abgestimmte Qualitätssicherung und Fortschrittsüberprüfung.

Bei DSDM-O müssen das Onshore- und das Offshore-Team DSDM einsetzen. DSDM-O enthält Techniken, um die Zusammenarbeit der Teams zu erleichtern. Da die Offshore-Entwickler anfangs meist Probleme haben, fachliche Zusammenhänge zu verstehen, betrifft dies vor allem den Ambassador User und die Offshore-Entwickler. Die Pre-Project und Post Project Phasen werden für DSDM-O kaum modifiziert. Im Project Lifecycle ergeben sich größere Änderungen:

Feasibility Study: In dieser Phase fällt die Entscheidung, ob Offshoring genutzt werden soll. Falls ja, werden spezifische Offshoring-Risiken festgehalten und die benötigte Kommunikations- und Entwicklungsinfrastruktur geplant.

Business Study: Der Development Plan berücksichtigt das Offshore-Team. Es wird festgelegt, welche Anforderungen offshore umgesetzt werden sollen.

Functional Model Iteration: In dieser Phase arbeiten Onshore- und Offshore-Team zum ersten Mal zusammen. Es finden persönliche Treffen sowie Telefon- und Videokonferenzen statt.

Design and Build Iteration: Onshore- und Offshore-Team arbeiten zusammen an der Produkterstellung. Dabei nutzen sie Kommunikations- und Unterstützungswerkzeuge. Es finden weiterhin auch persönliche Treffen sowie Telefon- und Videokonferenzen statt.

Implementation: In dieser Phase sollten sich keine größeren Änderungen gegenüber konventionellem DSDM ergeben. Es könnte jedoch sein, dass technische Unstimmigkeiten trotz Tests auf der Onshore-Seite erst jetzt sichtbar werden und behoben werden müssen.

Die DSDM-Rollen werden um zusätzliche Aufgaben erweitert. So liegt es z. B. in der Verantwortung des Project Manager, dafür zu sorgen, dass alle einem gemeinsamen Plan folgen. Die Anforderungen an das Produkt müssen allen Teammitgliedern klar sein. Sie sollten elektronisch festgehalten werden und für jeden abrufbar sein. Es gibt häufige

Releases, damit der Ambassador User die Weiterentwicklung sieht und der Project Manager den Fortschritt einschätzen kann. Dazu kommt ein häufiges und intensives Testen. Die Testerrolle ist aufgespalten in einen Onshore- und einen Offshore-Tester und eventuell einen zusätzlichen Test Coordinator. Weitere neue Rollen umfassen u. a. einen Ambassador Developer. Als Mitglied des Onshore-Teams vertritt er das Offshore-Team und kommuniziert mit diesem direkt. Sein Gegenstück ist der Onshore Ambassador User, der im Offshore-Team arbeitet und hauptsächlich für fachliche Fragen zuständig ist.

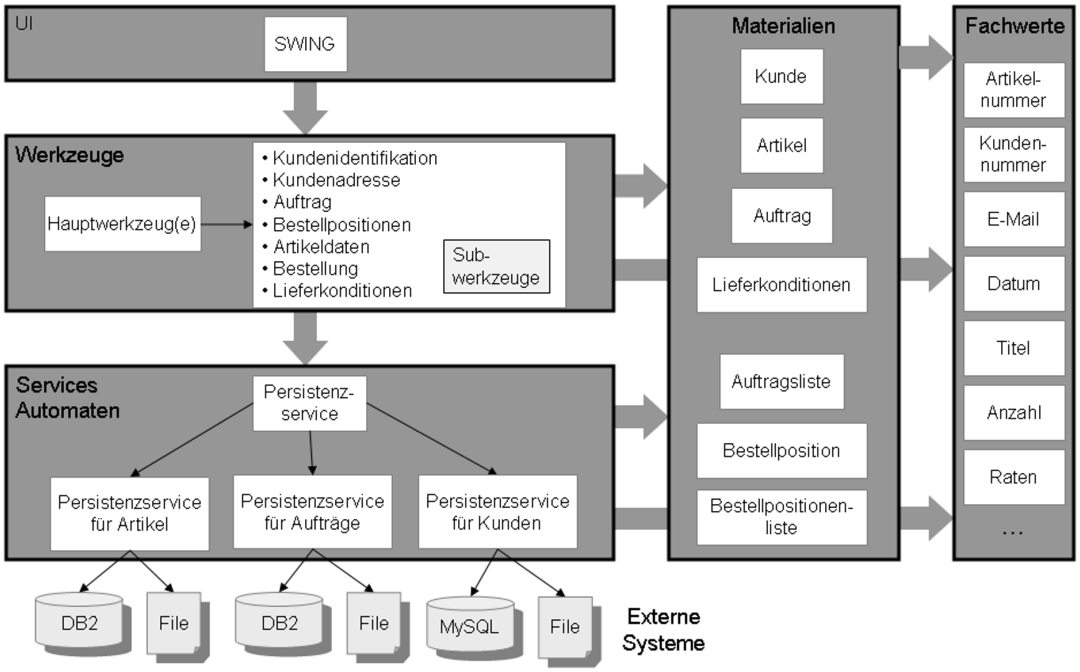
B Material zum Empirieprojekt Alpha

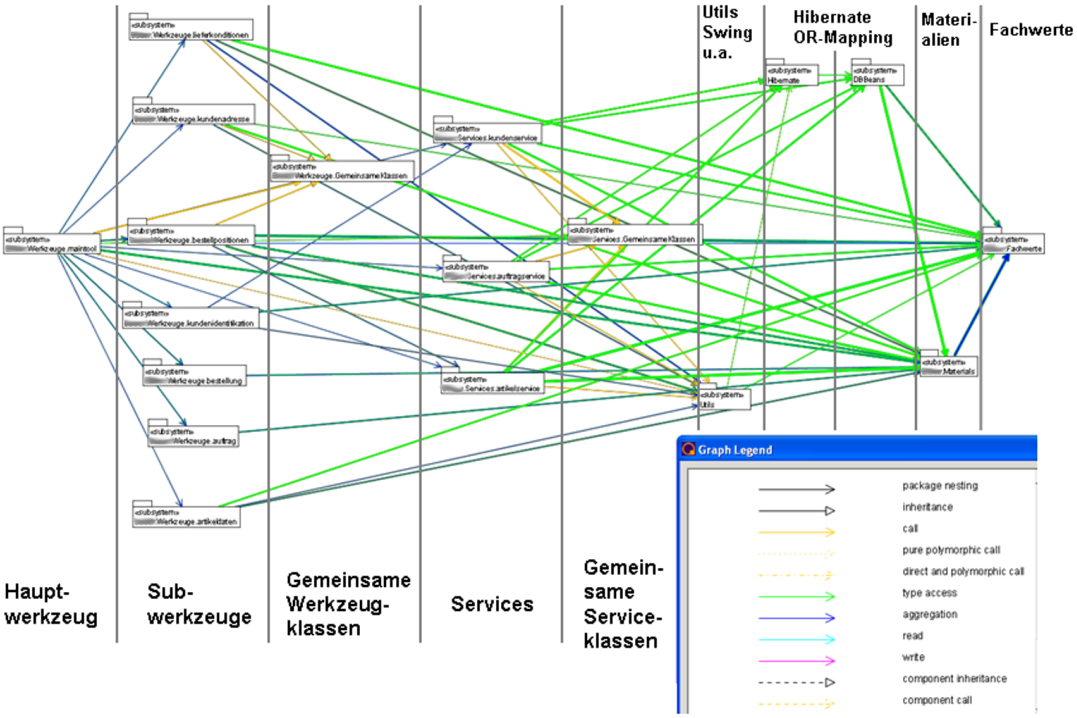
B.1 Statische Architektur

Die Architektur wurde in erster Linie durch die statische Sicht für alle Beteiligte veranschaulicht. Hier werden drei typische Dokumente abgedruckt: eine Beschreibung des grundsätzlichen Aufbaus der Architektur, eine grobe Übersicht über die Architektur und eine mit dem Architekturüberwachungswerkzeug Sotograph erzeugte Darstellung der Ist-Architektur des finalen Prototyps mit den Subsystemen und ihren Beziehungen (vgl. Abschnitt 5.1.2).

Die Modellarchitektur von WAM wurde für den Prototyp erweitert. Es wird eine Schichtenarchitektur verwendet. Die oberste Schicht besteht aus dem Hauptwerkzeug. Dieses erzeugt und verwaltet alle Subwerkzeuge. Die Subwerkzeuge kennen weder das Hauptwerkzeug noch sich gegenseitig. Die Kommunikation in Richtung Hauptwerkzeug findet über Events und Requests statt. Eine darunterliegende Schicht fasst gemeinsame Klassen aller Werkzeuge zusammen. Die nächste Schicht besteht aus den Services. Es gibt drei Services: für Kunden, für Artikel und für Aufträge. Für jeden dieser Services gibt es eine dateibasierte Implementierung und eine Datenbankversion. Die nächste Schicht beinhaltet Utilitys für Swing und die Konfiguration. Sie kann von allen darüberliegenden Schichten verwendet werden und verwendet selber Materialien und Fachwerte. Darunter liegt der OR-Mapper Hibernate mit den verwendeten Beans. Materialien bilden die zweitunterste Schicht. Die unterste Schicht stellen Fachwerte dar, die von vielen anderen Klassen benutzt werden.

Die Einhaltung der Regeln dieser Softwarearchitektur im Prototyp wird überwacht. Hauptsächlich wird dazu das Werkzeug Sotograph verwendet.


UI
SWING
Werkzeuge
Hauptwerkzeug(e)
• Kundenidentifikation
• Kundenadresse
• Auftrag
• Bestellpositionen
• Artikeldaten
• Bestellung
• Lieferkonditionen
Sub-
werkzeuge
Services
Automaten
Persistenz-
service
Persistenzservice
für Artikel
Persistenzservice
für Aufträge
Persistenzservice
für Kunden
DB2
File
DB2
File
MySQL
File
Externe
Systeme
Materialien
Kunde
Artikel
Auftrag
Lieferkonditionen
Auftragsliste
Bestellposition
Bestellpositionen-
liste
Fachwerte
Artikel-
nummer
Kunden-
nummer
E-Mail
Datum
Titel
Anzahl
Raten
...


Utils
Swing
u.a.
Hibernate
OR-Mapping
Materi-
alien
Fachwerte
Graph Legend
package nesting
inheritance
call
pure polymorphic call
direct and polymorphic call
type access
aggregation
read
write
component inheritance
component call
Haupt-
werkzeug
Sub-
werkzeuge
Gemeinsame
Werkzeug-
klassen
Services
Gemein-
same
Service-
klassen

B.2 Code Review

Der folgende Text enthält ein Code Review, das der Onshore-Softwarearchitekt zu Komponenten angefertigt hat, die vom Offshore-Dienstleister entwickelt worden waren. Das ursprüngliche Code Review ist ***fett*** *gedruckt. Anmerkungen des Offshore-Teams zum Review sind* kursiv *gedruckt. Antworten des Softwarearchitekten sind in der Standardschrift gedruckt.*

General

- **mind the codestyles, e. g. use brackets after if-statements**
- **order methods and attributes of a class (public at the top, private at the bottom), you should use the contextmenu "sort members" of a class when the templates are installed**
  We had run "sort members" and format on each file.
  Maybe you did not import the order of the members into Eclipse. The order is: Types, Static Initializers, Static Methods, Static Fields, Initializers, Constructors, Methods, Fields; visibility: public, private, protected, default.
- **use self-explanatory names for variables**
- **classes shouldn't contain unused variables (e. g. Lieferkonditionen-GUI), there are compile-warnings – so they are easy to recognize**
- **don't start local variables with "_", use it only for instance variables**
  The document "P2 Source conventions for Java" does not mention any difference between local and instance variables.
  I am not sure which version of the document you have. In my copy, it is paragraph 2.8. Also see the examples.

DV and Materials

- **overwrite the methods hashCode() and equals()**
  Had already done in each class. Can you please mention where is it missing?
  The methods should always be overriden in domain values and check for value equality.
- **RatenDV contains the value "13" twice, the test RatenDV_Test doesn't fail because tests for some values are lacking**
- **use reasonable default values, for GeldDV "0,00" is reasonable not "12,34"**
  Now onwards we will follow this.

- **why is there an assertion that 0 isn't a valid value for GeldDV?**
- **the String in GeldDV isn't evaluated, the implementation of the method isValid(String) is unuseful because the method geldDV(String) is missing**
- **not every attribute in Lieferkonditionen is initialised, the test Lieferkonditionen_Test is insufficient**
  Can you please clarify what else is expected in the Test. According to us, the test will become too large if we test for all the fields.
  Check for correct initialisation of all values after calling the constructor. Then check for correct setting of values by calling and testing the getters after the setters.

Tool

- **LieferkonditionenTool uses the class LiferkonditionenTool_Test**
- **why is the material Lieferkonditionen not used in LieferkonditionenTool?**
  As per the telecon, we were told not to populate Lieferkonditionen material.
  Yes, because we ran out of time. This is one of your new tasks for this iteration.
- **why is there the method getNewTestObject() in LieferkonditionenTool?**
- **unused labels in LieferkonditionenGUI**
- **there is a reference in LieferkonditionenGUI to LieferkonditionenGUI**

Test

- **the test GeldDVStringBasedFactory_Test doesn't work**
- **the test LieferkonditiontionenTool_Test doesn't work. It implements methods for a supertype-test, which isn't necessary, test of getter/setter for gutschein isn't implemented**
  Tests of getter/setters are not present in previous code.
- **be sure that assertions are enabled**
  Could you please elaborate?
  You have to add –ea to the default VM arguments for the JDK that you use in the preferences.
- **tests must be added to the class AllTests, to be sure that all tests are executed, you can use the class AllTest_automated**
  Added

- **if you use isEqual(), you mustn't test the opposite afterwards (e. g. see KonditionenDV_Test)**
  We have implemented as per KonditionenDV_Test.
  The logic behind this is that if you can test for the right value, you do not have to test for wrong values.
- **AnforderunghauptKatalogDV, Auftraggeprueft – testHashCode() are wrong, e. g. assertFalse(AnforderunghauptkatalogDV.J.hashCode() == AnforderunghauptkatalogDV.J.hashCode()) does not make sense. It should read assertTrue().**

B.3 Akzeptanztest

Der folgende Auszug zeigt beispielhaft einen Akzeptanztest für das Werkzeug „Auftragspositionen". Dieser wurde manuell nach größeren Änderungen durchgeführt.

- Eingabe 123456 → Text rot, rotes Kreuz auf Tab, Button 1 ausgegraut, Button Neu ausgegraut, Button Lösche ausgegraut, Bezeichnung leer
- Eingabe 1 → Text schwarz, grüner Haken auf Tab, Button 1 aktiviert, Button Neu aktiviert, Button Lösche ausgegraut, Bezeichnung: Body
- Eingabe 10, <Tab> in Feld Anzahl → Warnhinweis, Cursor in Anzahl
- Eingabe <Tab> → kein Warnhinweis
- Eingabe <ALT>+1 → Sprung zu Artikeldaten, Bild und Daten werden angezeigt
- Eingabe <ALT>+P → Sprung zu Auftragspositionen
- Eingabe <ALT>+N → neue Auftragsposition, Button 2 ausgegraut, Button neu ausgegraut, Button Lösche aktiviert, Cursor in Artikelnr. 2
- Eingabe <Tab>, 5 (in Feld Größe), <Tab> → Auftragsnummer 1234561 wird übernommen, alle Button aktiviert
- Eingabe <ALT>+L → Dialog Löschen, 2 in Eingabezeile
- Eingabe x, <RETURN> → Fehlerdialog ungültig, danach 2 in Eingabezeile
- Eingabe 5, <RETURN> → Fehlerdialog zu groß, danach 2 in Eingabezeile
- Eingabe -1, <RETURN> → Fehlerdialog > 0, danach 2 in Eingabezeile
- Eingabe 1, <RETURN> → Zeile 2 gelöscht, d. h. Zeile 1 mit Größe 5, Body in Bezeichnung, Button 1 aktiviert, Button Neu aktiviert, Button Lösche deaktiviert
- (Erzeugen vieler Auftragspositionen → Scrollbar erscheint, scrollt bei Erzeugen neuer Auftragsposition mit nach unten)

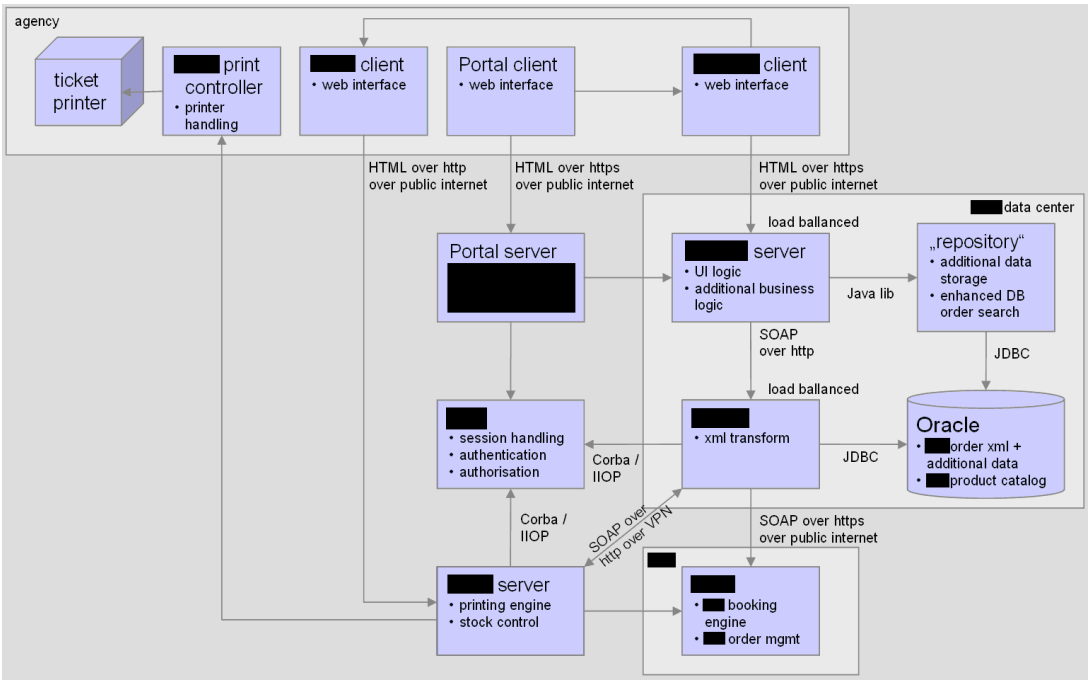
C Material zum Empirieprojekt Beta

C.1 Statische Architektur

Die Architektur wurde auch bei diesem Projekt in erster Linie durch die statische Sicht für alle Beteiligte veranschaulicht. Die grafische Dokumentation der statischen Architektur ist hier mit einem kurzen Erläuterungstext abgedruckt. Die Abbildung und der Text wurden durch den Autor dieser Arbeit anonymisiert.

Das System ist in die sehr komplexe, historisch gewachsene Systemlandschaft eingebettet. Architektonisch lässt sich diese Systemlandschaft sehr grob vereinfacht als eine Reihe von Buchungssystemen beschreiben, die sich um teils gemeinsam genutzte Infrastruktur-Komponenten, den Host, sowie diverse Buchungsysteme einzelner Anbieter gruppieren. Ein Merkmal der Systemlandschaft ist, dass zahlreiche Systeme nicht im Firmenkontext entstanden sind, sondern Entwicklungen zugekaufter Firmen sind, die nachträglich integriert wurden. Das System ist Teil eines ursprünglich zugekauften aber inzwischen integrierten Portals.

Ein weiteres Merkmal ist, dass die Buchungssysteme der Anbieter untereinander nicht kompatibel sind, d. h. an ihren Schnittstellen individuelle Protokolle und Datenstrukturen vorgeben. Diese Schnittstellen zu vereinheitlichen wird in verschiedenen, ihrerseits konkurrierenden Ansätzen verfolgt und gelingt unterschiedlich gut. Im Rahmen des Systems wurde versucht, sich an dem Standard der OTA (Open Travel Alliance) zu orientieren. In der Architektur wird daher auf das Buchungssystem nicht direkt zugegriffen, sondern über eine weitere Komponente.



C.2 AUP-Selbsteinschätzung

In der Selbsteinschätzung zur Nutzung von agilen Methoden werden zwei Ergebnisse, vom 15. Februar und vom 22. Juni 2006, gegenübergestellt. Das Dokument umfasst im Original 19 Seiten. Hier werden wichtige agile Techniken auszugsweise dargestellt.

Use the following ratings to indicate the extent to which each statement applies to your team:

1 = Disagree Completely (e. g. statement is 0% true)

2 = Disagree Somewhat (e. g. statement is 25% true)

3 = Neutral (e. g. statement is 50% true)

4 = Agree Somewhat (e. g. statement is 75% true)

5 = Agree Completely (e. g. statement is 100% true)

...

Unit Testing (TDD)

Feb 15	June 22	
1	2	There are enough automated unit tests covering your code.
1	3	Your code coverage increases with every Iteration.
1	3	All unit tests are run and pass when a task is finished and before checking in/integrating.
5	5	There is an automated way to run all unit tests.
5	5	Every unit test has an assertion.
5	5	Each unit test is completely independent of other unit tests.
5	5	Automated unit tests are run as part of your CI environment.
1	1	Code is only written in the presence of a failing unit test.
1	1	A coverage tool is used to measure unit test coverage.
1	1	All code is written through Test Driven Development (TDD).
1	2	When fixing bugs unit tests are used to capture the bug before fixing it.
1	5	Unit tests run fast enough to run all the time.
5	5	Mock objects are used to speed up unit tests.
5	5	The design supports the creation of new unit tests easily.
1	1	Unit tests are refactored.
		Unit Tests Comments: We have a lot of work to do here as we don't use TDD at present.

Acceptance Testing

Feb 15	June 22	
1	3	There are enough acceptance tests (manual or automated) covering your stories.
1	4	All acceptance tests are written before coding begins.
1	1	All acceptance tests that can be are automated.
3	4	The Customer Team works with Testers and Developers to define Acceptance Tests for each Story (before coding begins).
1	5	A Story isn't complete until its acceptance tests pass.
1	5	Acceptance tests are written in a language that the Customer Team can understand.
1	1	Automated acceptance tests are run frequently.
?	1	Automated acceptance tests can be easily run in any environment (dev, cert, etc).
1	1	A Story isn't complete until the accumulation of all implemented Stories acceptance tests pass (full regression).
1	1	A Story isn't complete until its acceptance tests are automated.

1	4	Written acceptance tests follow simple design rules.
?	?	The design supports the automation of new acceptance tests easily.
1	?	The Customer Team writes the test specification that can be used directly by an automated testing tool.
?	5	Automated acceptance tests can be run in any order.
1	1	Automated acceptance tests are run as part of your CI environment.
3	1	Bugs are illustrated in the form of failing acceptance tests.
		Acceptance Tests Comments: This is an area we are currently focusing on. We will have our first automated acceptance test this week but will have a lot of catch up work to automate the rest of our tests.

...

Refactoring

Feb 15 June 22

2	2	Duplication in code is removed systematically.
1	1	The team uses the Martin Fowler book "Refactoring; Improving the Design of Existing Code".
4	4	The team has the knowledge and skill to refactor effectively.
1	1	The team refactors all the time.
1	2	There are enough automated Unit Tests to safely refactor any code.
1	1	There are enough automated Acceptance Tests to safely refactor any code.
5	2	Tools are used to help detect places to refactor.
5	1	Tools are used to help enable easy refactoring (e. g. IntelliJ, Visual Slick Edit).
4	5	The source control system makes it easy to do big refactorings.
5	5	Any code is open to refactoring.
5	4	Refactorings are done in the context of a task (no speculative refactoring or premature patterns insertion).
5	1	Future refactorings have been identified.
?	1	Long-term refactorings are going on now.
		Refactoring Comments: We don't really refactor at present – needs focus.

Simple Design

Feb 15	June 22	
1	1	The team follows the Simple Design rules (In order of priority: Runs all the tests. No duplication. Express all ideas you want to express. Minimize number of classes and methods).
1	2	The team follows the rules: make it run, make it right, make it fast.
1	1	The team follows the Bob Martin (Uncle Bob) principles of good design (SOLID).
1	2	Every developer on the team has read Uncle Bob's book, "Agile Software Development"
4	4	The team has good programming language and tools skills.
1	1	The principles of good Class Design are used (Uncle Bob's principles).
1	1	The principles of good Package Design are used (Uncle Bob's principles).
1	2	The team refactors to keep the design clean.
2	1	The team continuously refactors (the team does not need to schedule refactoring tasks).
1	1	Patterns are never introduced before they fully emerge through Refactoring (patterns as design language).
1	1	Design does the simplest thing that could possibly work.
1	1	Time spent on initial design for a Story is much less than its implementation.
1	1	Unit tests are your design documentation.
1	1	Comments are not used to explain your code, rather the code expresses its intentions.
3	1	Methods are short, concise, and take on one responsibility.
2	1	There is no unused code or commented out code.
1	?	No piece of code exists that is over a cyclomatic complexity of 7.
3	1	An automated tool is used to measure code design quality.
1	1	Measuring code design quality is a part of your CI environment.
		Simple Design Comments: Again an area for future focus – we need to measure our cyclomatic complexity and look at code reviews etc.

...

D Material zum Empirieprojekt Gamma

D.1 Interviewleitfaden

Als ein Beispiel für einen bei der empirischen Untersuchung der Feldstudien verwendeten Interviewleitfaden ist hier derjenige aus dem Interview mit dem Softwarearchitekten abgedruckt.

1. Eigene Rolle im Projekt
   a) Zusammenarbeit mit den anderen Akteuren
   b) Einschätzung der Fähigkeiten der russischen Entwickler
   c) Auffälligkeiten/Probleme bei Arbeit mit unbekannten Entwicklern
   d) Sonstige Schwierigkeiten und Probleme bei der Zusammenarbeit
2. Aufgabenverteilung
   a) Wer gibt Aufgaben vor?
   b) Wer kontrolliert wie den Fortschritt?
   c) Wie läuft die Abnahme der Arbeiten?
   d) Verbesserungsmöglichkeiten?
3. Technologien
   a) Alle Technologien und Werkzeug offengelegt?
   b) Ihm vertraut, insb. JPF (Java Plug-In Framework)?
   c) Damit effizientes Arbeiten möglich?
4. Architektur
   a) Wie definiert?
   b) Wie fortgeschrieben?
   c) Wie dokumentiert?

5. Architekturanalyse
    a) Wie kam es dazu? / Wer gab den Anstoß?
    b) Auf welcher Basis durchgeführt?
    c) Welche Ergebnisse? (Metriken, Zyklen, Architekturvorstellung)?
    d) Wie den russischen Entwicklern kommuniziert?
    e) Wie Änderungen überprüft?
    f) Regelmäßig durchführbar?
6. Problem der fehlenden Tests
    a) Darstellung aus eigener Sicht
    b) Vorschläge zur Lösung
7. Verbesserungsmöglichkeiten?
    a) Anstehende Änderungen
    b) Eigene Ideen
    c) Stärkere Einbindung als Software-Architekt

Literaturverzeichnis

Anmerkung: Wie alle Internetadressen in dieser Arbeit wurden auch die Quellenangaben für elektronische Dokumente zuletzt am 23. April 2010 auf Erreichbarkeit überprüft.

[Abbas et al. 1998a] Abbas, Rasha ; Dart, Philip ; Kazmierczak, Ed: The Role of Software Architecture in Collaborative and Outsourcing Software Development Projects. In: Han, Jun (Hrsg.): *Proceedings of the 1998 Australasian Workshop on Software Architectures, Melbourne, Australien, 24. November*, 1998, S. 14–25

[Abbas et al. 1998b] Abbas, Rasha ; Dart, Philip ; Kazmierczak, Ed ; O'Brien, Fergus: Outsourcing Software Applications Development: Issues, Implications and Impact. In: Baets, Walter R. J. (Hrsg.): *Proceedings of the 6th European Conference on Information Systems (ECIS '98), Aix-en-Provence, Frankreich, 4.–6. Juni*, 1998, S. 628–642

[Abowd et al. 1993] Abowd, Gregory ; Allen, Robert ; Garlan, David: Using Style to Understand Descriptions of Software Architecture. In: *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '93), Los Angeles, CA, USA, 7.–10. Dezember*, 1993, S. 9–20

[Abrahamsson et al. 2010] Abrahamsson, Pekka ; Ali Babar, Muhammad ; Kruchten, Philippe: Agility and Architecture: Can They Coexist? In: *IEEE Software* 27 (2010), S. 16–22

[Abrahamsson et al. 2002] Abrahamsson, Pekka ; Salo, Outi ; Ronkainen, Jussi ; Warsta, Juhani: Agile Software Development Methods: Review and Analysis. In: *VTT Publications* 478 (2002), S. 61–68

[Abrahamsson et al. 2003] Abrahamsson, Pekka ; Warsta, Juhani ; Siponen, Mikko T. ; Ronkainen, Jussi: New Directions on Agile Methods: A Comparative Analysis. In: *Proceedings of the 25th International Conference on Software Engineering, Portland, OR, USA, 3.–10. Mai*, 2003, S. 244–254

[Ågerfalk u. Fitzgerald 2006] Ågerfalk, Pär J. ; Fitzgerald, Brian: Flexible and Distributed Software Processes: Old Petunias in New Bowls? In: *Commun. ACM* 49 (2006), Nr. 10, S. 27–34

[Ågerfalk et al. 2005] Ågerfalk, Pär J. ; Fitzgerald, Brian ; Holmström, Helena ; Lings, Brian ; Lundell, Björn ; Ó Conchúir, Eoin: A Framework for Considering Opportunities and Threats in Distributed Software Development. In: *Proceedings of*

the International Workshop on Distributed Software Development, Paris, Frankreich, 29. August, 2005, S. 47–61

[Agile Alliance 2001] Agile Alliance: *Manifesto for Agile Software Development.* http://www.agilemanifesto.org, 2001

[Ahern et al. 2001] Ahern, Dennis M. ; Turner, Richard ; Clouse, Aaron: *CMMI Distilled: A Practical Introduction to Integrated Process Improvement.* Addison Wesley Professional, 2001

[Albayrak et al. 2007] Albayrak, Can Adam ; Gadatsch, Andreas ; Olufs, Dirk: IT-Outsourcing im Kontext global tätiger Unternehmen. In: *HMD* 254 (2007), S. 27–38

[Alexander et al. 1977] Alexander, Christopher ; Ishikawa, Sara ; Silverstein, Murray ; Jacobson, Max ; Fiksdahl-King, Ingrid ; Angel, Shlomo: *A Pattern Language.* Oxford University Press, 1977

[Ali Babar 2009] Ali Babar, Muhammad: A Framework for Supporting the Software Architecture Evaluation Process in Global Software Development. In: *Proceedings of the 4th IEEE International Conference on Global Software Engineering (ICGSE 2009), Limerick, Irland, 13.–16. Juli*, 2009, S. 93–102

[Altmann u. Pomberger 1999] Altmann, Josef ; Pomberger, Gustav: Kooperative Softwareentwicklung: Konzepte, Modell und Werkzeuge. In: Scheer, August-Wilhelm (Hrsg.) ; Nüttgens, Markus (Hrsg.): *Electronic Business Engineering: 4. Internationale Tagung Wirtschaftsinformatik 1999, Saarbrücken, 3.–5. März*, 1999, S. 643–664

[Amberg u. Wiener 2006] Amberg, Michael ; Wiener, Martin: *IT-Offshoring. Management internationaler IT-Outsourcing-Projekte.* Physica-Verlag, 2006

[Ambler 2003] Ambler, Scott W.: *Outsourcing Examined.* http://www.ddj.com/article/184414972, 2003

[Ambler 2006] Ambler, Scott W.: *The Agile Unified Process (AUP).* http://www.ambysoft.com/unifiedprocess/agileUP.html, 2006

[ANSI/IEEE-Standard-1471 2000] ANSI/IEEE-Standard-1471: *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems – Description.* ANSI/IEEE Computer Society, 2000

[Aspray et al. 2006] Aspray, William (Hrsg.) ; Mayadas, Frank (Hrsg.) ; Vardi, Moshe Y. (Hrsg.): *Globalization and Offshoring of Software.* ACM Job Migration Task Force, http://www.acm.org/globalizationreport/, 2006

[A. T. Kearney 2007] A. T. Kearney: *Offshoring for Long-Term Advantage. The 2007 A. T. Kearney Global Services Location Index.* http://www.atkearney.com/res/shared/pdf/GSLI_2007.pdf, 2007

[Avritzer et al. 2008] Avritzer, Alberto ; Paulish, Daniel ; Cai, Yuanfang: Coordination Implications of Software Architecture in a Global Software Development Project. In: *Proceedings of the 7th Working IEEE/IFIP Conference on Software Architecture (WICSA'08), Vancouver, Kanada, 18.–22. Februar*, 2008, S. 107–116

[B. M. Shao u. Smith David 2007] B. M. Shao, Benjamin ; Smith David, Julie: The Impact of Offshore Outsourcing on IT Workers in Developed Countries. In: *Commun. ACM* 50 (2007), Nr. 2, S. 89–94

[Balaji u. Ahuja 2005] Balaji, S. ; Ahuja, Manju K.: Critical Team-Level Success Factors of Offshore Outsourced Projects: A Knowledge Integration Perspective. In: *Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS-38), Big Island, HI, USA, 3.–6. Januar*, 2005

[Bass et al. 2003] Bass, Len ; Clements, Paul ; Kazman, Rick: *Software Architecture in Practice*. 2. Auflage. Addison-Wesley Longman Publishing, 2003

[Bass u. Kazman 1999] Bass, Len ; Kazman, Rick: *Architecture-Based Development (CMU/SEI-99-TR-007)*. Carnegie Mellon University, 1999

[Bass 2006] Bass, Matthew: Monitoring GSD Projects via Shared Mental Models: A Suggested Approach. In: *Proceedings of the 2006 International Workshop on Global Software Development for the Practitioner (GSD 2006), Shanghai, China, 23. Mai*, 2006, S. 34–37

[Bass et al. 2007] Bass, Matthew ; Mikulovic, Vesna ; Bass, Len ; Herbsleb, James D. ; Cataldo, Marcelo: Architectural Misalignment: An Experience Report. In: *Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA'07), Mumbai, Indien, 6.–9. Januar*, 2007

[Beck 1999] Beck, Kent: *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing, 1999

[Beck 2002] Beck, Kent: *Test Driven Development: By Example*. Addison-Wesley Professional, 2002

[Beck u. Andres 2004] Beck, Kent ; Andres, Cynthia: *Extreme Programming Explained: Embrace Change*. 2. Auflage. Addison-Wesley Professional, 2004

[Becker-Pechau et al. 2006] Becker-Pechau, Petra ; Karstens, Bettina ; Lilienthal, Carola: Automatisierte Softwareüberprüfung auf der Basis von Architekturregeln. In: Biel, Bettina (Hrsg.) ; Book, Matthias (Hrsg.) ; Gruhn, Volker (Hrsg.): *Software Engineering 2006, Fachtagung des GI-Fachbereichs Softwaretechnik, Leipzig, 28.–31. März*, 2006, S. 27–37

[Bischofberger et al. 2004] Bischofberger, Walter R. ; Kühl, Jan ; Löffler, Silvio: Sotograph – A Pragmatic Approach to Source Code Architecture Conformance Checking. In: Oquendo, Flávio (Hrsg.) ; Warboys, Brian (Hrsg.) ; Morrison,

Ronald (Hrsg.): *Proceedings of the 1st European Workshop on Software Architecture (EWSA 2004), St Andrews, England, 21.–22. Mai*, 2004, S. 1–9

[BITKOM 2009] BITKOM (Hrsg.): *Outsourcing wächst in der Krise (Presseinformation vom 22. Januar 2009)*. BITKOM, 2009

[Bliesener 1994] Bliesener, Max-Michael: Outsourcing als mögliche Strategie zur Kostensenkung. In: *Betriebswirtschaftliche Forschung und Praxis* 46 (1994), Nr. 4, S. 277–290

[Boden et al. 2009] Boden, Alexander ; Nett, Bernhard ; Wulf, Volker: Offshoring in kleinen und mittleren Unternehmen der Software-Industrie. In: *HMD* 265 (2009), S. 92–100

[Böhm 2003] Böhm, Christoph: *What Makes Offshore IT Different?* `http://www.competence-site.de/it-management/What-makes-offshore-IT-different`, 2003

[Boos et al. 2005] Boos, Erik ; Iesalnieks, Jānis ; Keller, Florian ; Moczadlo, Regina ; Rathgeb, Karl ; Rohlfes, Morten ; Schmidt, Christof ; Stimmer, Jörg: *BITKOM Leitfaden Offshoring*. BITKOM, 2005

[Bortz u. Döring 2006] Bortz, Jürgen ; Döring, Nicola: *Forschungsmethoden und Evaluation. Für Human- und Sozialwissenschaftler*. 4. Auflage. Springer, 2006

[Bos u. Vriens 2004] Bos, Erik ; Vriens, Christ: An Agile CMM (Experience Paper). In: Zannier, Carmen (Hrsg.) ; Erdogmus, Hakan (Hrsg.) ; Lindstrom, Lowell (Hrsg.): *Proceedings of the 4th Conference on Extreme Programming and Agile Methods (XP/Agile Universe 2004), Calgary, Kanada, 15.–18. August*, 2004, S. 129–138

[Braithwaite u. Joyce 2005a] Braithwaite, Keith ; Joyce, Tim: XP Expanded: Distributed Extreme Programming. In: Baumeister, Hubert (Hrsg.) ; Marchesi, Michele (Hrsg.) ; Holcombe, Mike (Hrsg.): *Proceedings of the 6th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2005), Sheffield, England, 18.–23. Juni*, 2005, S. 180–188

[Braithwaite u. Joyce 2005b] Braithwaite, Keith ; Joyce, Tim: XP Expanded: Patterns for Distributed Extreme Programming. In: *Proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPLoP 2005), Irsee, 6.–10. Juli*, 2005, S. C3–1–C3–10

[Braun 2004] Braun, Andreas: *A Software Architecture for Knowledge Acquisition and Retrieval for Global Software Development Teams*, Fakultät für Informatik, Technische Universität München, Diss., 2004

[Bräutigam u. Grabbe 2004] Bräutigam, Peter ; Grabbe, Hartwig: Teil 2: Rechtliche Ausgangspunkte. In: Bräutigam, Peter (Hrsg.): *IT Outsourcing. Eine Darstellung aus rechtlicher, technischer, wirtschaftlicher und vertraglicher Sicht*. Erich Schmidt Verlag, 2004, S. 161–202

[Breitling et al. 2006] Breitling, Holger ; Kornstädt, Andreas ; Sauer, Joachim: Design Rationale in Exemplary Business Process Modeling. In: Dutoit, Allen H. (Hrsg.) ; McCall, Raymond (Hrsg.) ; Mistrik, Ivan (Hrsg.) ; Paech, Barbara (Hrsg.): *Rationale Management in Software Engineering*. Springer, 2006, S. 191–208

[Brockhaus 1996] Brockhaus: *Brockhaus Enyzklopädie, Band 30: Ergänzungen A–Z*. 19. Auflage. F. A. Brockhaus AG, 1996

[Brockhaus 2002] Brockhaus: *Der Brockhaus: in 15 Bänden. Permanent aktualisierte Online-Auflage*. Bibliographisches Institut & F. A. Brockhaus AG, 2002-2009

[Brooks 1987] Brooks, Fred P.: No Silver Bullet: Essence and Accidents of Software Engineering. In: *Computer* 20 (1987), Nr. 4, S. 10–19

[Bruch 1998] Bruch, Heike: *Outsourcing. Konzepte und Strategien, Chancen und Risiken*. Gabler Verlag, 1998

[Buchta et al. 2004] Buchta, Dirk ; Linss, Heinz ; Röder, Holger ; Ziegler, Robert: *IT-Offshoring und Implikationen für den Standort Deutschland*. A. T. Kearney, 2004

[Buschmann et al. 1996] Buschmann, Frank ; Meunier, Regine ; Rohnert, Hans ; Sommerlad, Peter ; Stal, Michael: *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996

[Carmel 1999] Carmel, Erran: *Global Software Teams: Collaborating Across Borders and Time Zones*. Prentice Hall, 1999

[Carmel et al. 2009] Carmel, Erran ; Dubinsky, Yael ; Espinosa, J. Alberto: Follow the Sun Software Development: New Perspectives, Conceptual Foundation, and Exploratory Field Study. In: *Proceedings of the 42nd Annual Hawaii International Conference on System Sciences (HICSS-42), Big Island, HI, USA, 5.–8. Januar*, 2009, S. 1–9

[Carmel u. Nicholson 2005] Carmel, Erran ; Nicholson, Brian: Small Firms and Offshore Software Outsourcing: High Transaction Costs and their Mitigation. In: *Journal of Global Information Management* 13 (2005), Nr. 3, S. 33–54

[Carmel u. Tjia 2006] Carmel, Erran ; Tjia, Paul: *Offshoring Information Technology – Sourcing and Outsourcing to a Global Workforce*. Cambridge University Press, 2006

[Cataldo et al. 2007] Cataldo, Marcelo ; Bass, Matthew ; Herbsleb, James D. ; Bass, Len: On Coordination Mechanisms in Global Software Development. In: *Proceedings of the 2nd IEEE International Conference on Global Software Engineering (ICGSE 2007), München, 27.–30. August*, 2007, S. 71–80

[Cheon et al. 1995] Cheon, Myun J. ; Grover, Varun ; Teng, James T. C.: Theoretical Perspectives on the Outsourcing of Information Systems. In: *Journal of Information Technology* 10 (1995), Nr. 4, S. 209–219

[Christiansen 2007] Christiansen, Henrik M.: Meeting the Challenge of Communication in Offshore Software Development. In: Meyer, Bertrand (Hrsg.) ; Joseph, Mathai (Hrsg.): *Proceedings of the 1st International Conference on Software Engineering Approaches for Offshore and Outsourced Development (SEAFOOD 2007). Revised Papers, Zürich, Schweiz, 5.–6. Februar*, 2007, S. 19–26

[Clements et al. 2002] Clements, Paul ; Garlan, David ; Bass, Len ; Stafford, Judith ; Nord, Robert ; Ivers, James ; Little, Reed: *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002

[Clerc et al. 2007] Clerc, Viktor ; Lago, Patricia ; Vliet, Hans v.: Global Software Development: Are Architectural Rules the Answer? In: *Proceedings of the 2nd IEEE International Conference on Global Software Engineering (ICGSE 2007), München, 27.–30. August*, 2007, S. 225–234

[Clerc et al. 2009] Clerc, Viktor ; Lago, Patricia ; Vliet, Hans v.: The Usefulness of Architectural Knowledge Management Practices in GSD. In: *Proceedings of the 4th IEEE International Conference on Global Software Engineering (ICGSE 2009), Limerick, Irland, 13.–16. Juli*, 2009, S. 73–82

[Cockburn 2002] Cockburn, Alistair: *Agile Software Development*. Pearson Education, 2002

[Cockburn 2003] Cockburn, Alistair: *People and Methodologies in Software Development*, Faculty of Mathematics and Natural Sciences, University of Oslo, Norwegen, Diss., 2003

[Conway 1968] Conway, Melvin E.: How do Committees Invent? In: *Datamation* 14 (1968), Nr. 4, S. 28–31. – Onlinefassung und Vorwort des Autors von 2001 unter `http://www.melconway.com/research/committees.html`

[Coplien u. Harrison 2004] Coplien, James O. ; Harrison, Neil B.: *Organizational Patterns of Agile Software Development*. Prentice Hall, 2004

[Corry et al. 2006] Corry, Aino V. ; Hansen, Klaus M. ; Svensson, David: Traveling Architects – A New Way of Herding Cats. In: Hofmeister, Christine (Hrsg.) ; Crnkovic, Ivica (Hrsg.) ; Reussner, Ralf (Hrsg.): *Proceedings of the 2nd International Conference on Quality of Software Architectures (QoSA 2006). Revised Papers, Västerås, Schweden, 27.–29. Juni*, 2006, S. 111–126

[Craddock u. Singhal 2006] Craddock, Andrew ; Singhal, Mukul: Embracing Agility – The Change Evolution. In: *Proceedings of the Agile Business Conference 2006, London, England, 7.–9. November*, 2006

[Cullen u. Willcocks 2003] Cullen, Sara ; Willcocks, Leslie: *Intelligent IT Outsourcing: Eight Building Blocks to Success*. Elsevier, 2003

[Cummings 2007] Cummings, Jonathon N.: Leading Groups from a Distance: How to Mitigate Consequences of Geographic Dispersion. In: Weisband, Suzanne P. (Hrsg.): *Leadership at a Distance: Research in Technologically-Supported Work*. Lawrence Erlbaum Associates, 2007, S. 33–50

[Curtis et al. 1988] Curtis, Bill ; Krasner, Herb ; Iscoe, Neil: A Field Study of the Software Design Process for Large Systems. In: *Commun. ACM* 31 (1988), Nr. 11, S. 1268–1287

[Cusumano et al. 2003] Cusumano, Michael ; MacCormack, Alan ; Kemerer, Chris F. ; Crandall, Bill: Software Development Worldwide: The State of the Practice. In: *IEEE Software* 20 (2003), Nr. 6, S. 28–34

[Damian u. Moitra 2006] Damian, Daniela ; Moitra, Deependra: Global Software Development: How Far Have We Come? In: *IEEE Software* 23 (2006), Nr. 5, S. 17–19

[Danait 2005] Danait, Ajay: Agile Offshore Techniques – A Case Study. In: *Proceedings of the AGILE 2005 Conference, Denver, CO, USA, 24.–29. Juli*, 2005, S. 214–217

[Daniels u. Dyson 2004] Daniels, John ; Dyson, Paul: *Dispersed Development*. `http://www.e2x.co.uk/resources/DispersedAgileDevelopment.pdf`, 2004

[de Souza et al. 2007] de Souza, Cleidson R. B. ; Hildenbrand, Tobias ; Redmiles, David F.: Toward Visualization and Analysis of Traceability Relationships in Distributed and Offshore Software Development Projects. In: Meyer, Bertrand (Hrsg.) ; Joseph, Mathai (Hrsg.): *Proceedings of the 1st International Conference on Software Engineering Approaches for Offshore and Outsourced Development (SEAFOOD 2007). Revised Papers, Zürich, Schweiz, 5.–6. Februar*, 2007, S. 182–199

[Dibbern 2004] Dibbern, Jens: *The Sourcing of Application Software Services: Empirical Evidence of Cultural, Industry and Functional Differences (Information Age Economy)*. Physica-Verlag, 2004

[Dibbern et al. 2004] Dibbern, Jens ; Goles, Tim ; Hirschheim, Rudy ; Jayatilaka, Bandula: Information Systems Outsourcing: A Survey and Analysis of the Literature. In: *SIGMIS Database* 35 (2004), Nr. 4, S. 6–102

[Dibbern u. Heinzl 2001] Dibbern, Jens ; Heinzl, Armin: Outsourcing der Informationsverarbeitung im Mittelstand: Test eines multitheoretischen Kausalmodells. In: *Wirtschaftsinformatik* 43 (2001), Nr. 4, S. 339–350

[Dijkstra 1968] Dijkstra, Edsger W.: The Structure of the T.H.E. Multiprogramming System. In: *Commun. ACM* 11 (1968), Nr. 5, S. 341–346

[DSDM Consortium 2005] DSDM Consortium: *Guidelines for Applying DSDM in an Offshore Environment (Whitepaper)*. 2005

[DSDM Consortium 2010] DSDM Consortium: *DSDM Public Version 4.2 Manual*. `http://www.dsdm.org/version4/2/public/`, 2010

[Duden 2009] Duden: *Duden Band 1. Die deutsche Rechtschreibung.* 25. Auflage. Bibliographisches Institut & F. A. Brockhaus AG, 2009

[Dumslaff 2007] Dumslaff, Uwe: *Der Faktor Software-Engineering in der Software-Industrialisierung (Keynote auf der Software Engineering 2007).* 2007

[Duvall et al. 2007] Duvall, Paul ; Matyas, Steve ; Glover, Andrew: *Continuous Integration: Improving Software Quality and Reducing Risk.* Addison-Wesley Professional, 2007

[Eckstein 2009] Eckstein, Jutta: *Agile Softwareentwicklung mit verteilten Teams.* dpunkt.verlag, 2009

[Espinosa u. Carmel 2003] Espinosa, J. Alberto ; Carmel, Erran: The Impact of Time Separation on Coordination in Global Software Teams: A Conceptual Foundation. In: *Software Process: Improvement and Practice* 8 (2003), Nr. 4, S. 249–266

[Espinosa et al. 2003] Espinosa, J. Alberto ; Cummings, Jonathon N. ; Wilson, Jeanne M. ; Pearce, Brandi M.: Team Boundary Issues Across Multiple Global Firms. In: *Journal of Management Information Systems* 19 (2003), Nr. 4, S. 157–190

[Espinosa et al. 2002] Espinosa, J. Alberto ; Kraut, Robert E. ; Slaughter, Sandra A. ; Lerch, Javier F. ; Herbsleb, James D. ; Mockus, Audris: Shared Mental Models, Familiarity, and Coordination: A Multi-Method Study of Distributed Software Teams. In: *Proceedings of the 23rd International Conference in Information Systems, Barcelona, Spanien, 15.–18. Dezember*, 2002, S. 425–433

[Espinosa et al. 2007] Espinosa, J. Alberto ; Nan, Ning ; Carmel, Erran: Do Gradations of Time Zone Separation Make a Difference in Performance? A First Laboratory Study. In: *Proceedings of the 2nd IEEE International Conference on Global Software Engineering (ICGSE 2007), München, 27.–30. August*, 2007, S. 12–20

[Eswaran u. Satpathy 2006] Eswaran, Suchitra ; Satpathy, Sabyasachi: *Offshore and Nearshore ITO and BPO Salary Report.* `http://www.neoit.com/PDFs/Whitepapers/OIv4i04_0506_ITO_and_BPO_Salary_Report_2006.pdf`, 2006

[Europäische Kommission 2006] Europäische Kommission: *Die neue KMU-Definition – Benutzerhandbuch und Mustererklärung.* `http://ec.europa.eu/enterprise/enterprise_policy/sme_definition/index_de.htm`, 2006

[Evans 2003] Evans, Eric: *Domain-Driven Design: Tackling Complexity In the Heart of Software.* Addison-Wesley Longman Publishing, 2003

[Farrell 2004] Farrell, Diana: *Can Germany Win from Offshoring?* McKinsey Global Institute, 2004

[Flick 2006] Flick, Uwe: *Qualitative Sozialforschung.* 4. Auflage. rowohlts enzyklopädie, 2006

[Flor 2006] Flor, Nick V.: Globally Distributed Software Development and Pair Programming. In: *Commun. ACM* 49 (2006), Nr. 10, S. 57–58

[Floyd 2007] Floyd, Christiane: Architekturzentrierte Softwaretechnik (Eröffnungsbeitrag auf der Software Engineering 2007). In: Bleek, Wolf-Gideon (Hrsg.) ; Raasch, Jörg (Hrsg.) ; Züllighoven, Heinz (Hrsg.): *Software Engineering 2007, Fachtagung des GI-Fachbereichs Softwaretechnik, Hamburg, 27.–30. März*, 2007, S. 19–24

[Floyd et al. 1989] Floyd, Christiane ; Reisin, Fanny-Michaela ; Schmidt, Gerhard: STEPS to Software Development with Users. In: Ghezzi, Carlo (Hrsg.) ; McDermid, John A. (Hrsg.): *Proceedings of the 2nd European Software Engineering Conference (ESEC '89), Coventry, England, 11.–15. September*, 1989, S. 48–64

[Fowler 2006a] Fowler, Martin: *Continuous Integration*. `http://www.martinfowler.com/articles/continuousIntegration.html`, 2006

[Fowler 2006b] Fowler, Martin: *Using an Agile Software Process with Offshore Development*. `http://www.martinfowler.com/articles/agileOffshore.html`, 2006

[Frost 2007] Frost, Randall: Jazz and the Eclipse Way of Collaboration. In: *IEEE Software* 24 (2007), Nr. 6, S. 114–117

[Gadatsch 2006] Gadatsch, Andreas: *IT-Offshore realisieren*. Vieweg, 2006

[Gallivan u. Srite 2005] Gallivan, Michael ; Srite, Mark: Information Technology and Culture: Identifying Fragmentary and Holistic Perspectives of Culture. In: *Information and Organization* 15 (2005), Nr. 4, S. 295–338

[Gamma et al. 1995] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns*. Addison-Wesley Longman Publishing, 1995

[Glaser u. Strauss 1967] Glaser, Barney G. ; Strauss, Anselm L.: *The Discovery of Grounded Theory: Strategies for Qualitative Research (Observations)*. Aldine Publishing Company, 1967

[Grinter et al. 1999] Grinter, Rebecca E. ; Herbsleb, James D. ; Perry, Dewayne E.: The Geography of Coordination: Dealing with Distance in R&D Work. In: *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work (GROUP '99), Phoenix, AZ, USA, 14.–17. November*, 1999, S. 306–315

[Gross u. Koch 2007] Gross, Tom ; Koch, Michael ; Herczeg, Michael (Hrsg.): *Computer-Supported Cooperative Work – Interaktive Medien zur Unterstützung von Teams und Communities*. Oldenbourg Verlag, 2007

[Grover et al. 1996] Grover, Varun ; Cheon, Myun J. ; Teng, James T. C.: The Effect of Service Quality and Partnership on the Outsourcing of Information Systems Functions. In: *J. Manage. Inf. Syst.* 12 (1996), Nr. 4, S. 89–116

[Gumm 2006] Gumm, Dorina C.: Distribution Dimensions in Software Development Projects: A Taxonomy. In: *IEEE Software* 23 (2006), Nr. 5, S. 45–51

[Hamm 2006] Hamm, Steve: *Bangalore Tiger – How Indian Tech Upstart Wipro is Rewriting the Rules of Global Competition.* McGraw-Hill, 2006

[Hanks 2005] Hanks, Brian: *Tool Support for Distributed Pair Programming*, Computer Science, University of California, Santa Cruz, Diss., 2005

[Hargreaves et al. 2004] Hargreaves, Elizabeth ; Damian, Daniela ; Lanubile, Filippo ; Chisan, James: Global Software Development: Building a Research Community. In: *SIGSOFT Softw. Eng. Notes* 29 (2004), Nr. 5, S. 1–5

[Hatch 2005] Hatch, Phillip J.: *Offshore 2005 Research: Preliminary Findings and Conclusions, Vers. 1.2.5.* Ventoro, 2005

[Hawthorne u. Perry 2005] Hawthorne, Matthew J. ; Perry, Dewayne E.: Software Engineering Education in the Era of Outsourcing, Distributed Development, and Open Source Software: Challenges and Opportunities. In: *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), St. Louis, MO, USA, 15.–21. Mai*, 2005, S. 643–644

[Hazra 2006] Hazra, Tushar K.: Creating a Global Delivery Model for Your Sourcing Initiatives. In: *Cutter Consortium Executive Report* 7 (2006), Nr. 1

[Heinzl 2001] Heinzl, Armin: Zum Aktivitätsniveau empirischer Forschung in der Wirtschaftsinformatik – Erklärungsansatz und Handlungsoptionen. In: Böhler, Heymo (Hrsg.) ; Sigloch, Jochen (Hrsg.): *Empirische Forschung und Unternehmensführung, Festschrift zum 65. Geburtstag von Peter R. Wossidlo.* Verlag Managementforschung, 2001, S. 127–160

[Heinzl u. Autzen 2006] Heinzl, Armin ; Autzen, Birte: Interview mit Holger Röder und Dirk Buchta über Offshore-Outsourcing und dessen Konsequenzen für das Berufsbild des Wirtschaftsinformatikers. In: *Wirtschaftsinformatik* 48 (2006), Nr. 4, S. 282–285

[Herbsleb u. Grinter 1999a] Herbsleb, James D. ; Grinter, Rebecca E.: Architectures, Coordination, and Distance: Conway's Law and Beyond. In: *IEEE Software* 16 (1999), Nr. 5, S. 63–70

[Herbsleb u. Grinter 1999b] Herbsleb, James D. ; Grinter, Rebecca E.: Splitting the Organization and Integrating the Code: Conway's Law Revisited. In: *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999), Los Angeles, CA, USA, 16.–22. Mai*, 1999, S. 85–95

[Hevner et al. 2004] Hevner, Alan R. ; March, Salvatore T. ; Park, Jinsoo ; Ram, Sudha: Design Science in Information Systems Research. In: *MIS Quarterly* 28 (2004), Nr. 1, S. 75–105

[Hildenbrand et al. 2007] Hildenbrand, Tobias ; Rothlauf, Franz ; Heinzl, Armin: Ansätze zur kollaborativen Softwareerstellung. In: *Wirtschaftsinformatik* 49 (Sonderheft) (2007), S. 72–80

[Hinds u. McGrath 2006] Hinds, Pamela ; McGrath, Cathleen: Structures that Work: Social Structure, Work Structure and Coordination Ease in Geographically Distributed Teams. In: *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work (CSCW 2006), Banff, Kanada, 4.–8. November*, 2006, S. 343–352

[Hirschheim et al. 2008] Hirschheim, Rudy ; Dibbern, Jens ; Heinzl, Armin: Foreword to the Special Issue on IS Sourcing. In: *Information Systems Frontiers* 10 (2008), Nr. 2, S. 125–127

[Hoare 1969] Hoare, C. A. R.: An Axiomatic Basis for Computer Programming. In: *Commun. ACM* 12 (1969), Nr. 10, S. 576–580

[Hofstede 2004] Hofstede, Geert: *Cultures and Organizations: Software of the Mind.* 2. Auflage. McGraw-Hill, 2004

[Hofstede 2006] Hofstede, Geert: *Lokales Denken, globales Handeln: Interkulturelle Zusammenarbeit und globales Management.* Deutscher Taschenbuch Verlag, 2006

[IABG 1997] IABG: *V-Modell 97, Teil 3: Handbuchsammlung – Tailoring und projektspezifisches V-Modell.* `http://v-modell.iabg.de/`, 1997

[Jacobson et al. 1999] Jacobson, Ivar ; Booch, Grady ; Rumbaugh, James: *The Unified Software Development Process.* Addison-Wesley Longman Publishing, 1999

[Jennex u. Adelakun 2003] Jennex, Murray E. ; Adelakun, Olayele: Success Factors For Offshore Information Systems Development. In: *Journal of Information Technology Cases and Applications* Bd. 5. Ivy League Publishing, 2003, S. 12–31

[Jensen u. Zilmer 2003] Jensen, Bent ; Zilmer, Alex: Cross-Continent Development Using Scrum and XP. In: Marchesi, Michele (Hrsg.) ; Succi, Giancarlo (Hrsg.): *Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2003), Genua, Italien, 25.–29. Mai*, 2003, S. 146–153

[Jepsen et al. 1989] Jepsen, Leif O. ; Mathiassen, Lars ; Nielsen, Peter A.: Back to Thinking Mode – Diaries as a Medium for Effective Management of Information Systems Development. In: *Behaviour and Information Technology* 8 (1989), Nr. 3, S. 207–217

[Jouanne-Diedrich 2004] Jouanne-Diedrich, Holger v.: 15 Jahre Outsourcing-Forschung: Systematisierung und Lessons Learned. In: Zarnekow, Rüdiger (Hrsg.) ; Brenner, Walter (Hrsg.) ; Grohmann, Helmut H. (Hrsg.): *Informationsmanagement: Konzepte und Strategien für die Praxis.* dpunkt.verlag, 2004, S. 125–133

[Jouanne-Diedrich 2007] Jouanne-Diedrich, Holger v.: *Die ephorie.de IT-Sourcing-Map. Eine Orientierungshilfe im stetig wachsenden Dschungel der Outsourcing-Konzepte.* `http://www.ephorie.de/it-sourcing-map.htm`, 2007

[Kalakota u. Robinson 2004] Kalakota, Ravi ; Robinson, Marcia: *Offshore Outsourcing: Will Your Job Disappear in 2004?* http://www.informit.com/articles/article.asp?p=169548, 2004

[Kalakota u. Robinson 2005] Kalakota, Ravi ; Robinson, Marcia: *Offshore Attrition: 7 Strategies for Recruiting and Retaining IT Employees.* http://www.informit.com/articles/article.asp?p=390818, 2005

[Kano 1984] Kano, Noraki: Attractive Quality and Must-Be Quality. In: *The Journal of the Japanese Society for Quality Control* 14 (1984), Nr. 4, S. 39–48

[Katzenbach u. Smith 1993] Katzenbach, Jon R. ; Smith, Douglas K.: *The Wisdom of Teams. Creating the High-Performance Organization.* Harvard Business School Press, 1993

[King u. Torkzadeh 2008] King, William R. ; Torkzadeh, Gholamreza: Information Systems Offshoring: Research Status and Issues. In: *MIS Quarterly* 32 (2008), Nr. 2, S. 205–225

[Kircher et al. 2002] Kircher, Michael ; Jain, Prashant ; Corsaro, Angelo ; Levine, David L.: Distributed Extreme Programming. In: Marchesi, Michele (Hrsg.) ; Succi, Giancarlo (Hrsg.) ; Wells, Don (Hrsg.) ; Williams, Laurie (Hrsg.) ; Wells, James D. (Hrsg.): *Extreme Programming Perspectives.* Pearson Education, 2002, S. 553–568

[Kleppe et al. 2003] Kleppe, Anneke G. ; Warmer, Jos ; Bast, Wim: *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing, 2003

[Kobayashi-Hillary 2004] Kobayashi-Hillary, Mark: *Outsourcing to India: The Offshore Advantage.* Springer, 2004

[Koch u. Sauer 2010] Koch, Jörn ; Sauer, Joachim: A Task-Driven Approach on Agile Knowledge Transfer. In: Šmite, Darja (Hrsg.) ; Moe, Nils B. (Hrsg.) ; Ågerfalk, Pär J. (Hrsg.): *Agility Across Time and Space.* Springer, 2010, S. 311–319

[Koenen 2004] Koenen, Jens: *Firmen fahren Outsourcing zurück.* http://www.handelsblatt.com/archiv/firmen-fahren-outsourcing-zurueck;724284, 2004

[Korkala et al. 2009] Korkala, Mikko ; Pikkarainen, Minna ; Conboy, Kieran: Distributed Agile Development: A Case Study of Customer Communication Challenges. In: Abrahamsson, Pekka (Hrsg.) ; Marchesi, Michele (Hrsg.) ; Maurer, Frank (Hrsg.): *Proceedings of the 10th International Conference on Agile Processes in Software Engineering and Extreme Programming (XP 2009), Pula (Sardinien), Italien, 25.–29. Mai*, 2009, S. 161–167

[Kornstädt u. Sauer 2007a] Kornstädt, Andreas ; Sauer, Joachim: Mastering Dual-Shore Development – The Tools & Materials Approach Adapted to Agile Offshoring. In: Meyer, Bertrand (Hrsg.) ; Joseph, Mathai (Hrsg.): *Proceedings of*

the 1st International Conference on Software Engineering Approaches for Offshore and Outsourced Development (SEAFOOD 2007). Revised Papers, Zürich, Schweiz, 5.–6. Februar, 2007, S. 83–95

[Kornstädt u. Sauer 2007b] Kornstädt, Andreas ; Sauer, Joachim: Tackling Offshore Communication Challenges with Agile Architecture-Centric Development. In: *Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA'07), Mumbai, Indien, 6.–9. Januar*, 2007

[Kraut u. Streeter 1995] Kraut, Robert E. ; Streeter, Lynn A.: Coordination in Software Development. In: *Commun. ACM* 38 (1995), Nr. 3, S. 69–81

[Krick u. Voß 2005] Krick, Ronald ; Voss, Stefan: Outsourcing nach Mittel- und Osteuropa – neue Chancen für kleine und mittlere Unternehmen. In: *HMD* 245 (2005), S. 37–47

[Krick u. Voß 2007] Krick, Ronald ; Voss, Stefan: Further Steps in Analyzing the Dimensions of Hofstede's Model of National Culture for Potential Relevance to Risk Analysis in Global Software Development. In: Mäkiö, Juho (Hrsg.) ; Betz, Stefanie (Hrsg.) ; Stephan, Rolf (Hrsg.): *Offshoring of Software Development.* Universitätsverlag Karlsruhe, 2007, S. 65–80

[Krishna et al. 2004] Krishna, S. ; Sahay, Sundeep ; Walsham, Geoff: Managing Cross-Cultural Issues in Global Software Outsourcing. In: *Commun. ACM* 47 (2004), Nr. 4, S. 62–66

[Kruchten 1995] Kruchten, Philippe: The 4+1 View Model of Architecture. In: *IEEE Software* 12 (1995), Nr. 6, S. 42–50

[Kruchten 2003] Kruchten, Philippe: *The Rational Unified Process: An Introduction.* 3. Auflage. Addison-Wesley Professional, 2003

[Küchler 2004] Küchler, Peter: Teil 1: Technische und wirtschaftliche Grundlagen. In: Bräutigam, Peter (Hrsg.): *IT Outsourcing. Eine Darstellung aus rechtlicher, technischer, wirtschaftlicher und vertraglicher Sicht.* Erich Schmidt Verlag, 2004, S. 51–159

[Kupka et al. 1982] Kupka, Ingbert ; Maass, Susanne ; Oberquelle, Horst: Kommunikation in Mensch-Rechner-Dialogen. In: Nehmer, Jürgen (Hrsg.): *Beiträge der 12. Jahrestagung der Gesellschaft für Informatik e.V., Kaiserslautern, 5.–7. Oktober*, 1982, S. 211–230

[Kussmaul et al. 2004] Kussmaul, Clifton ; Jack, Roger ; Sponsler, Barry: Outsourcing and Offshoring with Agility: A Case Study (Experience Paper). In: Zannier, Carmen (Hrsg.) ; Erdogmus, Hakan (Hrsg.) ; Lindstrom, Lowell (Hrsg.): *Proceedings of the 4th Conference on Extreme Programming and Agile Methods (XP/Agile Universe 2004), Calgary, Kanada, 15.–18. August*, 2004, S. 147–154

[Lacity u. Willcocks 2001] Lacity, Mary C. ; Willcocks, Leslie P.: *Global Information Technology Outsourcing: In Search of Business Advantage.* John Wiley & Sons, 2001

[Leavitt 1965] Leavitt, Harold J.: Applied Organisational Change in Industry: Structural, Technological and Humanistic Approaches. In: March, James G. (Hrsg.): *Handbook of Organizations.* Rand McNally, 1965, S. 1144–1170

[Leffingwell 2007] Leffingwell, Dean: *Scaling Software Agility: Best Practices for Large Enterprises (The Agile Software Development Series).* Addison-Wesley Professional, 2007

[Leszak u. Meier 2007] Leszak, Marek ; Meier, Manfred: Successful Global Development of a Large-Scale Embedded Telecommunications Product. In: *Proceedings of the 2nd IEEE International Conference on Global Software Engineering (ICGSE 2007), München, 27.–30. August*, 2007, S. 23–32

[Lewin 1946] Lewin, Kurt: Action Research and Minority Problems. In: *Journal of Social Issues* 2 (1946), Nr. 4, S. 34–46

[Lewis 2006] Lewis, Richard D.: *When Cultures Collide: Leading Across Cultures.* 3. Auflage. Nicholas Brealey Publishing, 2006

[Lilienthal 2008] Lilienthal, Carola: *Komplexität von Softwarearchitekturen – Stile und Strategien*, Universität Hamburg, Department Informatik, Diss., 2008

[Lippert et al. 2003a] Lippert, Martin ; Becker-Pechau, Petra ; Breitling, Holger ; Koch, Jörn ; Kornstädt, Andreas ; Roock, Stefan ; Schmolitzky, Axel ; Wolf, Henning ; Züllighoven, Heinz: Developing Complex Projects Using XP with Extensions. In: *IEEE Computer* 36 (2003), Nr. 6, S. 67–73

[Lippert et al. 2003b] Lippert, Martin ; Schmolitzky, Axel ; Züllighoven, Heinz: Metaphor Design Spaces. In: Marchesi, Michele (Hrsg.) ; Succi, Giancarlo (Hrsg.): *Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2003), Genua, Italien, 25.–29. Mai*, 2003, S. 33–40

[Lohr 2004] Lohr, Steve: High-End Technology Work Not Immune to Outsourcing. In: *The New York Times* 16. Juni (2004), S. C1

[Ludewig u. Lichter 2010] Ludewig, Jochen ; Lichter, Horst: *Software Engineering – Grundlagen, Menschen, Prozesse, Techniken.* 2. Auflage. dpunkt.verlag, 2010

[Lytle et al. 1995] Lytle, Anne L. ; Brett, Jeanne M. ; Barsness, Zoe I. ; Tinsley, Catherine H. ; Janssens, Maddy: A Paradigm for Confirmatory Cross-Cultural Research in Organizational Behavior. In: *Research in Organizational Behavior* 17 (1995), S. 167–214

[MacGregor et al. 2005] MacGregor, Eve ; Hsieh, Yvonne ; Kruchten, Philippe: Cultural Patterns in Software Process Mishaps: Incidents in Global Projects. In: *Proceedings of the 2005 Workshop on Human and Social Factors of Software Engineering (HSSE 2005), St. Louis, MO, USA, 16. Mai*, 2005, S. 1–5

[Martin et al. 2004] Martin, Angela ; Biddle, Robert ; Noble, James: When XP Met Outsourcing. In: Eckstein, Jutta (Hrsg.) ; Baumeister, Hubert (Hrsg.): *Proceedings of the 5th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2004), Garmisch-Partenkirchen, 6.–10. Juni*, 2004, S. 51–59

[Martin u. Chaney 2008] Martin, Jeanette S. ; Chaney, Lillian H.: *Global Business Etiquette: A Guide to International Communication and Customs.* Praeger, 2008

[Mason et al. 2005] Mason, Barry ; Tapper, David ; Bremner, Jason ; Hassey, Phil ; Kolding, Marianne ; Kroa, Vladimir ; Monteiro, Mauricio ; Rosenblood, Leslie: *IDC: Worldwide Offshore IT Services 2005–2009 Forecast.* IDC, 2005

[Massol 2004] Massol, Vincent: *AOSD: Agile Offshore (Presentation at JAVAPOLIS 2004).* `http://www.codehaus.org/~vmassol/blog/AOSD-AgileOffshore-20041217.ppt`, 2004

[Mathiassen 1998] Mathiassen, Lars: Reflective Systems Development. In: *Scandinavian Journal of Information Systems* 10 (1998), Nr. 1-2, S. 67–117

[Mathiassen 2000] Mathiassen, Lars: Collaborative Practice Research. In: *Proceedings of the IFIP TC9 WG 9.3 International Conference on Home Oriented Informatics and Telematics (HOIT 2000), Wolverhampton, England, 28.–30. Juni*, 2000, S. 127–148

[Melnik et al. 2004] Melnik, Grigori ; Read, Kris ; Maurer, Frank: Suitability of FIT User Acceptance Tests for Specifying Functional Requirements: Developer Perspective. In: Zannier, Carmen (Hrsg.) ; Erdogmus, Hakan (Hrsg.) ; Lindstrom, Lowell (Hrsg.): *Proceedings of the 4th Conference on Extreme Programming and Agile Methods (XP/Agile Universe 2004), Calgary, Kanada, 15.–18. August*, 2004, S. 60–72

[Mertens 2005] Mertens, Peter: Can Germany Win from Offshoring? – Für Sie gelesen. In: *Wirtschaftsinformatik* 47 (2005), Nr. 3, S. 226–227

[Meyer 2006] Meyer, Bertrand: Offshore Development – The Unspoken Revolution in Software Engineering. In: *IEEE Computer* 39 (2006), Nr. 1, S. 124, 121–123

[Mistrík et al. 2010] Mistrík, Ivan (Hrsg.) ; Grundy, John (Hrsg.) ; van der Hoek, André (Hrsg.) ; Whitehead, Jim (Hrsg.): *Collaborative Software Engineering.* Springer, 2010

[Moczadlo 2003] Moczadlo, Regina: *Chancen und Risiken des Offshore Development. Empirische Analyse der Erfahrungen deutscher Unternehmen.* `http://www.competence-site.de/it-management/Chancen-und-Risiken-des-Offshore-Development`, 2003

[Moczadlo 2005] Moczadlo, Regina: *Softwaredevelopment deutscher Unternehmen in Indien und Nearshore im Vergleich.* http://www.competence-site.de/it-outsourcing/Softwaredevelopment-deutscher-Unternehmen-Indien-Nearshore-Vergleich, 2005

[Moore u. Barnett 2004] Moore, Stephanie ; Barnett, Liz: *Offshore Outsourcing and Agile Development.* Forrester Research, 2004

[Muckel 2007] Muckel, Petra: Die Entwicklung von Kategorien mit der Methode der Grounded Theory. In: *HSR Supplement Nr. 19: Grounded Theory Reader* (2007), S. 211–231

[Murphy et al. 1995] Murphy, Gail C. ; Notkin, David ; Sullivan, Kevin: Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In: *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '93), Los Angeles, CA, USA, 7.–10. Dezember*, 1995, S. 18–28

[Myers 1978] Myers, Glenford J.: *Composite/Structured Design.* Van Nostrand Reinhold, 1978

[Narayanaswamy u. Henry 2005] Narayanaswamy, Ravi ; Henry, Raymond M.: Effects of Culture on Control Mechanisms in Offshore Outsourced IT Projects. In: *Proceedings of the 2005 ACM SIGMIS CPR Conference on Computer Personnel Research (SIGMIS CPR '05), Atlanta, GA, USA, 14.–16. April*, 2005, S. 139–145

[NASSCOM 2005] *NASSCOM–McKinsey Report 2005. Extending India's leadership of the Global IT and BPO Industries.* NASSCOM und McKinsey, 2005

[Nett et al. 2004] Nett, Bernhard ; Durissini, Marco ; Klann, Markus: *Wissensprozesse in kleinen Unternehmen der Softwarebranche aus der Sicht von Entwicklern.* ViSEK Report 047D v1.7., http://www.software-kompetenz.de/?21522, 2004

[Nicklisch 2006] Nicklisch, Gerd (Hrsg.): *Outsourcing: Der (Irr) Weg – Bestandsaufnahme, Trends und Analyse der Informationstechnologie in Deutschland.* Datakontext-Fachverlag, 2006

[Nicklisch et al. 2008] Nicklisch, Gerd ; Borchers, Jens ; Krick, Ronald ; Rucks, Rainer: *IT-Near- und -Offshoring in der Praxis.* dpunkt.verlag, 2008

[Oberquelle 1999] Oberquelle, Horst: *Computergestützte kooperative Arbeit, Skript zur Vorlesung, Wintersemester 1998/99.* Fachbereich Informatik, Universität Hamburg, 1999

[Object Management Group 2009] Object Management Group: *UML Resource Page.* http://www.uml.org/, 2009

[Olson u. Olson 2000] Olson, Gary M. ; Olson, Judith S.: Distance Matters. In: *Human-Computer Interaction* 15 (2000), Nr. 2&3, S. 139–178

[Olsson 2004] Olsson, Eric: European IT SMEs in India. In: Borchers, Jens (Hrsg.) ; Kneuper, Ralf (Hrsg.): *Software Management 2004: Outsourcing und Integration, Fachtagung des GI-Fachausschusses Management der Anwendungsentwicklung und -wartung im Fachbereich Wirtschaftsinformatik (WI-MAW), Bad Homburg, 3.–5. November*, 2004, S. 13–33

[Ostroff et al. 2004] Ostroff, Jonathan S. ; Makalsky, David ; Paige, Richard F.: Agile Specification-Driven Development. In: Eckstein, Jutta (Hrsg.) ; Baumeister, Hubert (Hrsg.): *Proceedings of the 5th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2004), Garmisch-Partenkirchen, 6.–10. Juni*, 2004, S. 104–112

[Ottosson et al. 2006] Ottosson, Stig ; Björk, Evastina ; Holmdahl, Lars ; Sándor, Vajna: Research Approaches on Product Development Processes. In: Marjanović, Dorian (Hrsg.): *Proceedings of the 9th International Design Conference (Design 2006), Dubrovnik, Kroatien, 15.–18. Mai*, 2006, S. 91–97

[Ovaska et al. 2003] Ovaska, Päivi ; Rossi, Matti ; Marttiin, Pentti: Architecture as a Coordination Tool in Multi-Site Software Development. In: *Software Process: Improvement and Practice* 8 (2003), Nr. 4, S. 233–247

[Overby 2003] Overby, Stephanie: *The Hidden Costs of Offshore Outsourcing*. `http://www.cio.com/article/29654/The_Hidden_Costs_of_Offshore_Outsourcing`, 2003

[Paasivaara 2005] Paasivaara, Maria: *Communication Practices in Inter-Organisational Product Development*, Department of Computer Science and Engineering, Helsinki University of Technology (Espoo, Finnland), Diss., 2005

[Paasivaara u. Lassenius 2004] Paasivaara, Maria ; Lassenius, Casper: Using Iterative and Incremental Processes in Global Software Development. In: *Proceedings of the 3rd International Workshop on Global Software Development, International Conference on Software Engineering (ICSE 2004), Edinburgh, Schottland, 24. Mai*, 2004, S. 42–47

[Paasivaara u. Lassenius 2006] Paasivaara, Maria ; Lassenius, Casper: Could Global Software Development Benefit from Agile Methods? In: *Proceedings of the IEEE International Conference on Global Software Engineering (ICGSE 2006), Florianopolis, Brasilien, 16.–19. Oktober*, 2006, S. 109–113

[Palmer u. Felsing 2002] Palmer, Stephen R. ; Felsing, John M.: *A Practical Guide to Feature-Driven Development*. Prentice Hall, 2002

[Pandit 1996] Pandit, Naresh R.: The Creation of Theory: A Recent Application of the Grounded Theory Method. In: *The Qualitative Report* 2 (1996), Nr. 4, S. 1–14

[Parnas 1972] Parnas, David L.: On the Criteria to be Used in Decomposing Systems into Modules. In: *Commun. ACM* 15 (1972), Nr. 12, S. 1053–1058

[Parvathanathan et al. 2007] Parvathanathan, Kamala ; Chakrabarti, Anindya ; Patil, Priti P. ; Sen, Sreerupa ; Sharma, Neeraj ; Johng, Yessong: *Global Development and Delivery in Practice*. IBM Rational India Lab, 2007

[Paulish 2001] Paulish, Daniel J.: *Architecture-Centric Software Project Management: A Practical Guide*. Addison-Wesley Professional, 2001

[Perry u. Wolf 1992] Perry, Dewayne E. ; Wolf, Alexander L.: Foundations for the Study of Software Architecture. In: *SIGSOFT Softw. Eng. Notes* 17 (1992), Nr. 4, S. 40–52

[Poole 2004] Poole, Charles J.: Distributed Product Development Using Extreme Programming. In: Eckstein, Jutta (Hrsg.) ; Baumeister, Hubert (Hrsg.): *Proceedings of the 5th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2004), Garmisch-Partenkirchen, 6.–10. Juni*, 2004, S. 60–67

[Posch et al. 2004] Posch, Torsten ; Birken, Klaus ; Gerdom, Michael: *Basiswissen Softwarearchitektur*. dpunkt.verlag, 2004

[Prechelt 2001] Prechelt, Lutz: *Kontrollierte Experimente in der Softwaretechnik*. Springer, 2001

[Pree 1995] Pree, Wolfgang: *Design Patterns for Object-Oriented Software Development*. Addison-Wesley Longman Publishing, 1995

[Prieß 2006] Priess, Uwe: Wettbewerbsvorteile für Software „made in Germany“. In: *InformationWeek, Special Sourcing Solutions* 22. Juni (2006), S. 32–33

[Ramachandran u. Shukla 2002] Ramachandran, Vinay ; Shukla, Anuja: Circle of Life, Spiral of Death: Are XP Teams Following the Essential Practices? In: Wells, Don (Hrsg.) ; Williams, Laurie A. (Hrsg.): *Proceedings of the 2nd XP Universe and 1st Agile Universe Conference on Extreme Programming and Agile Methods (XP/Agile Universe 2002), Chicago, IL, USA, 4.–7. August*, 2002, S. 166–173

[Ramesh et al. 2006] Ramesh, Balasubramaniam ; Cao, Lan ; Mohan, Kannan ; Xu, Peng: Can Distributed Software Development be Agile? In: *Commun. ACM* 49 (2006), Nr. 10, S. 41–46

[Reussner u. Hasselbring 2009] Reussner, Ralf ; Hasselbring, Wilhelm: *Handbuch der Software-Architektur*. 2. Auflage. dpunkt.verlag, 2009

[Ribeiro 2006] Ribeiro, John: *India's Offshore Outsourcing Revenue Grew 33%*. `http://www.computerworld.com/action/article.do?command=printArticleBasic&articleId=9000877`, 2006

[Robinson u. Kalakota 2004] Robinson, Marcia ; Kalakota, Ravi: *Offshore Outsourcing: Business Models, ROI and Best Practices*. 2. Auflage. Mivar Press, 2004

[Robinson et al. 2005] Robinson, Marcia ; Kalakota, Ravi ; Sharma, Suresh: *Global Outsourcing: Executing an Onshore, Nearshore or Offshore Strategy*. Mivar Press, 2005

[Rocha de Faria u. Adler 2006] Rocha de Faria, Henrique ; Adler, Gary: Architecture-Centric Global Software Processes. In: *Proceedings of the IEEE International Conference on Global Software Engineering (ICGSE 2006), Florianopolis, Brasilien, 16.–19. Oktober*, 2006, S. 241–242

[Röder et al. 2007] Röder, Holger ; Eul, Marcus ; Meyer, Konrad: *IT in Deutschland – was bringt die Zukunft*. A. T. Kearney, 2007

[Royce 1987] Royce, Winston W.: Managing the Development of Large Software Systems: Concepts and Techniques. In: *Proceedings of the 9th International Conference on Software Engineering (ICSE 1987), Monterey, CA, USA, 30. März – 2. April*, 1987, S. 328–338

[Ruiz Ben u. Claus 2005] Ruiz Ben, Esther ; Claus, Regina: Offshoring in der deutschen IT Branche. In: *Informatik Spektrum* 28 (2005), Nr. 1, S. 34–39

[Sahay et al. 2004] Sahay, Sundeep ; Nicholson, Brian ; Krishna, S.: *Global IT Outsourcing: Software Development Across Borders*. Cambridge University Press, 2004

[Salger 2009] Salger, Frank: Software Architecture Evaluation in Global Software Development Projects. In: *Proceedings of the Confederated International Workshops and Posters on On the Move to Meaningful Internet Systems (OTM'09), Vilamoura, Portugal, 1.–6. November*, 2009, S. 391–400

[Sangwan et al. 2006] Sangwan, Raghvinder ; Bass, Matthew ; Mullick, Neel ; Paulish, Daniel J. ; Kazmeier, Juergen: *Global Software Development Handbook*. Auerbach Publications, 2006

[Sauer 2006a] Sauer, Joachim: Agile Offshore Outsourcing – Concepts and Practices for Flexible Integration of Offshore Development Services. In: *Proceedings of the Agile Business Conference 2006, London, England, 7.–9. November*, 2006

[Sauer 2006b] Sauer, Joachim: Agile Practices in Offshore Outsourcing – An Analysis of Published Experiences. In: *Proceedings of the 29th Information Systems Research Seminar in Scandinavia (IRIS 29), Helsingør, Dänemark, 12.–15. August*, 2006

[Sauer 2008] Sauer, Joachim: Enabling Agile Offshoring with the Dual-Shore Model. In: Maalej, Walid (Hrsg.) ; Bruegge, Bernd (Hrsg.): *Software Engineering 2008 – Workshopband, Fachtagung des GI-Fachbereichs Softwaretechnik, München, 18.–22. Februar*, 2008, S. 35–42

[Schaaf 2004] Schaaf, Jürgen: *Offshoring: Globalisierungswelle erfasst Dienstleistungen*. Deutsche Bank Research, 2004

[Schaaf 2005] Schaaf, Jürgen: *Outsourcing nach Indien: der Tiger auf dem Sprung.* Deutsche Bank Research, 2005

[Schaaf u. Weber 2005] Schaaf, Jürgen ; Weber, Mathias: *Offshoring-Report 2005 – Ready for Take-off (Economics – Digitale Ökonomie und struktureller Wandel).* Deutsche Bank Research, 2005

[Scheer et al. 2003] Scheer, Christian ; Deelmann, Thomas ; Loos, Peter: *Geschäftsmodelle und internetbasierte Geschäftsmodelle – Begriffsbestimmung und Teilnehmermodell (ISYM-Arbeitspapier Nr. 12).* Lehrstuhl Wirtschaftsinformatik und Betriebswirtschaftslehre, Johannes Gutenberg-Universität Mainz, 2003

[Schumacher u. Olsson 2005] Schumacher, Peter ; Olsson, Eric: *European IT Companies in India.* `http://www.value-leadership.com/download/vlg_european_study.pdf`, 2005

[Schwaber 2005] Schwaber, Carey: *Corporate IT Leads The Second Wave Of Agile Adoption (Report Nr. 38334).* Forrester Research, 2005

[Schwaber 2004] Schwaber, Ken: *Agile Project Management with Scrum.* Microsoft Press, 2004

[Schwaber u. Beedle 2002] Schwaber, Ken ; Beedle, Mike: *Agile Software Development with Scrum.* Prentice Hall, 2002

[Schwarze u. Müller 2005] Schwarze, Lars ; Müller, Peter P.: IT-Outsourcing – Erfahrungen, Status und zukünftige Herausforderungen. In: *HMD* 245 (2005), S. 6–17

[Scott 2002] Scott, Kendall: *The Unified Process Explained.* Addison-Wesley Longman Publishing, 2002

[Sengupta et al. 2006] Sengupta, Bikram ; Chandra, Satish ; Sinha, Vibha: A Research Agenda for Distributed Software Development. In: Osterweil, Leon J. (Hrsg.) ; Rombach, H. D. (Hrsg.) ; Soffa, Mary L. (Hrsg.): *Proceeding of the 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, 20.–28. Mai*, 2006, S. 731–740

[Sengupta et al. 2004] Sengupta, Bikram ; Sinha, Vibha ; Chandra, Satish: Test-Driven Global Software Development. In: *Proceedings of the 3rd International Workshop on Global Software Development, International Conference on Software Engineering (ICSE 2004), Edinburgh, Schottland, 24. Mai*, 2004, S. 39–41

[Shaw u. Garlan 1996] Shaw, Mary ; Garlan, David: *Software Architecture – Perspectives on an Emerging Discipline.* Prentice Hall, 1996

[Sheppard u. Sherman 1998] Sheppard, Blair H. ; Sherman, Dana M.: The Grammars of Trust: A Model and General Implications. In: *Academy of Management Review* 23 (1998), Nr. 3, S. 422–437

[Sieber 2006] Sieber, Andrea: Kleine Softwareunternehmen und ihre Erfahrungen mit Softwareentwicklung in Osteuropa und Indien. In: Hochberger, Christian (Hrsg.) ; Liskowsky, Rüdiger (Hrsg.): *Beiträge der 36. Jahrestagung der Gesellschaft für Informatik e.V. (INFORMATIK 2006), Dresden, 2.–6. Oktober*, 2006, S. 642–652

[Simons 2002] Simons, Matt: *Internationally Agile*. `http://www.informit.com/articles/printerfriendly.asp?p=25929&rl=1`, 2002

[Simons 2004] Simons, Matthew: *Distributed Agile Development and the Death of Distance. Executive Report 5 (2004), Nr. 4*. Cutter Consortium, 2004

[Slama 2006] Slama, Dirk: Offshore Software-Entwicklung im Griff. In: *Information-Week, Special Sourcing Soluations* 9. März (2006), S. 26–27

[Šmite et al. 2010] Šmite, Darja ; Moe, Nils B. ; Ågerfalk, Pär J.: Fundamentals of Agile Distributed Software Development. In: Šmite, Darja (Hrsg.) ; Moe, Nils B. (Hrsg.) ; Ågerfalk, Pär J. (Hrsg.): *Agility Across Time and Space*. Springer, 2010, S. 3–7

[Smolander 2002] Smolander, Kari: Four Metaphors of Architecture in Software Organizations: Finding Out the Meaning of Architecture in Practice. In: *Proceedings of the 2002 International Symposium on Empirical Software Engineering (ISESE 2002), Nara, Japan, 3.–4. Oktober*, 2002

[Sommerville 2009] Sommerville, Ian: *Software Engineering*. 9. Auflage. Addison-Wesley, 2009

[Sosa 2000] Sosa, Manuel E.: *Analyzing the Effects of Product Architecture on Technical Communication in Product Development Organizations*, Dept. of Mechanical Engineering, Massachusetts Institute of Technology, Diss., 2000

[Sparrow 2003] Sparrow, Elizabeth: *Successful IT Outsourcing*. Springer, 2003

[Stapleton 2003] Stapleton, Jennifer (Hrsg.): *DSDM – Business Focused Development*. 2. Auflage. Pearson Education, 2003

[Statistisches Bundesamt 2009] Statistisches Bundesamt (Hrsg.): *Verflechtung deutscher Unternehmen mit dem Ausland (Begleitmaterial zur Pressekonferenz am 17. Februar 2009 in Berlin)*. Statistisches Bundesamt, 2009

[Steimle 2007] Steimle, Toni: *Softwareentwicklung im Offshoring*. Springer, 2007

[Stephan 2005] Stephan, Rolf: Kommunikation und Wissenstransfer – Schlüsselfaktoren für erfolgreiche Offshore Projekte. In: Hermes, Heinz-Josef (Hrsg.) ; Schwarz, Gerd (Hrsg.): *Outsourcing. Chancen und Risiken, Erfolgsfaktoren, rechtssichere Umsetzung*. Haufe, 2005, S. 213–234

[Steria Mummert Consulting AG 2007] Steria Mummert Consulting AG: *Benchmarking-Studie „Erfolgsmodelle im Outsourcing“*. `http://www.steria-mummert.de/presse/publikationen/studien/it-outsourcing`, 2007

[Stojanovic et al. 2003a] Stojanovic, Zoran ; Dahanayake, Ajantha ; Sol, Henk: Agile Modeling and Design of Service-Oriented Component Architecture. In: Weerawarana, Sanjiva (Hrsg.) ; Piccinelli, Giacomo (Hrsg.): *Proceedings of the 1st European Workshop on Object-Orientation and Web Services, in Conjunction with ECOOP 2003, Darmstadt, 21. Juli*, 2003, S. 54–63

[Stojanovic et al. 2003b] Stojanovic, Zoran ; Dahanayake, Ajantha ; Sol, Henk G.: Component-Oriented Agile Software Development. In: Marchesi, Michele (Hrsg.) ; Succi, Giancarlo (Hrsg.): *Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2003), Genua, Italien, 25.–29. Mai*, 2003, S. 315–318

[Strauss 1993] Strauss, Anselm L.: *Continual Permutations of Action.* Aldine de Gruyter, 1993

[Strauss u. Corbin 1996] Strauss, Anselm L. ; Corbin, Juliet: *Grounded Theory: Grundlagen qualitativer Sozialforschung.* Beltz Psychologie Verlags Union, 1996

[Susman u. Evered 1978] Susman, Gerald I. ; Evered, Roger D.: An Assessment of the Scientific Merits of Action Research. In: *Administrative Science Quarterly* 23 (1978), Nr. 4, S. 582–603

[Sutherland et al. 2007a] Sutherland, Jeff ; Jakobsen, Carsten R. ; Johnson, Kent: Scrum and CMMI Level 5: The Magic Potion for Code Warriors. In: *AGILE Conference* (2007), S. 272–278

[Sutherland et al. 2007b] Sutherland, Jeff ; Viktorov, Anton ; Blount, Jack ; Puntikov, Nikolai: Distributed Scrum: Agile Project Management with Outsourced Development Teams. In: *Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS-40), Big Island, HI, USA, 3.–6. Januar*, 2007

[Szyperski 2002] Szyperski, Clemens: *Component Software: Beyond Object-Oriented Programming.* 2. Auflage. Addison-Wesley Professional, 2002

[Taylor et al. 2006] Taylor, Philip S. ; Greer, Des ; Sage, Paul ; Coleman, Gerry ; McDaid, Kevin ; Keenan, Frank: Do Agile GSD Experience Reports Help the Practitioner? In: *Proceedings of the 2006 International Workshop on Global Software Development for the Practitioner (GSD 2006), Shanghai, China, 23. Mai*, 2006, S. 87–93

[Teufel et al. 1995] Teufel, Stephanie ; Sauter, Christian ; Mühlherr, Thomas ; Bauknecht, Kurt: *Computerunterstützung für die Gruppenarbeit.* Addison-Wesley, 1995

[Thomas 2007] Thomas, Kim ; McCauley, Denis (Hrsg.): *The Means to Compete: Benchmarking IT Industry Competitiveness.* Economist Intelligence Unit, 2007

[Turk et al. 2002] Turk, Dan ; France, Robert ; Rumpe, Bernhard: Limitations of Agile Software Processes. In: *Proceedings of the 3rd International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2002), Alghero (Sardinien), Italien, 26.–30. Mai*, 2002, S. 43–46

[UNCTAD 2004] *United Nations Conference on Trade and Development: World Investment Report 2004 – The Shift Towards Services.* United Nations, 2004

[VersionOne 2006] VersionOne: *Survey: The State of Agile Development.* `http://www.versionone.net/pdf/StateofAgileDevelopmentSurvey.pdf`, 2006

[Wagstrom u. Herbsleb 2006] Wagstrom, Patrick ; Herbsleb, James D.: Dependency Forecasting in the Distributed Agile Organization. In: *Commun. ACM* 49 (2006), Nr. 10, S. 55–56

[Wallace et al. 2002] Wallace, Nathan ; Bailey, Peter ; Ashworth, Neil: Managing XP with Multiple or Remote Customers. In: *Proceedings of the 3rd International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2002), Alghero (Sardinien), Italien, 26.–30. Mai*, 2002, S. 134–137

[Walther u. Bazarova 2007] Walther, Joseph B. ; Bazarova, Natalya N.: Misattribution in Virtual Groups. The Effects of Member Distribution on Self-Serving Bias and Partner Blame. In: *Human Communication Research* 33 (2007), S. 1–26

[Warden u. Shore 2007] Warden, Shane ; Shore, Jim: *The Art of Agile Development.* O'Reilly Media, 2007

[Weber 2006] Weber, Mathias (Hrsg.): *Compliance in IT-Outsourcing-Projekten. Leitfaden zur Umsetzung rechtlicher Rahmenbedingungen.* BITKOM, 2006

[West u. Grant 2010] West, Dave ; Grant, Tom: *Agile Development: Mainstream Adoption Has Changed Agility (Report Nr. 56100).* Forrester Research, 2010

[Westner u. Strahringer 2007] Westner, Markus ; Strahringer, Susanne: Current State of IS Offshoring Research: A Descriptive Meta-Analysis. In: Mäkiö, Juho (Hrsg.) ; Betz, Stefanie (Hrsg.) ; Stephan, Rolf (Hrsg.): *Offshoring of Software Development.* Universitätsverlag Karlsruhe, 2007, S. 7–20

[Westphal 2006] Westphal, Frank: *Testgetriebene Entwicklung mit JUnit & FIT.* dpunkt.verlag, 2006

[Wiener 2006] Wiener, Martin: *Critical Success Factors of Offshore Software Development Projects*, Lehrstuhl für BWL, insb. Wirtschaftsinformatik III, Friedrich-Alexander-Universität Erlangen-Nürnberg, Diss., 2006

[Wilson u. Purushothaman 2003] Wilson, Dominic ; Purushothaman, Roopa: *Dreaming with BRICs: The Path to 2050 (Global Economics Paper No. 99).* `http://www2.goldmansachs.com/ideas/brics/brics-dream.html`, 2003

[Winkler et al. 2008] Winkler, Jessica K. ; Dibbern, Jens ; Heinzl, Armin: The Impact of Cultural Differences in Offshore Outsourcing – Case Study Results from German-Indian Application Development Projects. In: *Information Systems Frontiers* 10 (2008), Nr. 2, S. 243–258

[Winkler 2005] Winkler, Martin: Mit dem Dual Shore Delivery Model Sprach- und Kulturbarrieren bei IT-Offshoreprojekten überwinden. In: Clement, Reiner (Hrsg.) ; Gadatsch, Andreas (Hrsg.) ; Kütz, Martin (Hrsg.) ; Juszczak, Jens (Hrsg.): *IT-Controlling in Forschung und Praxis – Tagungsband zur 2. Fachtagung IT-Controlling, Sankt Augustin, 21.–22. Februar*, 2005, S. 127–136

[Wiredu 2006] Wiredu, Gamel O.: A Framework for the Analysis of Coordination in Global Software Development. In: *Proceedings of the 2006 International Workshop on Global Software Development for the Practitioner (GSD 2006), Shanghai, China, 23. Mai*, 2006, S. 38–44

[Wolf et al. 2001] Wolf, Henning ; Roock, Stefan ; Kornstädt, Andreas ; Züllighoven, Heinz ; Gryczan, Guido: Das JWAM-Framework und Komponenten. Eine konzeptionelle Bestandsaufnahme. In: Turowski, Klaus (Hrsg.): *Tagungsband des 3. Workshops zu komponentenorientierter betrieblicher Anwendungsentwicklung, Frankfurt, 4. April*, 2001, S. 15–21

[Wolf et al. 2005] Wolf, Henning ; Roock, Stefan ; Lippert, Martin: *Extreme Programming. Eine Einführung mit Empfehlungen und Erfahrungen aus der Praxis.* 2. Auflage. dpunkt.verlag, 2005

[Yap 2005] Yap, Monica: Follow the Sun: Distributed Extreme Programming Development. In: *Proceedings of the AGILE 2005 Conference, Denver, CO, USA, 24.–29. Juli*, 2005, S. 218–224

[Züllighoven 1998] Züllighoven, Heinz: *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz.* dpunkt.verlag, 1998

[Züllighoven 2004] Züllighoven, Heinz: *Object-Oriented Construction Handbook: Developing Application-Oriented Software with the Tools & Materials Approach.* dpunkt.verlag. Copublication with Morgan-Kaufmann, 2004

[Züllighoven u. Raasch 2006] Züllighoven, Heinz ; Raasch, Jörg: Softwaretechnik. In: Rechenberg, Peter (Hrsg.) ; Pomberger, Gustav (Hrsg.): *Informatik Handbuch.* 4. Auflage. Hanser Verlag, 2006, S. 795–838