Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Entwicklung eines Prozesses zur Sicherung der Qualität von Schnittstellen in heterogenen Systemlandschaften

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Philipp Wasmayr
Matrikelnummer 0425147

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Prof. Dr. A Min Tjoa
Mitwirkung: Dr. Amin Anjomshoaa

Wien, 16.09.2011

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung

Wasmayr Philipp
Josefstädterstraße 14/34
A-1080 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 16.09.2011, Unterschrift

Abstract

Software systems need adaption due to the organizational changes in companies. Compared to former times, applications nowadays are not centrally maintainable, rather they are often distributed between various geographical locations and/or organizational units. Only the combination of all participating services creates a functional business process. Therefore quality assurance of each component and their composition is a very important topic: All software components should work properly and furthermore the interactions among them must be tested. Quality assurance is a big challenge since these pieces of the implementation are often developed by different organizational units or companies and are much harder to test compared to non-distributed systems. Current process models don't fully cover the requirements of modern distributed systems, such as security aspects or the test procedure for the governance tasks.

This master thesis answers the question how well existing software development processes assist the developers in terms of specification, implementation and testing of distributed systems. Moreover the work should cover the needed changes which lead to an efficient realization of the fields described above.

Therefore this work analyzes, compares and evaluates existing software development processes and their quality assurance measures with the focus on the special requirements of distributed systems (e.g. security or conformance to company rules) and finally presents an adapted process with quality assurance measures which fits to the specific characteristics. After a literature research the most important aspects of quality assurance measures in the field mentioned above are discussed. Different organizational and technical concepts that assure and raise the quality of software are presented. A research for existing tools, which represent the already identified requirements, will be conducted. The results are the fundament for the development of the process, which fullfills the identified quality criterias and furthermore introduces a method for efficient monitoring of a service-oriented architecture. An analysis finally compares all process models for their testing abilities in heterogeneous environments.

The result of this diploma thesis is a new software development process describing methods for the specification, implementation and testing procedures of a service-oriented architecture. Additionally a concept for the efficient monitoring of a heterogeneous environment will be introduced. A comparision of the testing procedures of all current software development models is conducted and the results are presented in the final chapter of this master thesis.

Kurzfassung

Getrieben durch die organisatorischen Veränderungen in Unternehmen müssen sich auch die Systemlandschaften anpassen. Heutzutage sind Anwendungen nicht mehr zentral verwaltbar sondern verteilen sich oft auf unterschiedlichste Standorte oder organisatorische Einheiten. Erst durch den Zusammenschluss dieser Services kommt es zu einem funktionierten Geschäftsprozess, umso wichtiger stellt sich die Qualitätssicherung in den einzelnen Bausteinen und der Komposition dieser dar. Es muss also nicht nur sichergestellt werden, dass jede Komponente für sich funktioniert, sondern auch, dass das Zusammenspiel die erwarteten Resultate liefert. Da diese meist von unterschiedlichen organisatorischen Einheiten beziehungsweise von unterschiedlichen Unternehmen entwickelt werden ist die Sicherung der Qualität eine größere Herausforderung als wenn nur Software ohne Schnittstellen zu anderen Systemen entwickelt wird. Aktuelle Softwareentwicklungsprozesse decken Qualitätsanforderungen verteilter Systeme, wie Sicherheitsaspekte oder die Überprüfung der Einhaltung von zuvor spezifizierten Unternehmensrichtlinien innerhalb einer Service-orientierten Architektur, nur begrenzt ab.
Dieses Werk beantwortet die Fragestellung, wie geeignet bestehende Prozessmodelle zur Spezifikation, Implementierung und Überprüfung heterogener Systeme sind. Des Weiteren wird untersucht welche Verbesserungen in der Vorgehensweise in diesen Bereichen die Umsetzung einer effizienten Service-orientierte Architektur ermöglichen.
Hierfür untersucht, vergleicht und bewertet diese Arbeit bestehende Prozesse der Softwareentwicklung inklusive ihrer Qualitätssicherungsmaßnahmen in Hinblick auf die speziellen Anforderungen verteilter Systeme (wie z.B.: Sicherheit oder die Einhaltung unternehmensinterner Richtlinien) und stellt einen adaptierten Prozess mit Qualitätssicherungsmaßnahmen vor, der auf die spezifischen Charakteristika zugeschnitten ist. Durch eine Literaturrecherche wurden zuerst wichtige Aspekte von Qualitätssicherungsmaßnahmen in dem angesprochenen Bereich erarbeitet. Verschiedene organisatorische und technische Konzepte wurden offengelegt, beispielsweise wie sich die Qualität von Software sichern bzw. steigern lässt. Außerdem wird festgestellt ob es bereits Tools gibt deren Einsatz die Qualität einer Service-orientierten Architektur steigern können. Aufbauend auf diesen Resultaten wird ein Prozess entwickelt, der die zuvor identifizierten Qualitätskriterien erfüllt und zusätzlich eine effiziente Überwachung von heterogenen Systemlandschaften bietet. Anschließend soll eine Analyse alle Prozessmodelle auf ihre Testfähigkeiten für verteilte Systeme vergleichen und bewerten.
Die Ergebnisse dieser Diplomarbeit sind ein neues Prozessmodell zur Spezifikation, Implementierung und Überprüfung einer Service-orientierten Architektur sowie die Einführung eines Konzeptes für effizientes Monitoring dieser heterogenen Systemlandschaft. Des Weiteren wurden die Testprozesse aller gängigen Software-Modelle auf Eignung für die Überprüfung von Service-orientierten Architekturen analysiert und mit Hilfe einer zuvor definierten Metrik bewertet. Die Ergebnisse dieses Vergleichs werden ebenfalls in dieser Diplomarbeit präsentiert.

Inhaltsverzeichnis

Erklärung i

Abstract ii

Kurzfassung iii

Inhaltsverzeichnis v

Abbildungsverzeichnis viii

Tabellenverzeichnis ix

1 Einführung 1
- 1.1 Motivation . . . 1
- 1.2 Definitionen . . . 2

2 Grundlagen 5
- 2.1 Qualitätssicherung von Software . . . 5
- 2.2 Verteilte Systeme . . . 8
- 2.3 Service-orientierte Architektur . . . 10
- 2.4 Event-basierte Systeme . . . 13
- 2.5 Workflow Management & Applications . . . 14
- 2.6 Mashup Applikationen . . . 17
- 2.7 Monitoring & Reporting für verteilte Systeme . . . 20
- 2.8 Software-Tests und Dokumentation . . . 22
- 2.9 Bestehende Software-Entwicklungsprozesse . . . 25
  - Das Wasserfallmodell . . . 25
  - Das V-Modell . . . 28
  - Das W-Modell . . . 30
  - Das Spiralmodell . . . 31
  - IBM Rational Unified Process . . . 34
  - Open Unified Process . . . 39
  - Extreme Programming (XP) . . . 41
  - Crystal Family . . . 44

Scrum . . . 46
Kanban . . . 48
Modellgetriebene Softwareentwicklung . . . 48
Feature Driven Development . . . 50
Test Driven Development . . . 52

3 Related Work **55**
3.1 Analyse bestehender Testkonzepte für Service-orientierte Architekturen . . . 55
SOA Test Methodik . . . 55
SOA Testing Governance . . . 58
Testen von Mobile Devices . . . 59
Integration von White- und Blackbox Testen . . . 59
Reflektives Architektur-basiertes Software Test Management Modell . . . 59
BPA Simulierte Ereignis Proxies . . . 60
Fortlaufende Integrationstests von Service-orientierten Architekturen . . . 60
Architektur für Monitoring von Laufzeitumgebungen . . . 61
Testen mit Hilfe von negativen Testcases . . . 61
Testwerkzeuge für SOA . . . 61
3.2 Analyse bestehender Schnittstellentests für SOA . . . 62
seekda! Suchdienst . . . 62
WSDL . . . 63
jUnit-Tests . . . 64

4 Prozess zur Qualitätssicherung von Schnittstellen **67**
4.1 Definition des Sensorenkonzeptes . . . 68
4.2 Anforderungen an den Entwicklungsprozess . . . 68
Anforderungen an die Dokumentation . . . 69
Anforderungen an den Testprozess . . . 71
Anforderungen an das Test-Team . . . 73
Anforderungen an den Aufbau des Prozesses . . . 74
Anforderungen an die Sicherheit . . . 74
Anforderungen an die Qualität . . . 74
Anforderungen an die Projektleitung . . . 76
4.3 Qualitätssicherungsmaßnahmen . . . 76
Konzeptuelle Qualitätssicherungsmaßnahmen . . . 77
Technische Qualitätssicherungsmaßnahmen . . . 79
4.4 Prozess zur Qualitätssicherung: Das WASI-Modell . . . 80
Business-Requirements & Zeitplan . . . 82
Guideline für Sicherheitstests . . . 83
Vorbereitung Abnahmetest . . . 83
Funktionaler Systementwurf . . . 84
Vorbereitung Systemtest . . . 84
Technischer Systementwurf . . . 85
Vorbereitung Integrationstests . . . 86

Spezifikation der Sensoren . . . 86
Service-Spezifikation . . . 87
Erstellung der Service-Level-Tests . . . 88
Komponentenspezifikation . . . 88
Erstellung der Komponententests . . . 88
Programmierung & Änderungen . . . 89
Erneuter Komponententest . . . 91
Erneuter Servicetest . . . 91
Integrationstest . . . 92
Systemtest . . . 92
Abnahmetests . . . 92
Abschließende Bemerkungen . . . 92
4.5 Architektur eines Monitoring-Frameworks . . . 93
4.6 Auswahl von Dimensionen für SOA Reporting . . . 95
4.7 Beispiele für den Einsatz der Sensoren . . . 96
Amazons EC2 Cloud Service . . . 97
Smartphones . . . 98
Militärische Zwecke . . . 98
Rettungseinsätze . . . 99
Automobilindustrie . . . 100
Computerspielindustrie . . . 101
Überwachung von Atomkraftwerken . . . 101
Überwachung von Lagerbeständen . . . 102
Gesundheitsüberwachung . . . 103
Videoüberwachung . . . 104
Fazit für die Einsatzmöglichkeiten . . . 104
4.8 Proof of Concept: WASI-Modell . . . 105
Business-Requirements . . . 105
Vorbereitung Abnahmetest . . . 107
Funktionaler Systementwurf . . . 107
Vorbereitung Systemtest . . . 108
Technischer Systementwurf . . . 109
Vorbereitung Integrationstests . . . 110
Sensorenspezifikation . . . 111
Service-Spezifikation . . . 111
Erstellung der Service-Level-Tests . . . 111
Komponenten-Spezifikation . . . 112
Erstellung der Komponententests . . . 112
Verteilte Implementierung inklusive der Testphasen . . . 113
Fazit: Proof of Concept . . . 114

5 Benchmark der Testprozesse aller Modelle 117
5.1 Parameter, Metrik & Bewertungsgrundlagen . . . 117

Definition der Parameter für den Benchmark . . . 117
Definition der Metrik . . . 118
Bewertungsgrundlagen . . . 119
5.2 Benchmark-Ergebnisse . . . 119
Bewertung: Wasserfallmodell . . . 119
Bewertung: V-Modell . . . 120
Bewertung: W-Modell . . . 121
Bewertung: Spiralmodell . . . 122
Bewertung: IBM Rational Unified Process . . . 123
Bewertung: Open-Unified Process . . . 124
Bewertung: Extreme Programming . . . 124
Bewertung: Crystal Family . . . 125
Bewertung: Scrum . . . 126
Bewertung: Kanban . . . 127
Bewertung: Modellgetriebene Entwicklungsmethode . . . 128
Bewertung: Feature Driven Development . . . 129
Bewertung: Test Driven Development . . . 129
5.3 Vergleich der Bewertungen . . . 130

6 Zusammenfassung **133**
6.1 Ergebnis & Bewertung . . . 133
6.2 Ausblick . . . 134

A Appendix **135**
A.1 Ergebnisse des Prozessvergleichs . . . 135

Literaturverzeichnis **139**

Abbildungsverzeichnis

2.1 PDCA-Zyklus (nach [NCL10]) . . . 6
2.2 Qualitätsmanagement Prozess Modell (nach [ISO08]) . . . 8
2.3 Vor und nach Einführung der SOA (aus [SUN]) . . . 11
2.4 Service-orientierte Architektur (nach [Har07]) . . . 12
2.5 Workflow Management Systeme in historischer Betrachtung (nach [Aal98]) . . . 15
2.6 Petri-Netz Beispiel: Bearbeitung einer Beschwerde (nach [Aal98]) . . . 16
2.7 Aufbau eines Mashup-Stacks (nach [Ger07]) . . . 18

2.8 Monitoring-Architektur (nach [Kur00]) . . . 20
2.9 Testdokumentation innerhalb eines Projekts(nach [IEE08]) . . . 24
2.10 Einfaches Wasserfallmodell (nach [Roy70]) . . . 26
2.11 Erweitertes Wasserfallmodell (nach [Roy70]) . . . 27
2.12 V-Model (nach [Boe79]) . . . 28
2.13 W-Modell (nach [SRWL06]) . . . 30
2.14 Spiralmodell mit vier Durchläufen (nach [Boe88]) . . . 32
2.15 Ein iterativer und inkrementeller Prozess (nach [Kru04]) . . . 35
2.16 Phasen, Iterationen und Arbeitsschritte im "Unified Process" (nach [JBR99]) . . . 36
2.17 Phasen, Iterationen und Arbeitsschritte im IBM Rational Unified Process (nach [Kru04]) . . . 37
2.18 OpenUP Delivery Process (nach [Ecl]) . . . 41
2.19 Extreme Programming Projekt (nach [Wel99]) . . . 42
2.20 Verwendete Methoden bei Extreme Programming (nach [Bec99]) . . . 44
2.21 Der Scrum-Prozess (nach [Sch07]) . . . 47
2.22 Modellgetriebene Entwicklung - Vom Modell zum Code (nach [SVEH07]) . . . 49
2.23 Modellgetriebene Entwicklung - System- und Testentwicklung (nach [BSGS04]) . 50
2.24 Feature-Driven Development Prozessmodell (nach [CLL99]) . . . 51
2.25 Test Driven Development - Ablauf (nach [Bec02]) . . . 54

3.1 SOA Komponenten inkl. Test-Levels (nach [Har07]) . . . 57

4.1 Identifizierte Anforderungen an den Entwicklungsprozess . . . 69
4.2 Qualitätssicherungsmaßnahmen für den Entwicklungsprozess . . . 77
4.3 Well Assisted SOA Implementation Model (WASI) . . . 81
4.4 Architektur für ein Framework für effizientes SOA Monitoring . . . 93
4.5 Skizze für "Advanced Health Monitoring System" . . . 105
4.6 Anwendungsfalldiagramm für "Advanced Health Monitoring System" . . . 107
4.7 Vereinfachtes BPMN-Diagramm für "Advanced Health Monitoring System" . . . 108
4.8 Klassendiagramm eines AHMS-Service . . . 109
4.9 Schnittstellen zwischen den AHMS-Services . . . 110

5.1 Vergleich der Ergebnisse . . . 131

Tabellenverzeichnis

2.1 Crystal Family (nach [dev]) . . . 45

4.1 Beispiel - Benötigte Sensoren für die Testfälle 113

A.1 Ergebnisse des Vergleichs aus Kapitel 5 (Tabelle 1/2) 136
A.2 Ergebnisse des Vergleichs aus Kapitel 5 (Tabelle 2/2) 137

KAPITEL 1

Einführung

Dieses Kapitel nennt im ersten Abschnitt die Gründe für die Entwicklung des eigenen Prozessmodells und definiert im zweiten Abschnitt die für das Verständnis dieser Arbeit notwendigen Begriffe.

1.1 Motivation

Getrieben durch die organisatorischen Veränderungen in Unternehmen müssen sich auch die Systemlandschaften anpassen. Heutzutage sind Anwendungen nicht mehr zentral verwaltbar, sondern verteilen sich oft auf unterschiedlichste Standorte oder organisatorische Einheiten. Erst durch den Zusammenschluss dieser Services kommt es zu einem funktionierten Geschäftsprozess, umso wichtiger stellt sich die Qualitätssicherung in den einzelnen Bausteinen und der Komposition dieser dar. Es muss also nicht nur sichergestellt werden, dass jede Komponente für sich funktioniert, sondern auch, dass das Zusammenspiel die erwarteten Resultate liefert. Da diese meist von unterschiedlichen organisatorischen Einheiten beziehungsweise von unterschiedlichen Unternehmen entwickelt werden ist die Sicherung der Qualität eine größere Herausforderung als wenn nur Software ohne Schnittstellen zu anderen Systemen entwickelt wird. Aktuelle Softwareentwicklungsprozesse decken Qualitätsanforderungen verteilter Systeme, wie Sicherheitsaspekte oder die Überprüfung der Einhaltung von zuvor spezifizierten Unternehmensrichtlinien innerhalb einer Service-orientierten Architektur, nur begrenzt ab. Diese Erkenntnis konnte durch ein Gespräch mit drei Entwicklern, die seit einigen Jahren Service-orientierte Architekturen für verschiedene Firmen implementieren, weiter gefestigt werden.
Die hohe Komplexität heterogener Systemlandschaften, welche sich über Unternehmensgrenzen hinweg erstrecken können, erschwert die Qualitätssicherung enorm: Fehler in in diesen Systemen sind nicht einfach zu lokalisieren und Ausbesserungen schwer durchzuführen. Nach intensiver Begutachtung ausgewählter Prozessdokumentationen (siehe Kapitel 2.9) erschließt sich, dass aus folgenden Gründen die Überprüfung einer Service-orientierten Architektur für diese betrachteten Modelle schwer möglich ist:

- Fehlende Dokumentation des Systementwurfs: Viele Prozesse verzichten auf eine für verteilte Systeme unerlässliche Dokumentation der gesamten Software-Umgebung. Sind wenige Informationen über das System vorhanden, lässt es sich entweder überhaupt nicht oder nur teilweise prüfen, dieser Umstand kann zu mangelhafter Qualität führen.

- Mangelhafte Beschreibung der verschiedenen Testarten: Genügt die Dokumentation den Anforderungen an Service-orientierte Architekturen, wird oftmals die Vorgehensweise bei verschiedenen Testarten (wie beispielsweise Integrations- oder Systemtests) von den einzelnen Modellen nicht oder nur mangelhaft beschrieben. Überprüfungen können somit nur mit einem hohen Einsatz an Kreativität durchgeführt werden, was bei komplexen heterogenen Systemen zu weiteren Problemen führt.

- Keine Methoden zur Überwachung des verteilten Systems während der Ausführung: Ist das System und die Testumgebung ausreichend dokumentiert und spezifiziert, kann in der Überprüfungsphase kein zentralisiertes Monitoring der gesamten Umgebung durchgeführt werden. Ein Konzept zur Überwachung der einzelnen Services und Testfälle während der Laufzeit für eine Service-orientierten Architektur fehlt.

- Unbekannte Zustände der einzelnen unabhängigen Dienste: In einem großen System mit vielen unabhängig arbeitenden Diensten kann die korrekte Funktionalität der einzelnen Komponenten nur mühsam überwacht werden, auch hierfür fehlt ein Konzept.

Um die bestehende Situation zu verbessern untersucht und evaluiert diese Arbeit bestehende Prozesse der Softwareentwicklung inklusive ihrer Qualitätssicherungsmaßnahmen in Hinblick auf die speziellen Anforderungen verteilter Systeme und stellt anschließend einen adaptierten Prozess mit Qualitätssicherungsmaßnahmen vor, der auf die spezifischen Charakteristika zugeschnitten ist. Anschließend wird dieses Prozessmodell mit den bestehenden Entwicklungsprozessen verglichen und Stärken/Schwächen des zentralen Ergebnisses dieser Arbeit ermittelt und dokumentiert.

1.2 Definitionen

Nachfolgend werden die bestehenden Definitionen einiger Begriffe erläutert, welche fundamental für das Verständnis dieser Arbeit sind:

Softwareentwicklungsprozess (auch Entwicklungsprozess oder Prozess genannt): ein Vorgangsmodell bestehend aus den konkreten Aktivitäten und deren Abfolge sowie den angewandten Methoden [GKZ04].

Representational State Transfer (kurz: REST): eine Software-Architektur für verteilte Informationssysteme wie beispielsweise dem World Wide Web. Der Begriff wurde erstmals von Roy Fielding im Jahre 2000 in seiner Dissertation definiert [Fie00].

Web-orientierte Architektur (kurz: WOA): stellt eine Erweiterung der Service orientierten Architektur (siehe Kapitel 2.3) für web-basierte Anwendungen dar, welche meistens mit Hilfe der REST-Architektur für eine leichtere Kommunikation sorgt [Ger07].

Agile Softwareentwicklung: bezeichnet ein Verfahren, bei welchem mit geringem Dokumentationsaufwand und einer geringen Menge an Vorschriften vorgegangen wird, der Fokus liegt auf dem Ziel [Coc03].

Error, Fault, Failure: Als Error definiert man eine Handlung eines Menschen (Programmierer oder Anwender), welche ein inkorrektes Resultat liefert. Ein Fault ist ein inkorrekter Ausführungsschritt, ein Prozess oder eine inkorrekte Datendefinition in einer Komponente oder einem System, welche die Komponente oder das System möglicherweise in ihrer geforderten Funktionsweise stört. Ein Failure ist schließlich die Auswirkung eines Faults auf das Service. Durch die fehlerhafte Funktionsweise kommt es zu einem inkorrekten unerwarteten Resultat [Par97].

Web-Service: eine Applikation, welche die direkte Interaktion zwischen Maschinen (unabhängig von den eingesetzten Betriebssystemen) über ein Netzwerk unterstützt [HB04].

Unit-Test (auch Modultest genannt): überprüfen die Funktionalität einer einzelnen Einheit, welche in der objektorientierten Programmierung oft äquivalent zu einer Methode ist [Ham04].

Integrationstest: testet die Funktionalität der Kombination verschiedener Komponenten eines Software-Systems [IEE91].

Systemtest: testet das ganze System gegen die Anforderungen, welche meist zu Beginn des Projektes ermittelt werden. Er wird von der Entwicklungsfirma durchgeführt und soll die Arbeitsumgebung des Kunden simulieren [IEE91].

Akzeptanztest (auch Abnahmetest genannt): ist im Testprozess die letzte Stufe: Die Kunden testen die fertige Software im Produktiveinsatz und stellen Mängel fest und/oder äußern letzte Änderungswünsche [IEE91].

Geschäftsprozess & -modell & Geschäftsobjektmodell: Ein "Business" Prozess ist eine Sammlung zusammenhängender, strukturierter Aktivitäten (Tasks), welche ein bestimmtes Service oder Produkt als Ergebnis haben. Ein Geschäftsobjektmodell ist eine Abstraktion eines Geschäftsprozesses in ein Modell, dessen Struktur den zeitlichen und logischen Ablauf widerspiegelt. Ein Geschäftsobjektmodell bezeichnet die Summe aller Geschäftsobjekte inkl. derer Assoziationen untereinander [SS07].

Unified Modeling Language-Diagramme: Das UML-Klassendiagramm bildet die vorhandenen Klassen einer Software inklusive ihrer Instanzvariablen und möglicher Wertebereiche ab. Des Weiteren werden die Schnittstellen definiert, die Vererbung abgebildet und die Beziehung zwischen den Klassen dargestellt. Ein Anwendungsfalldiagramm repräsentiert das Verhalten, welches von der Software erwartet wird, mit Hilfe der Darstellung von Anwendungsfällen und Akteuren mit ihren Assoziationen und Dependencies. Ablaufdiagramme helfen Programmierer den geplanten Programmablauf zu modellieren [Kec11].

Interoperabilität: ist die Fähigkeit eines Systems oder Produkts mit anderen Systemen oder Produkten ohne besonderen Aufwand auf der Seite des Kunden zusammenzuarbeiten [Har07].

Deadlocks: entstehen unter bestimmten Bedingungen, wobei zwei oder mehr Prozesse jeweils auf die Freigabe der benötigten Ressource durch einen Prozess warten, es jedoch niemals zu Dieser kommt und sich die Prozesse in einer Endlosschleife befinden. Ein weiteres Problem bei der parallelen Verarbeitung mehrerer Prozesse stellt die so genannte Race Condition dar. Zwei Prozesse oder Threads arbeiten mit der gleichen Ressource (bspw. eine Systemvariable) und je nach zeitlichem Ablauf beider Prozesse kann das Ergebnis ein Anderes sein, dies führt zu unerwarteten Ergebnissen und somit unerwünschtem Verhalten [Wes98].

KAPITEL 2

Grundlagen

In diesem Kapitel werden die essentiellen Grundlagen für die vorliegende Arbeit analysiert. Nachfolgend wird der Begriff Qualitätssicherung von Software näher erläutert und moderne Qualitätssicherungsmaßnahmen vorgestellt. Des Weiteren folgt eine Einführung in die Themen "Verteilte Systeme" (Distributed Systems) und "Service-Orientierte Architekturen" (SOA) um den eigens entwickelten Softwareprozess besser verstehen zu können. Weitere wesentliche Grundlagen stellen "Event-basierte Systeme" und "Workflow-Applikationen" dar, die ebenfalls detailliert in diesem Kapitel behandelt werden. Es folgt eine Präsentation der Themen "Mashup-Applikationen" und dem "Monitoring & Reporting", welche fundamentale Begriffe zum Verständnis dieser Diplomarbeit darstellen. Auch auf das Testen von Software wird in diesem Kapitel Bezug genommen. Die Vorstellung der Grundlagen wird schließlich durch den letzten Punkt, den aktuellen bestehenden Software-Entwicklungsprozessen, abgeschlossen.

2.1 Qualitätssicherung von Software

Nach [ISO08] ist die Qualität einer Einheit der Grad, in dem ein Satz inhärenter Merkmale Anforderungen erfüllt. Unter einer Einheit versteht man ein Produkt oder eine Dienstleistung, Merkmale sind kennzeichnende Eigenschaften einer Einheit. Unter Qualität versteht man also die Zufriedenstellung der Kunden durch die Erfüllung ihrer Wünsche, Anforderungen und Erwartungen.
Der Begriff Qualitätsmanagement (kurz: QM) bezeichnet alle Maßnahmen des Managements zur Abstimmung der qualitätsrelevanten Tätigkeiten (Leitung und Lenkung eines Unternehmens, Festlegung der Qualitätspolitik und -ziele, anschließende Verwirklichung dieser durch Qualitätsplanung, -lenkung, -sicherung und -verbesserung) zur Zufriedenstellung der Anforderungen und/oder der Wünsche der Kunden [BW08].
Qualitätssicherung hilft Fehler zu vermeiden und somit eine Kostenreduktion zu erreichen. Des Weiteren kommt es zu einer Erhöhung der Kundenzufriedenheit und zu einer Steigerung der Produktivität im Unternehmen. Dies führt auch zu einer Verbesserung des Gewinns und der Stabilität

des Unternehmens. Qualitätssicherung bedeutet Absicherung für die Zukunft. Qualitätssicherung hilft auf lange Sicht die Kosten zu reduzieren, jedoch verursacht sie auch welche. Unter qualitätsbezogenen Kosten versteht man die Ausgaben, welche für den Erhalt, die Sicherung und die Verbesserung des vom Kunden erwarteten Qualitätslevels, verwendet werden.
Eine bekannte Strategie des QM nennt sich "Kontinuierlicher Verbesserungsprozess" (kurz KVP,

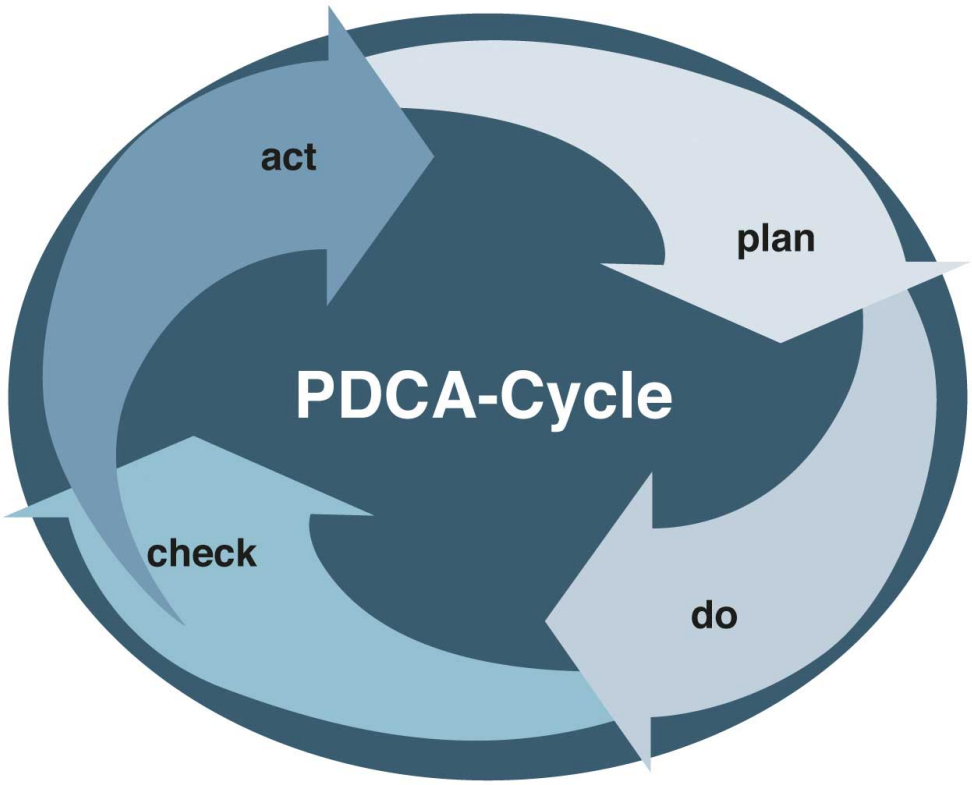

Abbildung 2.1: PDCA-Zyklus (nach [NCL10])

abgeleitet vom japanischen KAI-ZEN ==> Veränderung zum Besseren): Fehler müssen hierbei sofort nach ihrer Entstehung behoben werden und der Prozess hat kein Ende. Der so genannte Plan-Do-Check-Act Zyklus (PDCA oder Deming-Kreis) wird hierbei ständig durchlaufen. Dieser wird in Abbildung 2.1 illustriert und auf die einzelnen Elemente wird in den folgenden Zeilen näher eingegangen [NCL10]:

- Plan: In dieser Phase werden die Ziele des zu erstellenden Produkts definiert. Die Fragen, was, warum, wer, wo und wie das Erzeugnis erstellt wird, müssen beantwortet werden. Es kommt also zu einer Analyse des Problems bzw. zu einer Verbesserungsplanung nach der ersten Iteration.

- Do: Der Plan bzw. die Verbesserung wird umgesetzt. Bei der Verbesserung muss zu

jeder Zeit auf die Einhaltung der Definitionen des Verbesserungsplans aus der ersten Phase geachtet werden.

- Check: Kontrolle bzw. Messung, die Wirksamkeit wird überprüft.
- Act: Definiert eine Standardisierung der verwendeten Methoden und weitere Anpassungen.

Es gibt unzählige Werkzeuge zur Qualitätssicherung. Die Bekanntesten unter ihnen werden jetzt vorgestellt. Zur Erfassung von Fehlern sind eine Fehlersammelliste, ein Histogramm und Qualitätsregelkarten (Dokumentation durch regelmäßige Stichproben) empfehlenswert. Zur Fehleranalyse sollte ein Paretodiagramm (Statistische Fehlerauswertung), ein Ursache-Wirkungsdiagramm (auch Ishikawa-Diagramm genannt, untersucht die Ursachen für mögliche Fehler), Ein Brainstorming (spontane Sammlung von Ideen ohne einer Bewertung) oder ein Korrelationsdiagramm (Grafische Darstellung von paarweisen Merkmalen) verwendet werden.
Auch das Management bedient sich an Werkzeugen zur Qualitätssicherung. Probleme werden beispielsweise durch ein Relationen- (Ursachen eines zuvor definierten Problems werden aufgezeichnet) oder Affinitätsdiagramm (Brainstorming mit darauf folgender Einordnung der gefundenen Ideen in Gruppen) analysiert. Anschließend werden Lösungen gesucht und bewertet: Ein Matrix- (Stärke der Problemfaktoren wird dargestellt) und Baumdiagramm (ein gefundenes Problem wird zur besseren Übersicht in Teilprobleme aufgespaltet und analysiert) sind dabei genauso hilfreich wie die Erstellung eines Portfolios (gesammelte Daten werden qualitativ gegenübergestellt). Zur Planung der Umsetzung eignet sich ein Entscheidungsbaum (Definition des Ziels und möglicher Maßnahmen zu dessen Erreichung) oder ein Netzplan (der zeitliche Ablauf eines Projektes inkl. aller Aktivitäten wird grafisch illustriert).
Nun werden einige moderne Qualitätssicherungswerkzeuge vorgestellt, wie sie heute oft in der Praxis eingesetzt werden:

- Fehlermöglichkeits- und Einflussanalyse (FMEA): Beschreibt eine Vorgangsweise zur systematischen Fehlerfindung bei einem Produkt.
- Six Sigma: Kontinuierliche Verbesserung wird mit den fünf Schritten "Define - Measure -Analyze - Improve - Control" umgesetzt. Des Weiteren existiert eine zweite Variante von Six Sigma namens"DMADV", welche die Vorgehensweise "Define - Measure - Analyse - Design - Verify" definiert.
- Quality Function Deployment (QFD): Die Kundenanforderungen werden festgelegt. Danach werden Konkurrenzprodukte auf die Erfüllung dieser Anforderungen überprüft. Im nächsten Schritt werden die Merkmale des zu entwickelnden Produkts festgelegt. Die festgelegten Merkmale und die Anforderung der Kunden werden korreliert. Es folgt ein technischer Wettbewerbsvergleich und die gegenseitige Beeinflussung der Merkmale wird berücksichtigt.

Managementsysteme koordinieren die Zusammenarbeit der Arbeitskräfte, ermöglichen einen optimalen Ablauf und kreieren Vertrauen gegenüber Kunden, Mitarbeitern und Inhabern eines Unternehmens. Die ISO 9001 [ISO08] spezifiziert ein komplettes Prozessmodell für das Qualitätsmanagement, welches in der Abbildung 2.2 dargestellt wird. Der oben vorgestellte PDCA-Zyklus

wird für das Qualitätsmanagement konkretisiert.
Der letzte Begriff, der in diesem Kapitel erklärt werden soll, ist "Total Quality Management"

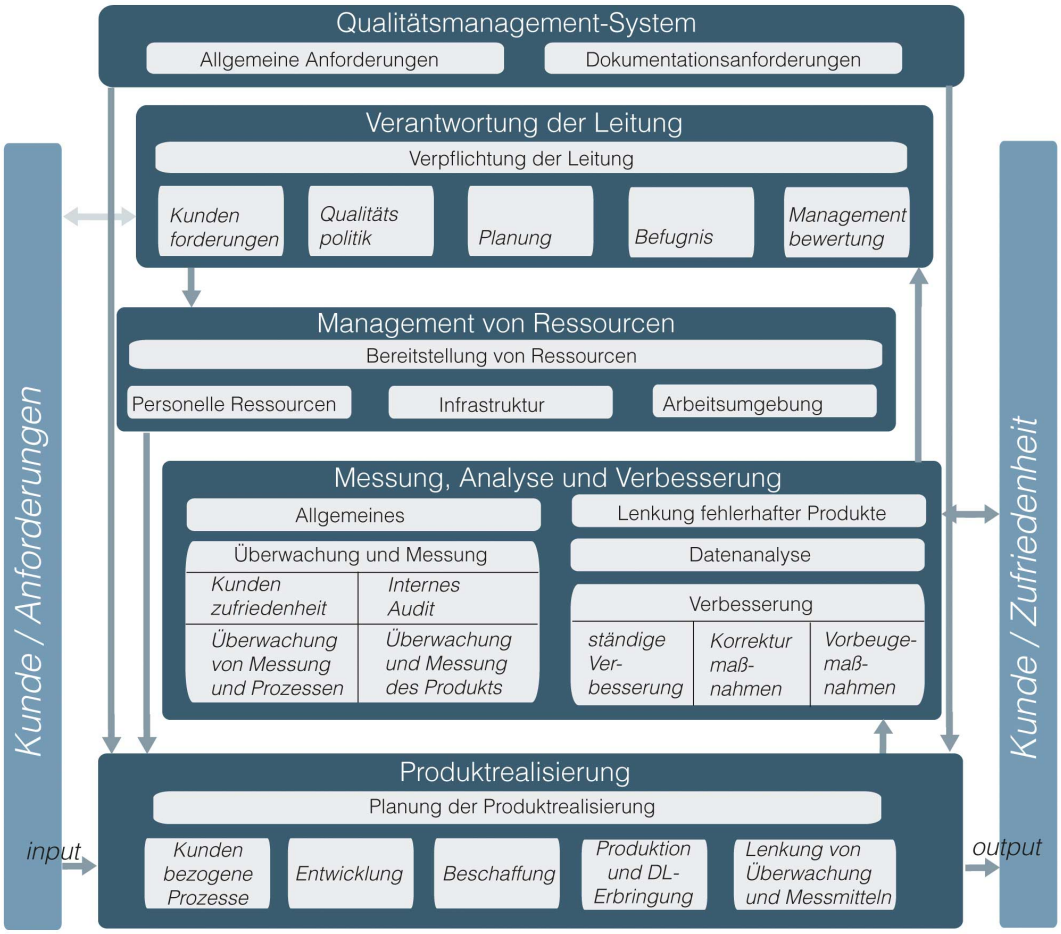

Abbildung 2.2: Qualitätsmanagement Prozess Modell (nach [ISO08])

(TQM). Es geht darum, die Qualität der Produkte, der Prozesse und das sich daraus ergebende Potenzial als zentrales Thema der gesamten Führung unternehmensweit zu definieren. Es lässt sich als System aus den in diesem Kapitel erwähnten Werkzeugen und Methoden beschreiben.

2.2 Verteilte Systeme

Nach [TS07] ist ein Verteiltes System (engl.: Distributed System) ein Zusammenschluss mehrerer Computer, die sich für den Anwender wie ein einziges System darstellen.
Es gibt verschiedenste Anwendungsgebiete für verteilte Systeme, wie beispielsweise das Internet [W3C]. Verteilte Systeme lassen sich sowohl zur Problemlösung (z.B. Datentransfer über örtlich

entfernte Rechner) und/oder zum Vorteil durch spezifische Systemeigenschaften verwenden, die nun näher erläutert werden [VKZ05]:

- Business Integration: Die informationstechnische Kommunikation und Zusammenarbeit mehrerer Unternehmen kann sicher gestellt werden.
- Security (Sicherheit): Die zentrale Speicherung von wichtigen Daten (wie beispielsweise Zugangsdaten) kann bei Ausfall des Servers zu massiven Problemen führen, deshalb ist eine Verteilung wichtiger Daten über mehrere Standorte durchaus sinnvoll.
- Geographische Unabhängigkeit der Services und Clients: In Systemen, in welchen Services und Clients öfter den Betriebsstatus (online, offline, standby, ...) wechseln, ist der Einsatz eines verteilten Systems ebenfalls anzuraten.
- Fault Tolerance (Toleranz gegenüber Fehlern): Fällt ein Service aus hardware- oder softwaretechnischen Gründen aus, führt dies bei einem verteilten System, nicht unweigerlich zur Unerreichbarkeit, es existiert eine gewisse Fehlertoleranz.
- Load Balancing: Eine komplexe Rechenaufgabe kann beispielsweise effizient durch die Kollaboration mehrere Rechner gelöst werden, ein verteiltes System unterstützt diesen Vorgang und kann die Performance eines Service steigern.
- Bessere Wartbarkeit und kostengünstige Bereitstellung: Große Systeme mit vielen Clients sind kostenintensiv und aufwändig in der Wartung. Clients, die durch ein verteiltes Service berechneter Ergebnisse nur anzeigen, sind eine Lösung für dieses Problem.

Es existieren verschiedene Software-Architekturen für Verteilte Systeme: Daten-zentrierte, Datenfluss-basierte, Call-and-Return und komponentenunabhängige Ansätze, auf welche nachfolgend näher eingegangen wird [DGH03]:
Bei den auf Daten zentrierten Architekturen wird der Fokus auf das Lesen und Schreiben eines Repositories, welches das elementare Bestandteil der Kommunikation mit den Clients darstellt, gelegt. Dieser Datenbehälter kann passiv in Form eines reinen Datenspeichers oder aktiv implementiert werden. Bei der zuletzt genannten Methode werden die Clients per Nachricht über die Datenänderung(en) informiert. Clients und Server sind hierbei voneinander unabhängig, das Hinzufügen oder Entfernen zusätzlicher Clients ändert nichts am bestehenden System.
Bei Datenfluss-basierten Architekturen liegt der Schwerpunkt auf der Modifizierbarkeit und der Wiederverwendbarkeit von Daten. Die Datenverarbeitung geschieht durch verschiedene Komponenten, die entweder in einem sequenziellen Batch-Prozess oder mit der Pipe-and-Filter Methode arbeiten. Bei ersterer Vorgangsweise verarbeiten die unabhängigen Komponenten die Daten erst, nachdem der gesamte Eingabestrom gelesen wurde. Erst nach der vollständigen Abarbeitung werden die Daten an die nächste Komponente weitergeschickt. Bei Pipe-and-Filter wird im Gegensatz nicht auf die komplette Eingabe gewartet, das heißt es können mehrere Komponenten gleichzeitig an der Datenverarbeitung beteiligt sein. Komponenten können somit leicht ausgetauscht werden.
Die Call-and-Return Softwarearchitektur baut auf einem Kontrollfluss auf, der jede Komponente nach ihrer bestimmten hierarchischen Ordnung nacheinander aufruft. Die parallele Verarbeitung

der Daten wird durch eine Aufteilung verschiedener Komponenten auf verschiedenen Rechnern gewährleistet. Werden Komponenten auf unterschiedliche Schichten hierarchisch aufgeteilt, spricht man von einer "Layered Architecture", diese ist allerdings nicht überall praktisch einsetzbar.
Die Architektur der unabhängigen Komponenten basiert auf der Existenz von Prozessen, die durch den Austausch von Nachrichten miteinander kommunizieren. Event-basierte Systeme bauen beispielsweise auf dieser Architektur auf. Eine Komponente (Publisher) verschickt Informationen, alle Komponenten, die diese erhalten wollen, müssen sich hierfür registrieren (Subscribers). Ein eigener Dienst verwaltet die asynchrone Kommunikation, das System ist völlig entkoppelt, wodurch sich mögliche Änderungen leicht umsetzen lassen. In modernen Softwaresystemen sind meist mehrere Architekturstile zu finden.
Es gibt auch einige Probleme, die bei dem Entwurf und der Konstruktion eines verteilten Systems berücksichtigt werden müssen [VKZ05]:

- Mögliche Fehler in Teilen des Systems: Sollte aufgrund eines Hardware- oder eines Softwarefehlers (möglicherweise auch vom Betriebssystem ausgelöst) ein Fragment eines Verteilten Systems nicht mehr korrekt arbeiten, muss eine Ersatzlösung gefunden werden. Das System muss somit eine intelligente Fehlerbehandlung beinhalten.

- Mögliche Latenzen bei der Kommunikation über Netzwerk: Sendevorgänge entfernter Computern sind langsamer als Nachrichten lokaler Dienste. Auf diesen Umstand muss besonders bei zeitkritischen Distributed Systems bei der Implementierung geachtet werden.

- Es ist sehr schwer feststellbar, wann welche Komponenten des verteilten Systems welche Methoden ausführen. Die Bezeichnung für dieses Verhalten ist unter dem Begriff Nebenläufigkeit definiert und ist die Ursache für Probleme wie Race Conditions oder Deadlocks.

- Die Software und die Hardware eines verteilten Systems müssen zu jeder Zeit in der Größe anpassungsfähig bleiben, weil sich die Belastungen auf gewisse Komponenten des Distributed Systems laufend verändern können, als Beispiel hierfür sei auf die schwankenden Besucherzahlen von einigen Websites verwiesen.

- Vorhersagbarkeit: Es ist schwierig vorauszusagen, wie lange ein Aufruf in einem Verteilten System dauert, weil verschiedene Aspekte wie die Leistungsfähigkeit des angesprochenen Rechners und etliche andere Faktoren berücksichtigt werden müssen. Auch ist die Kommunikation nicht zu jeder Zeit verlässlich.

2.3 Service-orientierte Architektur

Die Service-orientierte Architektur (kurz: SOA) stellt eine mögliche Implementierung eines verteilten Systems, welches im letzten Kapitel vorgestellt wurde, dar. Sie ist so zu sagen als Architekturparadigma für Umgebungen der Informationstechnologie zu verstehen. Der Begriff fand in einem wissenschaftlichen Artikel von Gartner zum ersten Mal Verwendung [Gar96].
Die Service-orientierte Architektur baut auf dem Konzept von kleinen, wiederverwendbaren Services auf, welche die Wartungskosten und die laufenden Betriebskosten reduzieren können.

Die Wiederverwendung der Dienste spart Implementierungsaufwand und macht Änderungen an bestehenden Systemen einfacher umsetzbar. Die Kompaktheit der Services führt auch zu einer Reduktion der Komplexität großer Softwaresysteme. Einzelne Komponenten der SOA werden durch das Zusammensetzen einzelner Dienste produziert, dies steigert die Agilität des Entwicklungsprozesses enorm.
Die Service-orientierte Architektur spezifiziert folgende Kriterien für einen Dienst:

1. Abgeschlossenheit und Eigenständigkeit eines Service
2. Offene Schnittstellen für jeden Dienst
3. Gewährleistung der Interoperabilität
4. Zugriff auf das Service über ein Netzwerk muss möglich sein
5. Zentrale Suche der Services im System
6. Dynamische Einbindung der Dienste zur Laufzeit muss möglich sein

Diese Spezifikation wird in realen Projekten jedoch kaum vollständig beachtet, es wird mehr auf die Zusammensetzung der einzelnen Dienste, welche untereinander schwach gebunden sind, zu funktionsfähigen Business-Prozessen Wert gelegt [BKM07].
In der Abbildung 2.3 sieht man die Unterschiede zwischen einer nicht-Service-orientierten und

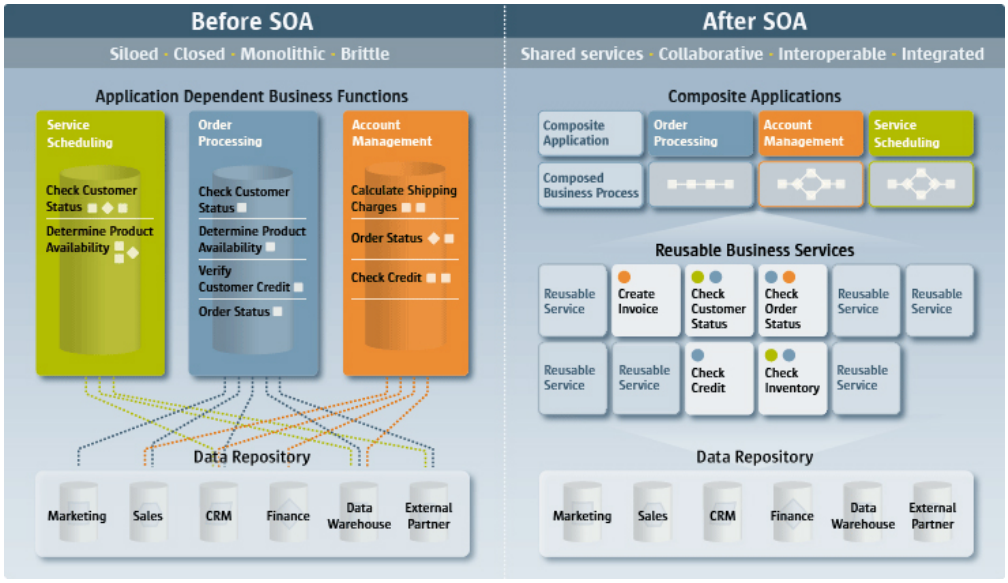

Abbildung 2.3: Vor und nach Einführung der SOA (aus [SUN])

einer SO-Architektur. Auf der linken Hälfte der Grafik sind die Prozesse, Aktivitäten und Daten, welche unabhängigen und inkompatiblen Applikationen zugeordnet sind, aufgelistet. Wenn ein Anwender einen Business-Prozess ausführen will, muss er die Applikationen wechseln und die

unterschiedlichen Datenstrukturen verstehen. Diese Missstände werden durch die SOA, welche auf der rechten Hälfte der Abbildung 2.3 dargestellt wird, behoben. Des Weiteren wird die Wartbarkeit der Software erheblich verbessert.
Die Kombination verschiedener Dienste in einer Anwendung ermöglicht dem Anwender eine stark vereinfachte Usability. Die Services werden zentral verwaltet, die vereinfachte Wartung führt zu einer erheblichen Zeit- und Kostenreduktion.
Bei der Implementierung von SOA-Services sollte man folgende Punkte beachten [Har07]:

- Die Business Prozesse müssen dokumentiert werden.
- Die Implementierung muss aus lose-gekoppelten Services bestehen um den höchstmöglichen Flexibilitätsgrad zu erreichen, Kosten dank Wiederverwendung einzusparen und die Wartbarkeit zu verbessern.
- Die Schnittstellen der Services sollten sich an Standards anlehnen um eine einfache Integration und Interoperabilität mit anderen Services zu erreichen.
- Das Business definiert die Services und die Services definieren die zu verwendende Technologie.
- Die Agilität im Business ist fundamental für Service-orientierte Architekturen.

Die Abbildung 2.4 zeigt den Aufbau einer Service-orientierte Architektur mit ihren Komponenten. Der Service Provider (Service) erstellt das Service und gibt Informationen über die Schnittstelle an das Service Directory weiter, welches wiederum für die Bereitstellung von Interface-Informationen an Service-Consumer zuständig ist. Der Service-Consumer holt sich die Einträge vom Service Directory um den Service Provider mit dem Service einzubinden [Har07].

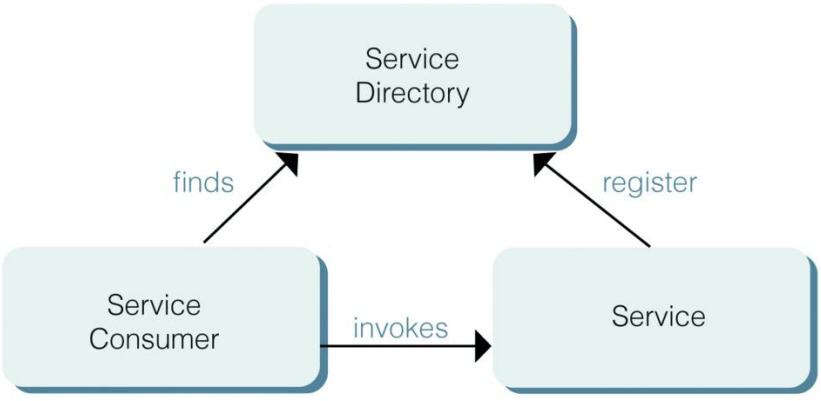

Abbildung 2.4: Service-orientierte Architektur (nach [Har07])

2.4 Event-basierte Systeme

Es existieren mehrere Definitionen vom Begriff der event-basierten Systeme (englisch: event-based systems oder event-driven systems), welche sich je nach Anwendung voneinander unterscheiden. Service-orientierte Architekturen werden auch oft in die Kategorie der "ereignisbasierten" Systeme eingeordnet. Diese werden in diesem Kapitel konkret vorgestellt.
Bei einem event-basierten System findet die Kommunikation zwischen den einzelnen Komponenten durch das Generieren und dem Empfang von so genannten Event Notifications (Ereignisbenachrichtigungen) statt. Ein Ereignis ist beispielsweise eine Statusänderung bei einer Komponente oder jedes andere Geschehen, welches für das System von Bedeutung sein kann. Eine Benachrichtigung ist eine Repräsentation des Events in Datenform. Es existieren verschiedenste Benachrichtigungen für ein Ereignis und auch verschiedene Darstellungen (Modelle) für diese. Auf unterster Ebene können Nachrichten (Messages) zur Notification eines Ereignisses verwendet werden. Sie repräsentieren die Daten und können mit Hilfe unterschiedlichster Kommunikationstechniken verschickt werden. Es kommt zu einer klaren Trennung der Darstellung der Daten und der Kooperationsmethode. Ein Event Notification Service erfüllt die Aufgabe Benachrichtigungen mit Hilfe von unterschiedlichen technischen Methoden unter Verwendung verschiedener Daten- und Filtermodelle zwischen den Komponenten eines ereignisbasierten Systems zu verteilen. Die Komponenten sind Produzenten und/oder Konsumenten von Benachrichtigungen. Erstere beschreiben mit Hilfe ihres Interfaces (Output-Schnittstelle) die Notification, die erzeugt wird, Zweitere spezifizieren eine Eingabe-Schnittstelle, welche die erwarteten Benachrichtigungen spezifiziert [FMG02].
Die Repräsentation der Daten (und der verwendeten Filter) lässt sich in drei verschiedene Klassen einteilen [FMG02]:

- Subject-based addressing: Der Erzeuger der Nachricht definiert einen Pfad von einem Knoten des Baums für das Thema zu jeder veröffentlichten Benachrichtigung und die Bezieher benutzen die Methode des pattern-matching für das Thema. Die Implementierung ist eher einfach, aber die Methode hat Grenzen.

- Type-based approach: Hier werden Hierarchietypen zur Filterung der Benachrichtigungen verwendet. Die Integration in bestehenden Projekte mit objekt-orientierter Programmiersprache gestaltet sich als unproblematisch.

- Content-based filtering: Hier werden Boolesche Ausdrücke zur Inhaltsfilterung für die Benachrichtigungen verwendet. Diese Methode ist von allen drei Genannten die Flexibelste.

Es wird im Folgenden kurz auf das Event-basierte Kooperationsmodell eingegangen, welches die Grundlage für ereignisbasierte Systeme darstellt. Die Kommunikation beginnt beim Ersteller der Benachrichtigungen, welcher auch Produzent genannt wird. Diese Notifications sind nicht an bestimmte Empfänger adressiert, Konsumenten können von verschiedensten Quellen die Nachrichten beziehen. Ein Produzent weiß nichts von seinen Konsumenten. Wenn ein Ereignis die Kriterien erfüllt, wird es geschickt. Innerhalb dieses Ereignis sollte auch der Zustand des Produzenten mitgeteilt werden, damit potentielle Konsumenten informiert sind. Berechnung und Kommunikation werden bei Event-basierten Systemen klar voneinander getrennt [FMG02].

Zum Abschluss dieses Kapitels folgt die Beschreibung eines exemplarisch, einfach aufgebauten Event-basierten Systems. Es besteht aus einer Menge von Produzenten und Konsumenten, welche Clients für einen Dienst zur Benachrichtigung von Ereignissen darstellen. Alle Clients sind mit diesem gleichen Dienst verbunden und nutzen dessen Funktionalität zum Abonnieren oder zum Veröffentlichen von Benachrichtigungen. Aus der Sicht der Klienten ist der Benachrichtungs-Service zentralisiert und er berechnet für jede veröffentlichte Benachrichtigung die Untermenge aller Konsumenten, welche die Benachrichtigung abonniert haben und schickt die Benachrichtigung zu ihnen. Mögliche Implementierungen reichen von einer zentralisierten Implementierung mit Hilfe von Datenbanken (z.B. aktiven Datenbanken) bis hin zu einer völligen Verteilung des Routings von Benachrichtigungen [FMG02].
Ein weiteres Thema im Zusammenhang mit Event-basierten Systemen stellt "Complex Event Processing" (kurz: CEP) dar, welches hauptsächlich im Geschäftsprozessmanagement verwendet wird. Dieser Begriff steht für die Verarbeitung von vielen Ereignissen in allen Bereichen einer Organisation, die Identifikation der Wichtigsten dieser Events, die Analyse ihrer Auswirkungen und die Auswahl einer abschließenden Aktion in Echtzeit. Hierfür wird für jeden Prozess ein Zustand definiert: überschreitet dieser Zustand ein definiertes Maß, wird eine Handlung (beispielsweise Inkrementierung eines Zeitwertes), definiert als Ereignis, ausgeführt. Dies erfordert die respektive Überwachung, das Reporting, das Aufzeichnen und die Filterung dieser Events. Ein Ereignis kann also als Zustandsänderung in einem technischen oder wirtschaftlichen System betrachtet werden. Jede Zustandsinformation enthält eine "time stamp" um den Zeitpunkt des Auftritts zu bestimmen und eine Definition des Auftrittorts festzusetzen [Luc02]. "CEP" stützt sich auf eine Vielzahl von Techniken, darunter die Erkennung von Ereignismustern und Abhängigkeiten von Events, die Abstraktion eines Events und von Prozessen sowie die Modellierung von Ereignishierarchien [EN10].

2.5 Workflow Management & Applications

Das so genannte Workflow Management ist ein Lösungsansatz für ein altes Problem: Das Controlling, Monitoring, sowie die Optimierung und Unterstützung von Business Prozessen. Beim Workflow Management wird die Logik eines Business-Prozesses explizit dargestellt, was die Abbildung der Abläufe auf Computer ermöglicht. Es reicht nicht mehr einzelne Aufgaben mit Hilfe von Rechnern zu automatisieren, auch die logischen Aspekte von Business-Prozessen müssen von IT Systemen heutzutage abgedeckt werden: Die Arbeitsabläufe im Unternehmen müssen mit Hilfe von Rechnern verwaltbar sein. Viele Organisationen mit komplexen Business-Prozessen haben die Dringlichkeit von Konzepten, Techniken und Werkzeugen zur Unterstützung des Workflow Managements erkannt und somit wurde dieser Begriff offiziell etabliert [Aal98].
Es gab früher keine Methoden, welche zentral die Zusammenhänge und Abhängigkeiten verschiedener einzelner Aufgaben abbilden konnten, nur die Reihenfolge von zwei Tasks wurde beispielsweise bestimmt. Sobald sich der Business-Prozess ändert, kann sich auch die Abhängigkeit und die Reihenfolge dieser Aufgaben rasch ändern. Des Weiteren mussten die Zusammenhänge oft in mehreren Applikationen laufend verändert/neu implementiert werden und es war nicht möglich den gesamten Arbeitsfluss zu überprüfen. Software-Hersteller erkannten die Notwendigkeit von so genannten "Workflow Management Systemen" (kurz: WFMS). Diese

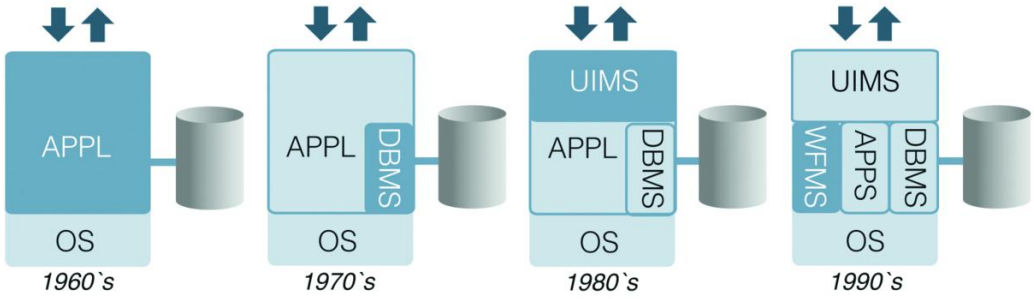

Abbildung 2.5: Workflow Management Systeme in historischer Betrachtung (nach [Aal98])

sind generische Software Tools, welche die Definition, Ausführung, Registrierung und Kontrolle von Arbeitsflüssen ermöglichen [Aal98].
Die Abbildung 2.5 zeigt die historische Entwicklung von Workflow Management Systemen über die letzten fünf Jahrzehnte. Die verwendete Architektur von Software-Systemen zu dieser Zeit wird jeweils illustriert: In den Sechzigern bestand Software nur aus einer reinen Applikationskomponente, in den Siebzigern kam noch ein Datenbankmanagementsystem (DBMS) hinzu. In den Achtzigern wurde das Konzept des "User Interface Management Systems" (UIMS) eingeführt, welches eine strikte Trennung von grafischer Benutzeroberfläche und der eigentlichen Programmlogik ermöglichte. In den Neunzigern wurde noch die "Workflow Management System Komponente" (WFMS) hinzugefügt um die Abläufe im Unternehmen klar und detailliert modellieren und implementieren zu können. Es ist klar ersichtlich, dass die letzt genannte Komponente einen eigenen Block bei der Software-Entwicklung darstellt, weil viele Organisationen bereits den Nutzen von fortschrittlichen Werkzeugen zur Unterstützung des Designs und der Ausführung von Business-Prozessen erkannt haben. Management Philosophien wie beispielsweise der Prozess der kontinuierlichen Verbesserung, welcher bereits im Kapitel Qualitätssicherung von Software vorgestellt wurde, haben das Bewusstsein für Business Prozesse innerhalb der Unternehmen erheblich gesteigert. Des Weiteren steigt die Anzahl der Arbeitsabläufe in den meisten Organisationen mehr und mehr an. Aus diesen Gründen ist die Komponente Workflow Management Systeme heutzutage in vielen Projekten nicht mehr weg zu denken [Aal98].
Der primäre Zweck eines WFMS ist die Unterstützung für die Definition, Ausführung, Registrierung und Kontrolle von Business Abläufen. Da die Prozesse im Vordergrund beim Workflow Management stehen, muss ein solides Framework zur Modellierung und Analyse von Workflow Prozessen gewählt werden. Eine sehr häufig genutzte Methode stellen hierbei die so genannten Petri-Netze dar. Sie wurden in den Sechzigern von Carl Adam Petri erfunden und werden heute noch zur Modellierung und Analyse von Applikationen in den Bereichen Protokolle, Hardware, flexible Produktionssysteme, eingebettete Systeme, Interaktion des Benutzers und noch viele weitere eingesetzt. In den letzten Jahrzehnten wurde das klassische Petri-Netz Modell noch um Farben, Zeit und eine Hierarchie erweitert [Aal98]. Ein Beispiel für ein Petri-Netz ist in Abbildung 2.6 ersichtlich. Einige Gründe für den Einsatz von Petri-Netzen für die Workflow

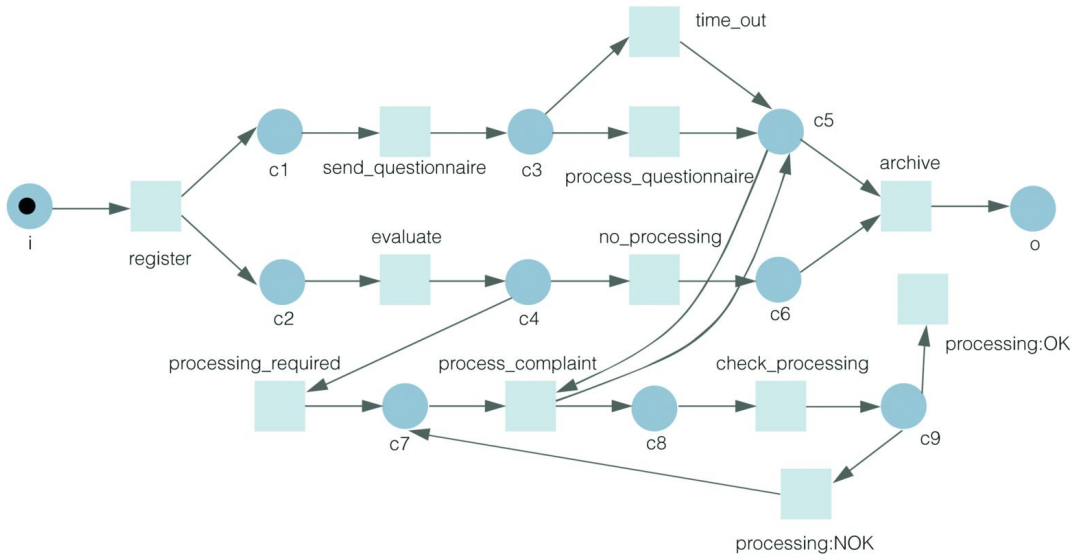

Abbildung 2.6: Petri-Netz Beispiel: Bearbeitung einer Beschwerde (nach [Aal98])

Modellierung werden nun angeführt [Aal98]:

- Formale Semantik: Ein Workflow besitzt eine einheitliche und präzise Definition.
- Graphische Repräsentation: Durch die graphische Modellierungssprache sind Petri-Netze sehr intuitiv, leicht zu erlernen und verbessern die Kommunikation mit dem Kunden.
- Ausdrucksstärke: Alles, was in Workflow Management Systemen definiert ist, lässt sich durch Petri-Netze abbilden. Des Weiteren lassen sich auch Meilensteine und implizierte Entscheidungen definieren.
- Eigenschaften: Die Eigenschaften von Petri-Netzen sind mathematisch belegt, es existiert eine große Menge gemeinsamen Wissens, welche in Form von Büchern und Artikeln zu finden sind.
- Analyse: Es existieren viele Analyse-Techniken um die Eigenschaften von Petri-Netzen zu verifizieren und Performance Merkmale (Antwortenzeiten, Wartezeiten, Auslastung, ...) zu berechnen.
- Unabhängigkeit: Petri-Netze stellen eine herstellerunabhängige Lösung zur Modellierung von Business Prozessen dar. Die Existenz wird durch Firmenübernahmen oder neuere Software nicht bedroht.

Eine Workflow Applikation hilft die oben erwähnten Business-Prozesse zu automatisieren. Sie können mit jeder Programmiersprache erstellt werden, jedoch gibt es Sprachen, die speziell zur

Unterstützung der Implementierung von Workflows konzipiert wurden. Diese besitzen meist auch eine Form der graphischen Notation (wie beispielsweise die Business Processs Modelling Notation (BPMN)), was einige Vorteile gegenüber rein text- oder xml-basierten Lösungen hat. Die Diagramme können auch von weniger technisch versierten Benutzern verstanden und interpretiert werden, was klare Vorteile in der Kommunikation mit den Kunden bringt. Normalerweise wird das Diagramm wiederum in XML Sprache transformiert. Des Weiteren gibt es noch eine andere Art Workflow Application zu entwickeln: Man verwendet Programmiersprachen in Verbindung mit Bibliotheken und Schnittstellen, welche eine Abstraktion der Aufgabenkoordination ermöglichen, wie beispielsweise die Workflow OSID oder die Windows Workflow Foundation. Diese Bibliotheken sollte man verwenden, wenn die Diagrammnotationen nicht ausreichend mächtig sind um die volle Funktionalität der Software abbilden zu können. Die oben bereits genannten Petri-Netze stellen eine formale mathematische Grundlage zur Abbildung von Workflows dar.

2.6 Mashup Applikationen

Applikationen, welche durch die geschickte Kombination bestehender Inhalte erstellt werden, sind so genannte Mashups. Ein Beispiel hierfür ist die Webpage http://www.fastfood-maps.com. Die geographischen Positionen der Filialen bekannter Fastfood-Ketten werden auf einer Karte dargestellt. Die bestehende Google-Maps Lösung wurde hier mit den Adressen der Fastfood-Betreiber verknüpft. Der Erfolg dieses speziellen Applikationstyps baut auf den Technologien der "Service-orientierten Architekturen" (Services werden für neue Businessprozesse kombiniert) und "Web 2.0" (Erweiterung des Inhalts einer Website durch den Besucher) auf. Die Entwicklung von Mashups ist heute nicht mehr nur den versierten Entwicklern vorbehalten, Editoren erlauben die einfache Erstellung, sodass täglich mehrere neue Mashups im Web vorgestellt werden. Enterprise Mashups erlauben Defizite von Service-orientierten Architekturen im Bereich der Benutzer-Integration zu verringern [Ger07].
Mashups lassen sich in Web- und Enterprise Mashups einteilen. Erstere sind, wie der Name schon vermuten lässt, im Internet zu finden und werden eher von technisch interessierten Leuten verwendet. Zweitere werden für den Einsatz im Unternehmen entwickelt und sollen den Mitarbeitern ihre Arbeit erleichtern [Ger07].
Es gibt drei verschiedene Mashup-Typen, welche nachfolgend erläutert werden [Ger07]:

- Presentation Mashup: Der Inhalt von mehreren Quellen wird in einer gemeinsamen graphischen Oberfläche präsentiert (Beispiele: iGoogle und MyYahoo)

- Data Mashup: Wie der Name bereits verrät, handelt es sich um eine Kombination unterschiedlicher Daten von verschiedenen Quellen (Beispiel: auf GoogleMaps basierende Applikationen)

- Logic Mashup: Es werden zwei oder mehrere Applikationen verbunden oder bestimmte Tasks automatisiert. Meistens verbirgt eine einheitliche graphische Oberfläche die Komplexität hinter solchen Mashups. Die Kommunikation und Integration der Anwendungen kann durch eine Adapter- oder Kopplungskomponente realisiert werden. Beim Einsatz im

Unternehmen gibt es starke Parallelen zu klassischen Workflow-Applikationen. (Beispiel: www.checkfelix.com)

Die wichtigsten Eigenschaften von Mashups lassen sich wie folgt zusammenfassen [Ger07]:

- Zusammenarbeit vieler Benutzer: Mehrere Anwender können an der Erstellung einer Web-Anwendung beteiligt sein ohne auf die Informationstechniker angewiesen zu sein. Änderungen oder Verbesserungen können von jedermann durchgeführt werden.
- Entwicklung endet nicht: Enterprise-Mashups folgen nicht den Prinzipien klassischer Software-Entwicklung (Anforderungsanalyse, Planung, Implementierung des Systems und anschließendes Testen). Die Applikation wird innerhalb kürzester Zeit mit Komponenten zusammengebaut, der Fokus liegt auf der agilen Vorgangsweise, welche schnelle Resultate und die Funktionen der Software in den Vordergrund und eine penible Einhaltung von Spezifikation und Dokumentation in den Hintergrund stellt.
- End User Innovation: Dieser von Von Hippel geprägte Begriff basiert auf dem Konzept, dass die Anwender der Applikation an der Entwicklung beteiligt sind. Der Hersteller liefert den Kunden Innovations-Toolkits zur eigenen Entwicklung, das heißt dieser kann die Applikation in einer Art Vorschau betrachten und nach seinen Vorstellungen gestalten und anpassen. Dieses Prinzip lässt sich in den Enterprise-Mashup Bereich übertragen. Das User Innovation Toolkit stellt in diesem Fall der Mashup-Editor dar.

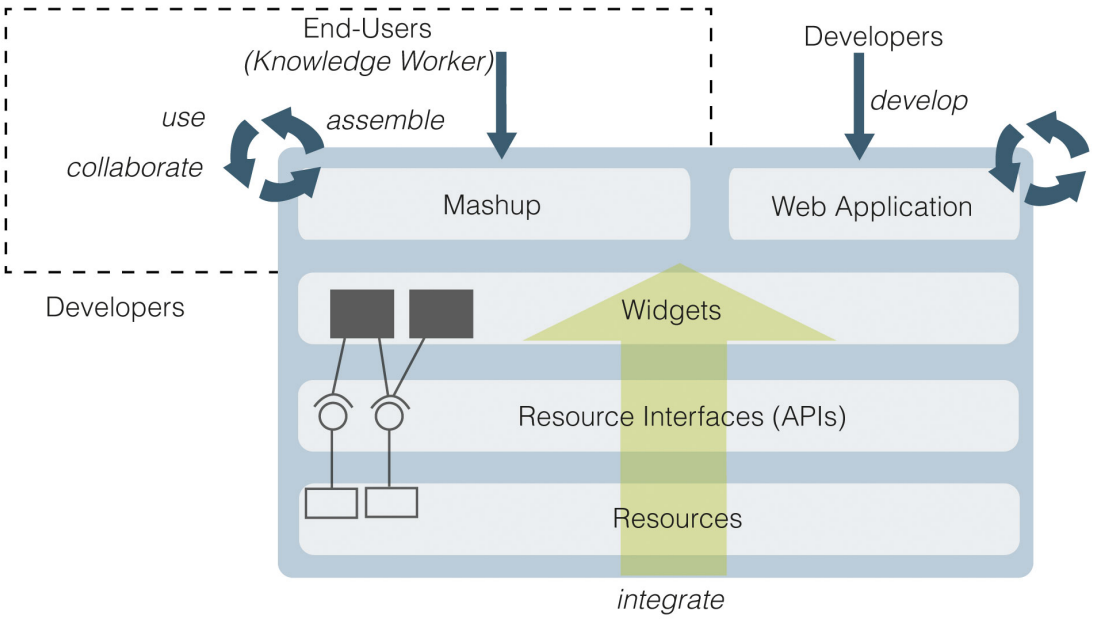

Abbildung 2.7: Aufbau eines Mashup-Stacks (nach [Ger07])

Die Architektur des Mashups-Stacks (der technische Aufbau eines Mashups) (siehe Abbildung 2.7)und eng verbundene Standards werden jetzt vorgestellt [Ger07].
Auf unterstem Level des Stacks befinden sich die **Web-Ressourcen**, welche meist im REST (Representational State Transfer) Standard oder als RSS-Feed (XML-Datei) verwendet werden. Mashups bedienen sich sowohl der Service-orientierten als auch der web-orientierten Architektur.
Eine Ebene über den Ressourcen befindet sich das **Application Programming Interface (kurz: API)**, welches die Nutzung der darunterliegenden Ressourcen mit Hilfe einer Dokumentation ermöglicht. Ohne das API würden Änderungen in der Implementierung zu Änderungen im Mashup führen, deshalb müssen standardisierte Schnittstellen verwendet werden.
So genannte **Widgets** greifen über die APIs auf die Ressourcen zu, erlauben Veränderungen, und Kombinationen dieser Inhalte gewährleisten die Interaktion mit dem Anwender. Ein Widget ist ein Stück Programm-Code, der ohne Widget-Engine nicht ausgeführt werden kann. Diese Engine stellt die Grundfunktionalität für das Widget zur Verfügung. Es lassen sich drei verschiedene Widget-Typen unterscheiden:
Ressourcenbezogene erledigen die Kommunikation mit den Inhalten.
Programmablaufbezogene sorgen für die Steuerung des Ablaufs der Mashup-Applikation.
Präsentationsbezogene bestimmen das Design (Graphische Repräsentation) oder den Output (Datenpräsentation) der Anwendung.
Ein Widget kann allerdings auch alle drei vorher genannten Bereiche abdecken. Es existiert kein einheitlicher Standard für Widgets, wichtige Vertreter sind W3C Widgets, OpenAjax Metadata und das Google Gadget "Application Programming Interface". Widgets sind bei Mashup-Editoren als einzelne Objekte repräsentiert.
Die Kombination von verschiedenen Widgets mit Hilfe eines Mashup-Editors führen zu einem fertigen **Mashup**. Hier gibt es zwei verschiedene Konzepte: Piping bezieht sich auf die Verknüpfung von Ressourcen und Anwendungen, eine Assoziation verschiedener Widgets untereinander bezeichnet man hingegen als Wiring. Das fertige Mashup kann als Web-Anwendung verwendet oder als Web-Service anderen Applikationen zur Verfügung gestellt werden.
Ein Enterprise-Mashup bildet die gesamte IT innerhalb des Unternehmens (und darüber hinaus) ab und unterscheidet sich hauptsächlich durch die nachfolgend angeführten Komponenten [Ger07]:

- Konnektor: baut Verbindungen zu Ressourcen auf, gibt dem Mashup Zugriff darauf und enthält die Informationen darüber wie die Ressourcen verwendet werden müssen. Oft werden die Daten im REST-Stil oder als Feed zur Verfügung gestellt, sodass nicht nur das eigene Mashup sondern auch Web-Applikationen darauf zugreifen können.

- Mashup-Editor: Der Editor ermöglicht die Erstellung von Mashups durch die Kombination von Widgets und den Ressourcen. Mit Hilfe einer graphischen Oberfläche werden die Widgets miteinander verknüpft und Abhängigkeiten festgelegt. Die meisten Editoren sind sehr benutzerfreundlich und benötigen keine speziellen Programmierkenntnisse.

- Katalog: Diese Komponente verwaltet Ressourcen, Widgets und fertige Mashups und fördert die Wiederverwendung von Teilen der implementierten Software.

2.7 Monitoring & Reporting für verteilte Systeme

Das Kapitel behandelt ein für diese Arbeit zentrales Thema: Das Monitoring im Bereich der Softwarequalitätssicherung. Auf ersteren Begriff wird in den nachfolgenden Absätzen eingegangen. **Monitoring** ist ein Oberbegriff für das wiederholte und geplante Überwachen, Beobachten und Dokumentieren von Software mit der Hilfe von verschiedenen Techniken und/oder Tools. Durch den Vergleich der Resultate jeder Iteration kann man Schlussfolgerungen über das beobachtete System ziehen. Kommt es zu unerwarteten Ergebnissen, kann man leitend eingreifen, es ist somit eine Form der Qualitätskontrolle.
Legt man den Fokus auf das Testen für verteilte Systeme, lassen sich drei verschiedene Schritte des Monitoring feststellen, die nun beschrieben werden [MS95]:

1. **Erzeugung der Informationen über den Zustand des Systems:** Typische Abläufe sind beispielsweise der Empfang einer Message, ein Methodenaufruf und viele mehr. Während der Prozess läuft, wird der Bericht durch diese Abläufe erstellt und man kann die Ausführung des Prozesses genau betrachten und die Vorgangsweise erschließen.

2. **Verarbeitung der erzeugten Informationen:** Die einzelnen Reports aus dem ersten Schritt müssen zu einem einzigen Bericht verknüpft werden.

3. **Visualisierung der verarbeiteten Informationen:** Eine Präsentation der erzeugten Informationen muss erstellt werden, diese soll bei der durchgeführten Tätigkeit helfen.

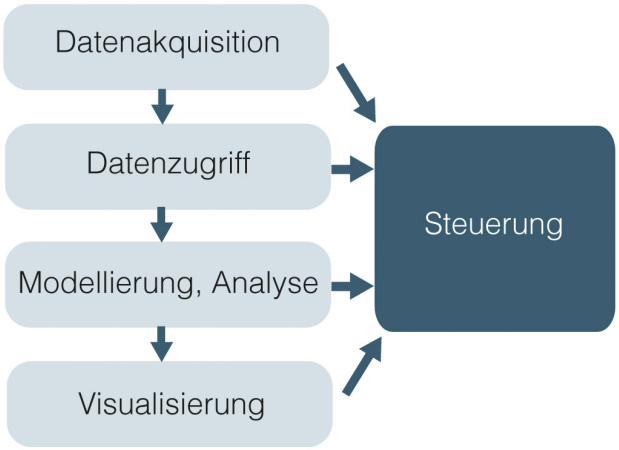

Abbildung 2.8: Monitoring-Architektur (nach [Kur00])

Die Abbildung 2.8 beschreibt eine Monitoring-Architektur: Auf unterster Ebene liegt die Datenakquisitionsschicht: Die benötigten Events für das Monitoring werden im System eingetragen. Trifft der Prozess auf einen eingetragenen Kontrollpunkt, wird das Ereignis ausgelöst und ein Bericht mit zusätzlichen Metadaten (Zeit, Datum, Ort, Status, ...) erzeugt (sofern alle Bedingungen, die im Event gesetzt wurden, auch zutreffen). Das nächste Level bildet der so genannte

Datenzugriff. Die Daten aus der Datenakquisition werden gespeichert. Die gesamte Struktur der Daten wird in der nächsthöheren Schicht gespeichert um die klare Trennung von Messung und Auswertung zu erreichen. Es wird auch festgelegt, wo die Daten gespeichert werden, die Leistung der Analyse wird von diesen Entscheidungen stark beeinflusst. Die nächste Ebene stellt die Modellierungs- und Analyseschicht dar, in welcher der globale Zustand der Software ermittelt wird. Einzelne Event Reports werden von der Datenzugriffsschicht in ein Modell integriert, Attribute dieser Events können zur Vermeidung inkonsistenter Systemzustände überprüft werden. Es müssen nicht alle Ereignisse bearbeitet werden, wenn sinnvolle Filter gesetzt werden. Die höchste Ebene bildet die Visualisierung der Daten: Es gibt verschiedenste Repräsentationsformen, von der grafischen bis zur textuellen Darstellung. Die Steuerung aller Schichten wird von der Ereigniskonfiguration übernommen, welche die einzelnen Ebenen durch Änderungen aufeinander abstimmen kann und muss (beispielsweise Sensoren- oder Strategieanpassung) [Kur00].

Die Vorteile des Monitoring lassen sich wie folgt zusammenfassen [Sch95]:

- Evaluierung und Verbesserung der Leistung des Systems
- Permanente Überwachung des Systemzustands
- Verbesserung von Sicherheitsaspekten
- Überprüfung auf korrekte Funktionalität
- Steigerung der Verlässlichkeit

Die Ereignisse lassen sich in Hardware-Level, Process-Level und Anwendungsabhängige Events unterteilen. Erstere werden auf Hardwareebene ausgelöst, beispielsweise durch Input/Output Fehler. Process-Level Ereignisse werden durch Vorgänge im Prozess ausgelöst, lassen sich aber nur von außen beobachten. Die Identifikation von Kommunikations- und anderen schweren Softwarefehlern wird durch diese Events ermöglicht. Beispiele für anwendungsabhängige Events sind die Änderung des Inhalts einer Variable oder ein Method-Call.
Es lassen sich drei mögliche Konzepte der Implementierung beschreiben [MS95]:
Hardware-Monitoring: Die Hardware übernimmt die Überwachung der Events durch die Beobachtung des Steuer-, Adress-, und Datenbusses. Diese Art des Monitoring stört das System, welches beobachtet wird, überhaupt nicht oder nur gering (non-intrusive). Allerdings führen Änderungen auf Softwareebene auch immer zu einer kostspieligen Neuentwicklung der Hardware. Des Weiteren sind die Systeme wenig flexibel, schwer zu portieren und die Datenaufbereitung gestaltet sich aufwendig.
Software-Monitoring: Das Monitoring geschieht nur auf Programmebene durch Erweiterung der bestehenden Software mit Monitoring-Methoden. Ein Nachteil dieser Methode ist, dass Teile der Ressourcen, die normalerweise dem System zur Verfügung stehen, nun vom Monitoring verwendet werden. Im Vergleich zum Hardware-Monitoring sei die deutlich bessere Portabilität, Flexibilität und Kosteneffizienz hervorzuheben. Außerdem ist die Datenaufbereitung wesentlich einfacher, weil der Abstraktionsgrad wesentlich geringer als beim Hardware-Monitoring ausfällt.
Hybrid-Monitoring: Die Vorteile aus dem Hardware- und Softwaremonitoring werden miteinander kombiniert. Es werden spezielle Monitoring-Aufrufe in den Programmcode eingefügt, die

Ausführung erledigt aber eigene Hardware. Die Portabilität sowie die Flexibilität bleiben erhalten, des Weiteren kommt es zu keiner erhöhten Ressourcennutzung durch das Monitoring und durch die Auswertung der Daten. Eine Software visualisiert schließlich die analysierten Dateien.
Unter dem Begriff Reporting versteht man schließlich die Aufarbeitung und anschließende Präsentation der aus dem Monitoring-Verfahren gewonnenen Daten. Dies wird oft mit graphischen Hilfsmitteln wie Diagrammen oder Bildern bewerkstelligt. Große Datenmengen aus verschiedenen Bereichen eines Unternehmens werden mit der Rechenleistung von Computern zu einheitlichen Berichten verarbeitet, dieser Vorgang ist im Allgemeinen als Begriff des Enterprise Reporting bekannt ist. Diese Berichterstattung ist ein essentieller Bestandteil der so genannten Business Intelligence, welche mit Hilfe einer komplexen Datenbank (Data Warehouse) und einigen Reporting-Hilfsmitteln nützliche Informationen innerhalb eines Firmennetzwerks bereitstellt.

2.8 Software-Tests und Dokumentation

Zunächst nimmt dieses Kapitel Bezug auf die unterschiedlichen Testtechniken und die verschiedenen Testlevels, welche nun vorgestellt werden. Ein Softwaretest überprüft die Software auf korrekte Funktionalität. Es lassen sich drei verschiedene Testtechniken unterscheiden, auf die nun näher eingegangen wird [Tha02]:
Blackbox-Testing: Die Software wird getestet ohne den Quellcode berücksichtigt zu haben. Nur die Ein- und Ausgaben der zu testenden Implementierung stehen im Fokus. Es wird versucht mit einer kleinen Menge an Eingabewerten einen Großteil der möglichen Tests abzudecken. Hierfür wurden Techniken wie beispielsweise eine Einteilung der Input-Daten in Äquivalenzklassen (mit oder ohne der Festsetzung von Grenzwerten) oder so genannte Übergangstests, welche die möglichen Zustandstransitionen einer Software testen, entwickelt.
Whitebox-Testing: Bei diesen Tests ist der Quellcode bekannt, der Ursprung eines Fehlers kann somit eindeutig festgestellt werden. Auch hierfür wurden eigene Techniken und Technologien entwickelt. Mittels Statement Coverage ist beispielsweise feststellbar, wie viel Prozent des implementierten Codes bereits getestet wurden. Die so genannte Decision Coverage überprüft, wie der Name schon mitteilt, ob alle möglichen Entscheidungen (typischerweise if- oder switch-Anweisungen) getestet wurden. Die Path Coverage Testmethode überprüft alle möglichen Wege einer Implementierung (geschachtelte if-Anweisungen führen beispielsweise zu unterschiedlichen Pfaden, welche die Software durchlaufen kann).
Experience-Based Testing: Es werden keine speziellen Testverfahren verwendet, die Tests bauen ausschließlich auf dem Wissen und der Erfahrung des Testleiters auf.
Grey-Box Testing: Wird als Technik aus der "Test-Driven Development" Methode, welche im Kapitel Testgetriebene Entwicklung ausführlich vorgestellt wird, verstanden. Die Vorzüge aus dem Black- & Whitebox-Testing kommen dabei zu tragen. Die Tests lassen sich in sieben verschiedene Kategorien einordnen, welche nachfolgend detailliert betrachtet werden [Tha02]:

- Eigene Tests der Entwickler: Der/die Entwickler testen die Software auf korrekte Funktionalität ohne die Tests zu dokumentieren. Jedoch werden häufig Coverage-Techniken, welche

bereits unter dem Punkt White-Box-Testing vorgestellt wurden, für die Überprüfungen verwendet.

- Modul-Tests (Unit-Tests): Wie der Name schon andeutet, wird nur ein einziges Modul getestet, die Unit-Tests werden während der Code-Erstellung durchgeführt. Ein bestimmter Eingabewert muss mit dem erwarteten Ausgabewert übereinstimmen, sonst schlägt der Test fehl.

- Komponententests: Mehrere voneinander abhängige Module werden getestet. Verschiedene Eingabewerte müssen zu zuvor spezifizierten, erwarteten Resultaten führen. Sowohl die Unit- als auch die Komponententests helfen bei der frühen Aufdeckung von Fehlern während der Implementierung.

- Integrationstests: Die Integrationstests überprüfen das Zusammenspiel mehrerer Komponenten.

- System-Tests: Das System wird verifiziert, die Gesamtheit aller Komponenten und deren Interaktion werden auf korrekte Funktionalität geprüft.

- Akzeptanztests: Der Kunde überprüft ob die von ihm gesetzten Erwartungen an das gewünschte Produkt durch die Implementierung erfüllt werden können.

Bei der Automation werden Tests laufend ausgeführt, sollte sich bei einer Änderung des erstellten Codes Fehler einschleichen, werden diese ebenfalls zuverlässig entdeckt. Wird ein Test manuell mehrere Male im Laufe des Projekts ausgeführt, spricht man von Regressionstest. Die Testautomatisierung bringt den Vorteil der Kostenreduktion [Tha02].
Die Planung und Dokumentation von Tests ist ebenfalls essentiell bei der Betrachtung der Test-Domäne. Die IEEE Association hat einen Standard zur Software- und Systemtestdokumentation definiert, welcher in der Abbildung 2.9 illustriert wird (die Testdokumentation in einem Projekt wird veranschaulicht). Wichtige Elemente dieser Illustration werden nachfolgend erläutert [IEE08]:

- Anomaliebericht: Alle möglichen Events, die während eines Testprozesses auftreten können, werden in diesem Bericht inklusive einer zutreffenden Beschreibung, dem Datum des Eintretens, dem erwarteten Verhalten und dem Status innerhalb des Projekts dokumentiert.

- Der Teststatusbericht dokumentiert die Ergebnisse von Tests zur späteren Auswertung. Es werden verschiedene Metriken definiert um den gesamten Teststatus quantitativ bewerten zu können.

- Der Abschlusstestbericht stellt das logische Ende der Testkette dar, es werden alle zuvor erstellten Berichte zu einem einzigen Report zusammengefasst. Hier werden ebenfalls die Daten ausgewertet und daraus Empfehlungen für das Vorgehen in der Zukunft abgeleitet. Alle Testergebnisse werden hier zusammengefasst, das Vorgehen festgehalten und Pläne für die Zukunft dokumentiert.

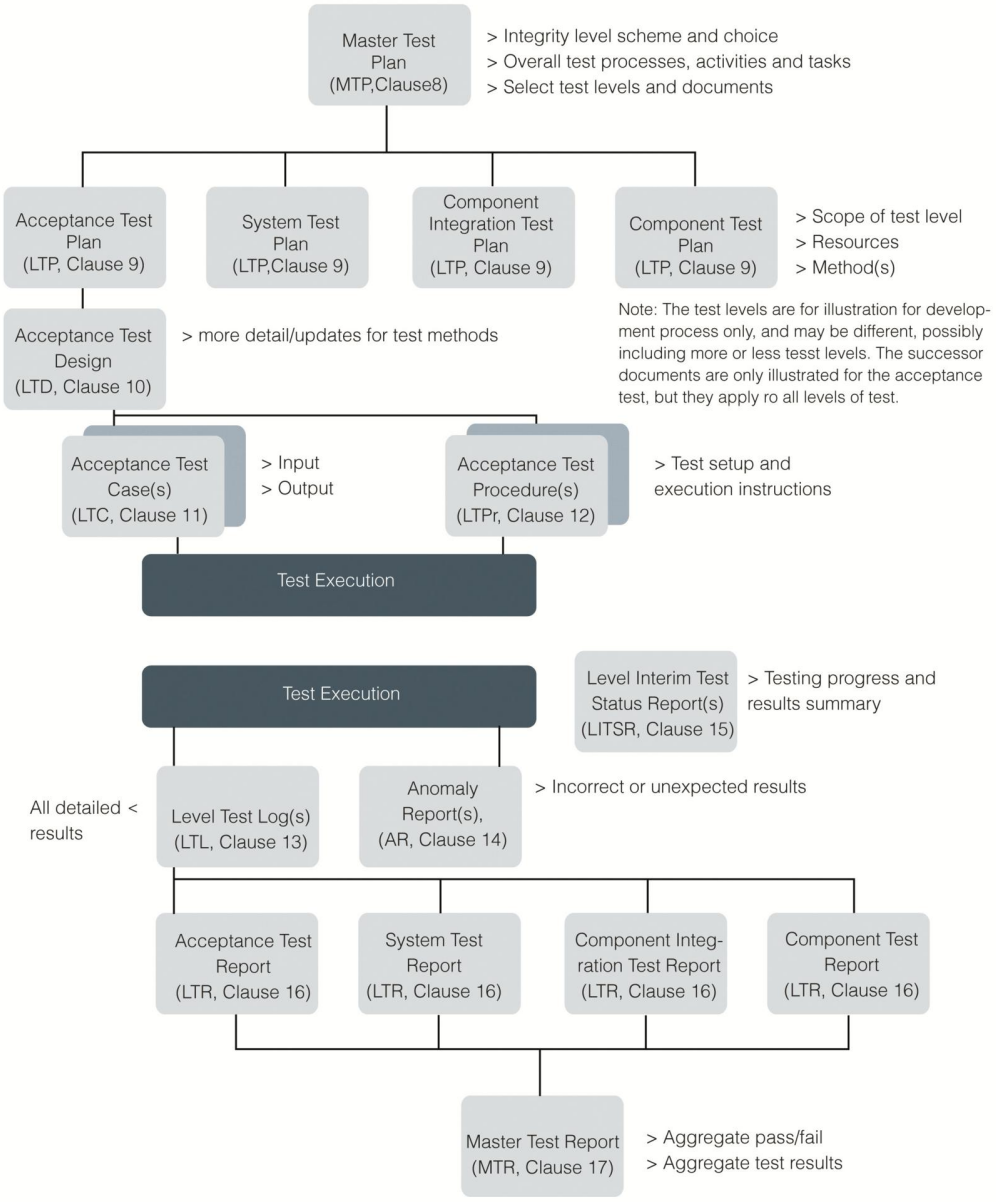

Abbildung 2.9: Testdokumentation innerhalb eines Projekts(nach [IEE08])

Nicht nur die erwarteten Ergebnisse werden dokumentiert, auch die Durchführung der einzelnen Tests wird protokolliert. Auch hier sind unterschiedliche Test-Levels definiert: Der Master Test Plan (MTP) ist für die Koordination der gesamten Planung und des Managements aller Tests auf allen Ebenen zuständig. Die geschätzte benötigte Zeit und der benötigte Aufwand wird für alle Testaktivitäten festgelegt. Darüber hinaus kommt es zu einer Definition der Abhängigkeiten zwischen den einzelnen Testschritten, der verantwortlichen Personen, der Metriken, der eingeset-

zten Hilfssoftware und zu einer Risikoanalyse. Der Level Test Plan (LTP) wird hauptsächlich für Komponenten-, Integrations- und Akzeptanztests verwendet und spezifiziert die Vorgangsweise, benötigte Ressourcen und den Zeitplan für jeden einzelnen Test. Des Weiteren kommt es zu einer Festlegung der Verantwortlichkeiten inklusive der Analyse von damit verbundener Risiken. Das Level Test Design (LTD) spezifiziert dann Details für jede Aktivität des Level Test Plans. Der Level Test Case (LTC) stellt schließlich die unterste Ebene dar: Die bisher ungeklärten Daten werden spezifiziert: Die Ein- und Ausgaben werden für jeden Test protokolliert, die Testzielsetzung dokumentiert und die Kriterien für einen positiven Testabschluss festgelegt.

2.9 Bestehende Software-Entwicklungsprozesse

Dieses Kapitel stellt die bekanntesten Software-Entwicklungsprozesse vor, die jedem Entwicklungsteam helfen in allen Phasen des Projekts den Überblick zu behalten und große Vorhaben zeitgerecht abschließen zu können. Sie unterscheiden sich in vielen Punkten und nicht jedes Vorgehensmodell ist für jedes Projekt geeignet. Um die Vorgangsweise jedes Modells zu verstehen, werden diese nun, mit speziellem Fokus auf den Testvorgang, beschrieben und die praktische Anwendung einiger Modelle jeweils anhand eines Beispiels zum besseren Verständnis demonstriert.

Das Wasserfallmodell

Wie aus Abbildung 2.10 klar ersichtlich, ist der einfache Wasserfallprozess ein sequentielles, aber nicht iteratives Modell [Roy70]. Das Projekt wird hier in mehrere Phasen unterteilt, die nun näher beschrieben werden.

- System & Software Requirements: Die Anforderungen an das System und die Software werden analysiert und dokumentiert. Hier können auch die zukünftigen Benutzer befragt werden.
- Analyse: Ein Pflichtenheft wird erstellt und das Modell wird definiert (eventuell mit graphischer Repräsentation wie beispielsweise mittels UML-Anwendungsfall- und Aktivitätsdiagramm). Auch hier können die zukünftigen Benutzer in den Prozess mit einbezogen werden.
- Design des Programmes: Definition der verwendeten Programmiersprache/n, der Interfaces, der Klassen, der Methoden usw... UML-Klassendiagramme helfen bei der Erstellung des Designs und ermöglichen mittlerweile für viele Sprachen automatisierte Codegenerierung.
- Coding-Phase: Die Software wird von den Programmierern mit Hilfe der bestehenden Definition aus der Designphase implementiert.
- Test-Phase: Die produzierte Software wird getestet und eventuelle Fehler festgestellt.
- Operations: Das erstellte Produkt befindet sich im operativen Einsatz. Dies ist die letzte Phase des einfachen Wasserfallprozesses.

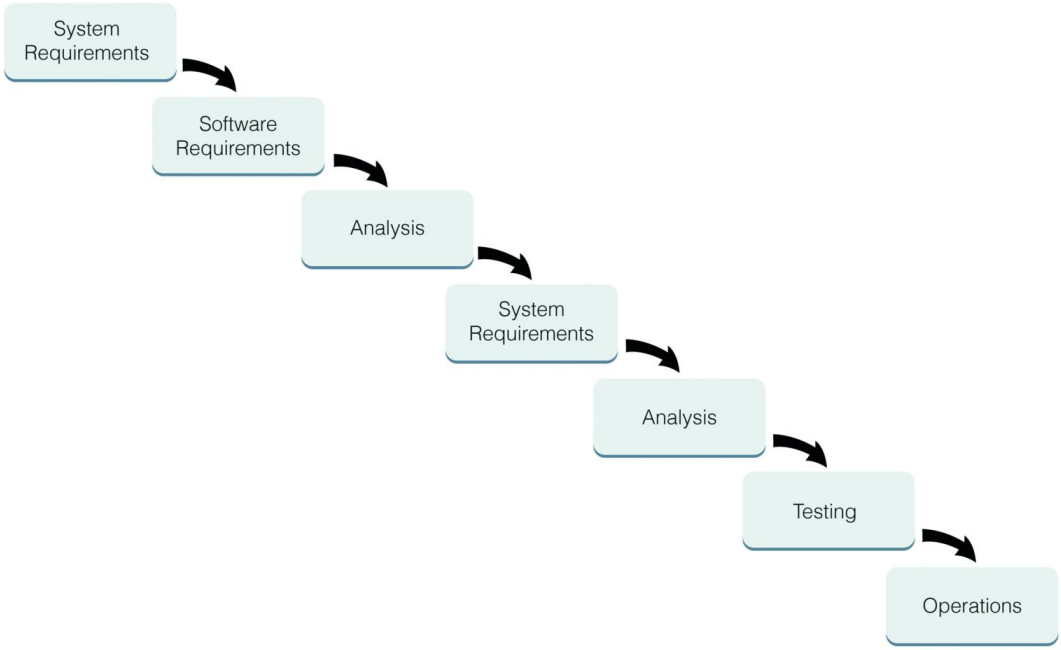

Abbildung 2.10: Einfaches Wasserfallmodell (nach [Roy70])

Jeder dieser sechs Arbeitsschritte kann erst dann beendet werden, wenn alle dafür bestimmten Dokumente erstellt wurden. Diese Vorgangsweise dient der Reduktion der Risiken [GKZ04]. Das einfache Wasserfallmodell erlaubt keinen Rückschritt auf die vorherige Phase, dies kann vor allem bei der Testphase Probleme bereiten: Ist die Software fehlerhaft, muss der Prozess wieder bei Schritt 1 (System & Software Requirements) beginnen, dies ist nicht kosteneffizient [Roy70].
Beim erweiterten Wasserfallmodell stellt sich die Vorgehensweise anders dar (siehe Abbildung 2.11). Es definiert den Rückschritt auf die jeweils vorherige Phase des Projekts, doch auch hier kann jeder Arbeitsschritt erst abgeschlossen werden, wenn alle benötigten Dokumente fertig gestellt wurden. Dies bringt den Vorteil, dass Fehler in vorigen Phasen leichter ausgebessert werden können [GKZ04].
Bei einer strikten Trennung der einzelnen Schritte lässt sich der Wasserfallprozess sehr gut anwenden, allerdings müssen vor Beginn des Projektes die Risiken bereits ausgeschlossen werden. Es ist vor allem für kleine Projektteams geeignet, weil sich bei dieser Vorgehensweise alle Mitarbeiter gleichzeitig auf die Vervollständigung eines Arbeitsschrittes konzentrieren können. Eine Arbeitsgruppe mit vielen Mitgliedern sollte sich jedoch auf vollständig iterative Modelle stützen [GKZ04].
Sowohl das einfache als auch das erweiterte Wasserfallmodell sind für verteilte Systeme nicht geeignet, weil eine sequentielle Erstellung der einzelnen Services der Software aus Zeit- und kostentechnischen Gründen nicht möglich ist. Des Weiteren gestaltet sich die Automatisierung

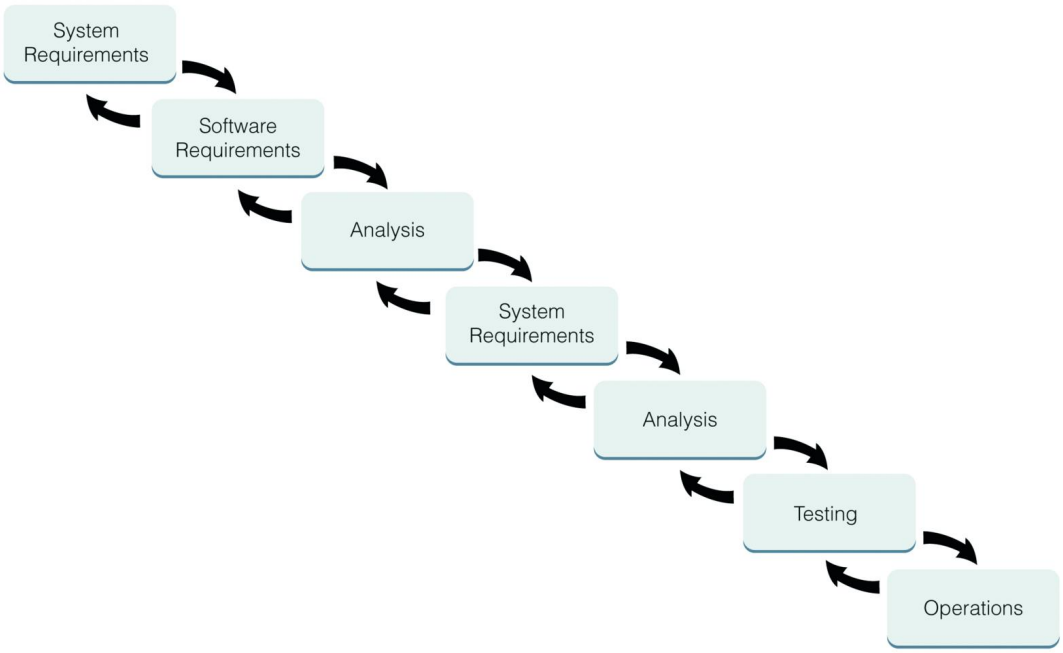

Abbildung 2.11: Erweitertes Wasserfallmodell (nach [Roy70])

der Integrationstests (wer erstellt diese Tests?) sowie die nachträgliche Anpassung der einzelnen Software-Services als schwierig: Die Kommunikation und Kollaboration der Projektleiter für jedes Services und der gesamten Umgebung stellt eine beinahe unlösbare Herausforderung dar.
Abschließend wird nun das Testen der Software im Wasserfallprozess näher erläutert. Während der Coding-Phase werden die einzeln erstellten Module bereits isoliert mit so genannten Unit-Tests auf Fehler überprüft. In der Test-Phase werden mit Hilfe von Integrationstests die voneinander abhängigen Komponenten getestet, im finalen Systemtest sollen letzte Fehler gefunden werden. Die Erstellung des Testplans mit seinen Testfällen obliegt dem leitenden Tester oder bei sehr kleinen Teams dem Projektleiter und ist durch den Wasserfallprozess nicht näher spezifiziert. Erfüllt die Software allerdings nicht die spezifizierten Systemanforderungen, müssen das Design überarbeitet und in weiterer Folge Teile der Software verändert beziehungsweise vollständig neu implementiert werden um die geforderte Funktionalität zu erreichen [Roy70].
Ein praktisches Beispiel zur Anwendung des Wasserfallprozesses kann durch folgende Situation beschrieben werden: Ein kleines Team bestehend aus einem Projektleiter, drei Programmierern und einem Testleiter, erhält die Aufgabe für eine Firma eine spezifische Software für den Finanzbereich zu erstellen. Der Projektmanager befragt im ersten Schritt die zukünftigen Benutzer des Systems über die gewünschte Funktionalität, dokumentiert und hält die Anforderungen an das System und die Software fest. Im nächsten Schritt wird aufbauend auf den Requirements ein Pflichtenheft erstellt, welches die weitere Vorgangsweise (Zeitplan und Meilensteine)

definiert. Es wird ein allgemeines Modell der Software mit Hilfe von UML-Anwendungsfall-, und Ablaufdiagrammen erstellt. In Absprache mit den Programmierern wird im nächsten Schritt die verwendete Programmiersprache festgelegt und die Klassenstrukturen inkl. Schnittstellen mittels UML-Klassendiagramm festgelegt. Die Implementierung beginnt, fertige Codefragmente werden mit Hilfe von Modultests bereits auf korrekte Funktionalität überprüft. Ist die Entwicklung abgeschlossen übernimmt der Testleiter die Analyse der Software und verifiziert mittels Integrations- und Systemtests das von den Anforderungen spezifizierte Verhalten. Kommt es dabei zu schweren Fehlern beziehungsweise Abweichungen wird der Tester diese Anomalien kommunizieren und leitet Veränderungen und eine erneute Überprüfung des Systems ein. Verläuft die Testphase allerdings fehlerfrei, wird die Software an die Kunden ausgeliefert und der Wasserfallprozess wird beendet.

Das V-Modell

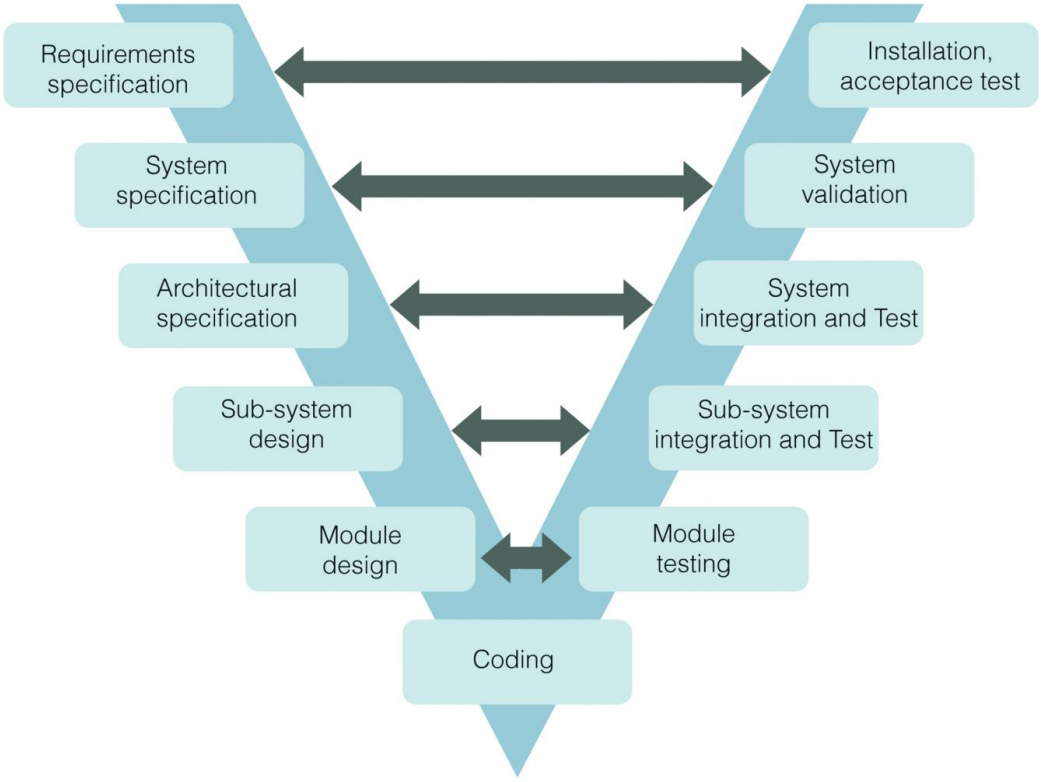

Abbildung 2.12: V-Model (nach [Boe79])

Das V-Modell stützt sich, wie auch das Wasserfallmodell, auf eine sequentielle Vorgehensweise. Das Projekt wird allerdings in zwei Phasen geteilt: Zu jedem bestehenden Arbeitsschritt in der ersten Phase, welche sich an den Aktivitäten des Wasserfallmodells orientiert, wird

jeweils ein Test erstellt, der die Funktionalität jedes Schrittes garantieren soll und die zweite Phase darstellt. Die Abbildung 2.12 zeigt die grafische Repräsentation des Modells [Boe79].
Eine Erweiterung des V-Modells stellt das so genannte V-Modell XT dar. Es versucht bei jedem Schritt den rein dokumentbasierten Ansatz mit dem modellbasierten Ansatz moderner Softwaresysteme zu verbinden [Bar05]. XT steht für Extreme Tailoring und drückt die Anpassungsfähigkeit des Modelles aus. Auch der Auftraggeber wird in die Erstellung mit eingebunden. Die starre Vorgehensweise des alten V-Modells wurde durch eine individuelle Zusammenstellung (mit Hilfe von Vorgehensbausteinen) ersetzt, welche eine individuelle Anpassung an das Projekt ermöglicht. Des Weiteren existieren keinerlei zeitliche Vorgaben für die Bausteine, die Agilität des Prozesses steht im Mittelpunkt [Nie05]. Die Vorgehensbausteine lassen sich in folgende Kategorien einteilen:

- Projektmanagement: Das Projekt wird geplant, die Auftraggeber in diesen Prozess eingebunden.
- Qualitätssicherung: Die Software wird auf korrekte Funktionalität überprüft.
- Konfigurationsmanagement: Alle geplanten Produkte sind in einer Produktbibliothek mit ihrem jeweiligen Bearbeitungsstatus aufgelistet.
- Problem- und Änderungsmanagement: Die Änderungen an fast fertigen Produkten werden dokumentiert.

Die Abbildung 2.12 definiert zu jeder Phase auch entsprechende Tests auf welche nun näher eingegangen wird [Boe79]:

1. **Anforderungsspezifikation, Installations- und Akzeptanztests (Requirements Specification, Installation & Acceptance Test):** Wie im Wasserfallmodell werden die Anforderungen an die Software in Phase I spezifiziert, durch Installations- und Akzeptanztests werden diese dann in Phase II verifiziert.
2. **Systemspezifikation, Validierung des Systems (System Specification, System Validation):** Auch hier lassen sich Parallelen zum Wasserfallmodell ziehen, in Phase I wird das zu entwickelnde System mit UML-Diagrammen als Hilfsmittel definiert und in der zweiten Phase wird die erstellte Software mit der Spezifikation validiert.
3. **Design der Architektur, Integrationstests (Architectural Design, System Integration and Test):** Hier wird in Phase I die Architektur der Software mit UML-Klassendiagrammen festgelegt und in Phase II diese mit Integrationstests auf korrekte Implementierung überprüft.
4. **Sub-System Design, Sub-System Integration Test:** In der ersten Phase wird die Architektur in kleinere voneinander abhängige Komponenten geteilt und diese im Detail beschrieben, in der zweiten Phase werden diese spezifizierten Komponenten getestet.
5. **Design der Module, Modultests:** Das Design der einzelnen Module der zuvor spezifizierten Sub-Systeme werden in der ersten Phase spezifiziert, in der zweiten Phase wird die korrekte Funktionalität dieser analysiert.

6. **Programmierung (Coding):** Dies ist die einzige Aktivität im V-Modell, welche sich nicht in zwei Phasen teilt; Die Software wird implementiert und stellt anschließend, mit der in Phase I erstellten Dokumentation, die Grundlage für die zweiten Phase.

Auch hier wird abschließend ein konkretes Beispiel zur Umsetzung eines Softwareprojekts mit Hilfe des V-Modells gegeben. Als Grundlage dient das im Kapitel "Der Wasserfallprozess" (2.9) vorgestellte Szenario, allerdings ändert sich dieses Mal modellbedingt die Vorgangsweise. Das Team beginnt mit Phase I: Es kommt wie im Wasserfallmodell zur Spezifikation der Anforderungen und im zweiten Schritt zur Spezifikation des Systems. Dann wird grob eine Übersicht der gesamten Software-Architektur erstellt, welche anschließend in mehrere Sub-Systeme unterteilt wird. Diese Sub-Systeme werden detailliert spezifiziert und im nächsten Schritt das genaue Design der einzelnen zusammenhängenden Module dieser festgelegt. Es folgt der Schritt der Implementierung, welcher zusammen mit der zuvor erstellten Dokumentation die Grundlage für die zweite Phase bildet. In dieser werden die einzelnen Module mit Hilfe von Modultests überprüft, die Subsysteme mit Integrationstests verifiziert, das System mit einem Systemtest untersucht und die Architektur des Systems validiert. Finden sich in diesen Schritten Fehler, werden die zuständigen Teammitglieder informiert und es folgt ein Sprung zu Beginn von Phase I, anderenfalls können die finalen Installations- und Akzeptanztests durchgeführt werden. Der V-Modell-Prozess ist somit beendet. In diesem Beispiel wird auch ein großer Nachteil des V-Modells ersichtlich: Nachträgliche Änderungen im Code bei Fehlverhalten sind nicht umsetzbar. Diesen Missstand beseitigt das nachfolgend vorgestellte W-Modell.

Das W-Modell

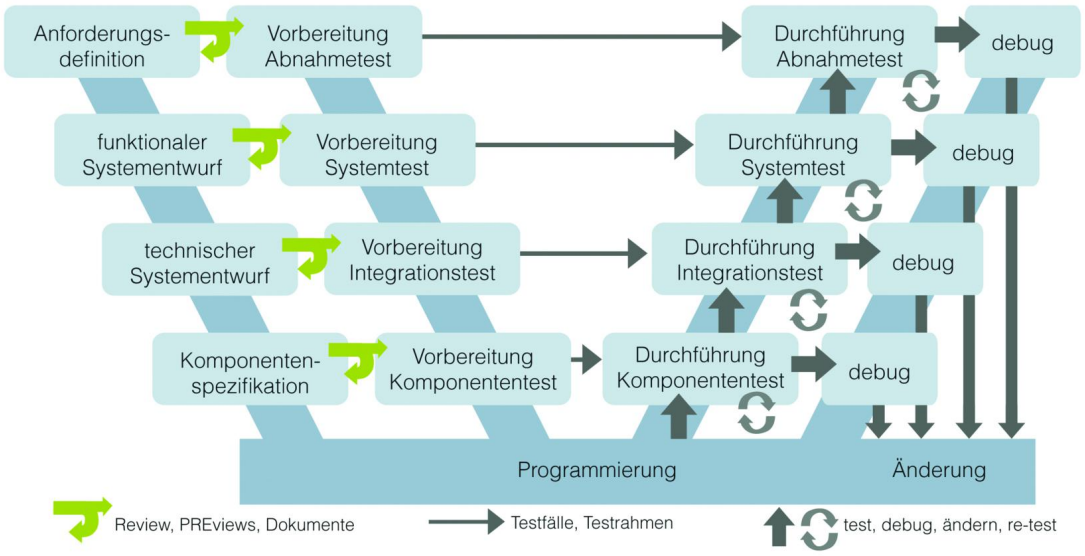

Abbildung 2.13: W-Modell (nach [SRWL06])

Abbildung 2.13 zeigt eine graphische Repräsentation des W-Modells, welches eine Erweiterung des klassischen V-Modells, welches in 2.9 (Das V-Modell) vorgestellt wurde, um parallele Änderungs- und Debugverfahren in der Testphase darstellt. Es wird sozusagen graphisch parallel ein zweites V hinzugefügt. Dokumente zur Sicherung der Qualität werden von den Testexperten zur Erstellung der Testfälle verwendet. Die Tester können dann jederzeit Änderungen in der Software beantragen. Neben der Anforderungsspezifikation sind diese Tests dann ein Hinweis darauf ob die Spezifikation vollständig bzw. ausreichend für die Erstellung der Software ist. Diese Vorgehensweise wird als Preview-Verfahren bezeichnet, welches im Gegensatz zum gegenteiligen Review-Prozess, einen Blick in die Zukunft wirft [SRWL06].
Das W-Modell führt einige Neuerungen bezüglich des Testvorgehens im Vergleich zu dem starren V-Modell ein: Jeder Schritt der ersten Phase wird um Vorbereitungen für die korrespondierenden Tests aus Phase II erweitert, der Akzeptanz- und Systemtest sowie der Integrations- und Komponententest wird zusätzlich zur bestehenden Dokumentation erstellt. Der Schritt der Programmierung wird um jederzeit durchführbare Änderungen ergänzt. In der zweiten Phase werden alle Tests um ein Debugging ergänzt. Werden Fehler gefunden, können diese nun jederzeit ausgebessert und die Tests erneut durchgeführt werden [SRWL06].
Wieder wird auf das praktische Beispiel, welches im vorangegangenen Kapitel (2.9) bereits verwendet wurde, Bezug genommen. In der ersten Phase werden mit Hilfe des W-Modells bereits die jeweiligen Tests der zweiten Phase vorbereitet. In der zweiten Phase können dieses Mal ein oder mehrere Fehler mit Hilfe des Debugging leicht(er) lokalisiert und durch die eingeführten Programmieränderungen flexibel ausgebessert werden. Der Prozess endet wie beim V-Modell ebenfalls nach Ende des Akzeptanztests. Der große Vorteil des W-Modells liegt somit auf der flexiblen Reaktion bei Fehlern und Codeveränderungen.

Das Spiralmodell

Beim Spiralmodell wird der Entwicklungsprozess in vier Phasen aufgeteilt, die alle mehrmals durchlaufen werden: In jedem Durchgang werden bestimmte Produkte entwickelt, diese bauen auf den Erzeugnissen des vorherigen Laufes auf und stellen die Basis für den folgenden Durchlauf, dabei wird auf die potentiellen Risiken des Projektes eingegangen. Für jedes Projekt werden unterschiedlich viele Durchläufe verwendet, die Anzahl dieser Durchgänge bestimmt der/die Projektleiter während des Projekts, wobei hierfür sehr viel Erfahrung benötigt wird. Es folgt eine Beschreibung der Phasen des Spiralmodells [Boe88]:

- Bestimmung des Zieles: Die Ziele und Erzeugnisse des Durchlaufs inklusive möglicher Alternativen und Restriktionen werden bestimmt und analysiert.
- Risikoanalyse: Die Risiken für die Ziele und deren Varianten werden festgestellt und Strategien zur Beseitigung der Ursachen werden erarbeitet.
- Durchführung der Arbeitsschritte: Die für das Projekt benötigten Produkte werden erstellt. Die Entwicklung kann beispielsweise mit Hilfe des Wasserfall- oder V-Modells erstellt werden.
- Planung des nächsten Zyklus: Am Ende jedes Durchlaufs wird ein Review erstellt und darauf aufbauend beginnt die Planung des nächsten Durchganges.

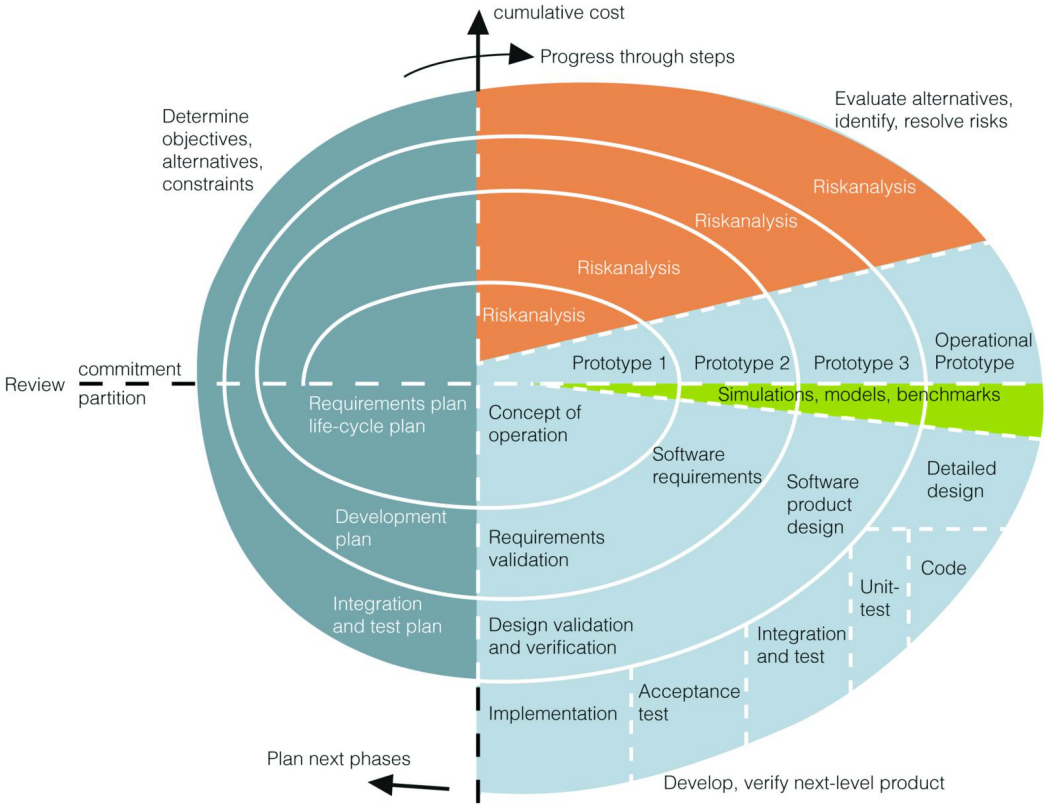

Abbildung 2.14: Spiralmodell mit vier Durchläufen (nach [Boe88])

Die Abbildung 2.14 demonstriert das Vorgehen im Spiralmodell mit vier Durchläufen. Das Projekt endet im letzten Durchgang mit der Implementierung der Software nachdem alle Projektrisiken eliminiert wurden.
Das Modell ist für Projekte mit vielen Beteiligten geeignet. Die Bestimmung der benötigten Durchgänge wird während der Laufzeit des Projekts festgestellt, eine vorherige Planung der Zeit und Kosten ist kaum möglich [GKZ04]. Der Einsatz des Spiralmodells kann die Produktivität um bis zu 50% steigern, allerdings benötigt die Durchführung sehr viel Erfahrung. Übervorsichtiges Vorgehen kann das Projekt verzögern, eine unvollständige Risikoanalyse führt allerdings wiederum zu Schwierigkeiten im nächsten Durchlauf. Die Erstellung eines Risikoplans ist allerdings auch in allen anderen gängigen Software-Vorgehensmodellen meist leicht einzubinden und wird für einen erfolgreichen Projektabschluss unbedingt empfohlen [Boe88].
Folgende Projekteigenschaften haben entscheidenden Einfluss auf den Einsatz des Spiralmodells [Boe88]:

- Ist ein niedriges Risiko für Themen wie beispielsweise der Erstellung eines schlechten User-Interfaces oder einer Verfehlung der angestrebten Applikationsleistung gegeben bei gleichzeitigem hohen Risiko in Bereichen wie dem Budget, dem Zeitplan und der

Projektkontrolle, sind Gemeinsamkeiten zwischen dem Spiral- und dem Wasserfallmodell beobachtbar.

- Falls die Anforderungen an ein Software-Produkt sehr stabil sind, das heißt es wird von einem niedrigen Risiko für große Code-Veränderungen durch Änderungen im Entwicklungsplan ausgegangen, und falls Fehler im Projekt ein großes Risiko darstellen, wird das Spiralmodell in Richtung präziser Spezifikation und formaler Programmentwicklung angepasst.

- Falls das Projekt ein niedriges Risiko beim Budget, dem Zeitplan und der Projektkontrolle hat, aber es enorme Integrationsprobleme gibt und ein hohes Risiko für die Verfehlung von Kundenwünschen gegeben ist, wird das Spiralmodell in Richtung eines evolutionären Entwicklungsmodells angepasst.

- Ist eine automatisierte Software-Erstellung (Code-Generierung) möglich, wird das Spiralmodell durch ein Transformationsmodell erweitert.

- Kommt es zu einem Mix der in den letzten vier Punkten genannten Risiken, wird das Spiralmodell mit einem angepassten Mix aus den oben angeführten Modellen erweitert.

Das Testverfahren im Spiralmodell wird von Böhm nicht näher spezifiziert, es wird allerdings in der Abbildung 2.14 auf die Erstellung eines Test- und Integrationsplans sowie die Durchführung von anschließenden Unit-, Integration- und Acceptance-Tests bei der beispielhaften Durchführung eines Projekts verwiesen. Die Verantwortung zur Ausführung der Tests im Projekt trägt somit in erster Linie der Projektleiter. Der große Vorteil des Spiralmodells gegenüber dem Wasserfallmodell oder dem V-Modell liegt in den möglichen Optionen (wie beispielsweise der frühen Beachtung einer Wiederverwendung von bestehenden Code-Fragmenten) sowie die iterative Risikoanalyse während der Projektdurchführung [Boe88].
Abschließend wird ein Beispiel zur Projektdurchführung mit dem Spiralmodell gegeben. Es wird von einem Team bestehend aus zwanzig Mitgliedern ausgegangen, welches die Aufgabe hat, das Online-Banking für eine kleine deutsche Investmentbank zu programmieren und für die Kunden einzuführen. Es besteht hohes Risiko in allen Bereichen: Das Budget, der Zeitplan, Fehler im Projekt und die Verfehlung der Kundenwünsche sind nur einige Beispiele signifikanter Projektrisiken. Durch den risikobasierten Ansatz ist hierfür das Spiralmodell sehr gut geeignet. Es werden am Projektanfang die Anforderungen an die Software mit Hilfe des Kunden festgesetzt und die Ziele und Erzeugnisse des ersten Durchlaufs bestimmt. Die erste Risikoanalyse definiert die möglichen Projektrisiken innerhalb dieser Phase. Für die Implementierung entscheidet sich das Team für das W-Modell mit integrierten Tests- und Änderungen. Ist dieser Prozess abgeschlossen, wird der Durchlauf analysiert und es wird der nächste Zyklus nach gleichem Muster geplant. Sind alle Fragmente der Software fertiggestellt, beginnt der Durchlauf des Zusammenfügens aller Teile des erstellten Produkts inklusive Integrationstests. Im allerletzten Durchgang wird mit Hilfe eines Akzeptanztests die Zufriedenheit des Kunden überprüft. Der Vorgang ist abgeschlossen. Die Projektlaufzeit hängt stark von der Erfahrung des Projektleiters bei der Planung ab, jeder Durchlauf kann unterschiedlich viel Zeit in Anspruch nehmen.

IBM Rational Unified Process

Der Unified Process wurde von drei Experten der Firma Rational, welche ein Teil von IBM wurde, entwickelt und wird heute von vielen großen Firmen (wie beispielsweise Intel, Visa, Oracle, u.v.m.) verwendet. Der "IBM Rational Unified Process" stellt eine konkrete Implementierung des "Unified Process" dar. Die Art der Verwendung gestaltet sich sehr unterschiedlich: Viele Organisationen haben aus dem IBM Rational Unified Process (kurz RUP) eigene Software-Entwicklungsprozesse abgeleitet und verwenden nur die formalen Grundlagen dieser Methode der Softwareentwicklung. RUP wird nach wie vor kommerziell vertrieben und ständig, im Auftrag von IBM, die derzeitigen Eigentümer der Firma Rational, überarbeitet und verbessert [Kru04].
Aus den Arbeiten am "IBM Rational Unified Process" entstand die "Unified Modeling Language" (kurz UML), welche eine graphische Repräsentation der Spezifikation, Konstruktion und Dokumentation von Software-Artefakten ermöglicht und aus der modernen Entwicklung heutiger Informationssysteme nicht mehr weg zu denken ist [BD04]. UML konnte dann zur Entwicklung eines objektorientierten Entwicklungsmodells eingesetzt werden, welches durch den Einfluss verschiedener Unternehmen schließlich 1999 zum "Rational Unified Process" weiterentwickelt wurde [GKZ04].
Die zentralen Eigenschaften des "Unified Process" lassen sich wie folgt zusammenfassen [GKZ04]:

- iterativ und inkrementell: Nach der Planungsphase am Anfang des Projektes werden die erforderlichen Arbeitsschritte mehrmals durchlaufen um die Qualität der erstellten Software zu steigern und neue Features einbauen zu können. Die Parallelität in der Erstellung der einzelnen Schritte zur Erstellung wird dabei forciert. Die Abbildung 2.15 zeigt die Vorgehensweise bei einem iterativen Prozessmodell. Im Vergleich zu einem linearen Prozess (wie etwa dem Wasserfallmodell 2.9) werden durch die Iterationen im Endeffekt sehr viel Zeit und somit auch Kosten für das Projekt eingespart [JBR99].

- Steuerung des Unified Process basiert auf Anwendungsfällen: Nach der Analyse der benötigten Anforderungen an die Software wird basierend auf den Anwendungsfällen das Design der Software festgelegt. In der Implementierung wird die erstellte Software stets mit der beschriebenen Funktionalität des Anwendungsfalls verglichen, in der späteren Testphase wird diese verifiziert. Die Anwendungsfälle sind somit fester Bestandteil jeder Phase des Projektes im Unified Process.

- Fokus liegt auf der Architektur: Sie liefert das Fundament für die gesamte Softwareerstellung. Die benötigten Komponenten, das Betriebssystem, auf welchem die Software ausgeführt werden soll und alle weiteren notwendigen Einzelheiten werden idealerweise von der Architektur beschrieben.

Der Unified Process besteht aus folgenden Elementen [GKZ04]:

- Rollen: Definition des Verhaltens und der Zuständigkeit einer Person oder einer Menge von Personen in einem Team. Jede Rolle wird mit bestimmten Aktivitäten und Artefakten verknüpft.

- Aktivitäten: Eine Aktivität wird durch mehrere bestimmte Arbeitsschritte definiert, pro Aktivität wird ein Ziel bestimmt.

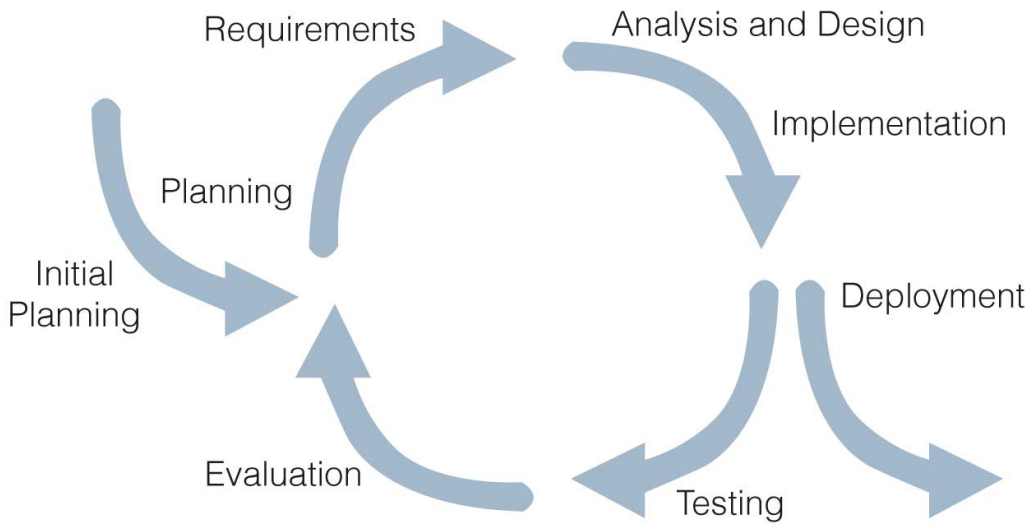

Abbildung 2.15: Ein iterativer und inkrementeller Prozess (nach [Kru04])

- Artefakte: Artefakte sind Teile des gesamten Produkts, die im Unified Process in jeder Phase des Projektes zu finden sind (Dokumentation, Modelle, Code-Ausschnitte, ...).
- Vorgehen: Abbildung 2.16 zeigt die Arbeitsschritte und Phasen des UP. Pro Iteration findet jeder Arbeitsschritt einmal statt, die Iterationen sind in Phasen gegliedert, diese werden nachfolgend erläutert [GKZ04].

Die erste Phase stellt die Konzeptionsphase(engl. Inception) dar: Es wird ein Projektplan erstellt und mögliche Kriterien zum Abschluss der Phase müssen gefunden werden. Vorschläge hierfür sind in [JBR99] zu finden:

- Bestimmung des Fokus des Systems: Die Grenzen des Systems werden ermittelt, beteiligte Personen und die Schnittstellen werden definiert.
- Unstimmigkeiten werden beseitigt
- Entwurf: Definition der Anforderungen und der Architektur
- Präsentation des Projekts für den Abnehmer

In der Entwurfsphase (engl. Elaboration) werden die Anforderungen analysiert und in ein Anwendungsfalldiagramm übersetzt. Anschließend wird über eine mögliche Umsetzung der Anwendungsfälle in Klassen und deren Schnittstellen nachgedacht, es folgt eine erste Implementierung dieser Struktur. Ein erster Projektplan wird erstellt und erste Testfälle werden entworfen.

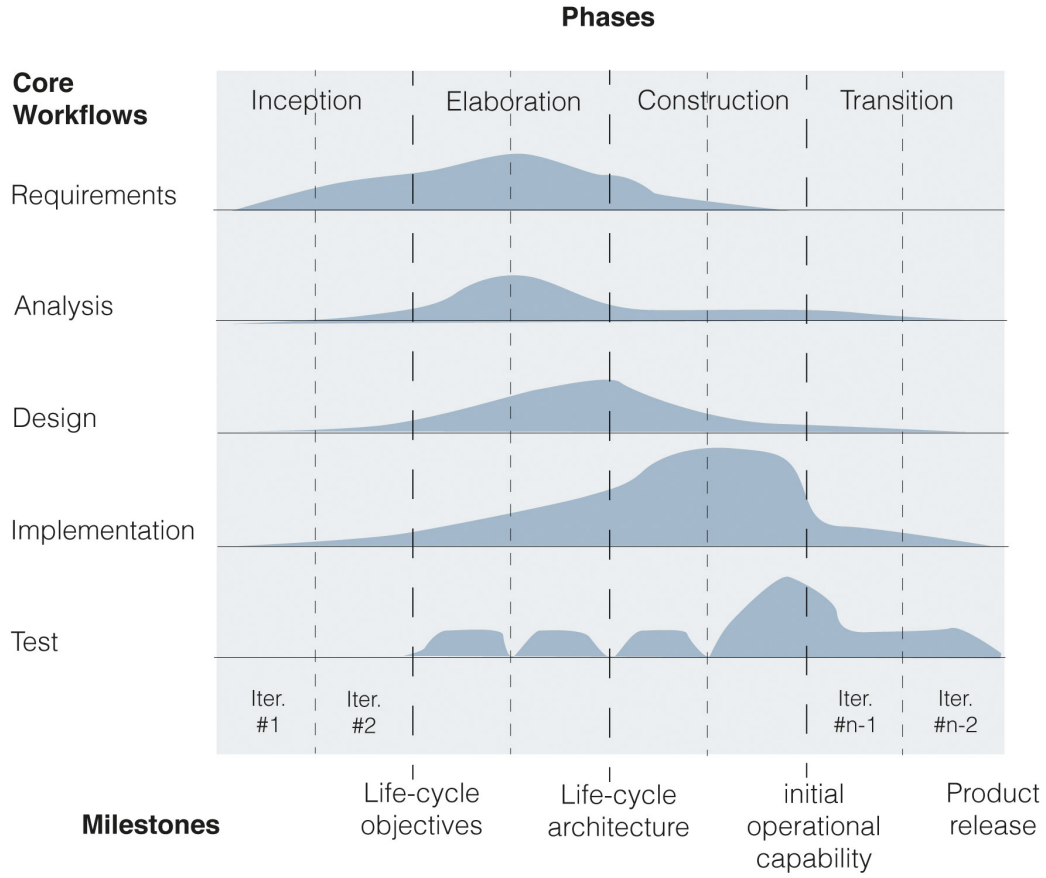

Abbildung 2.16: Phasen, Iterationen und Arbeitsschritte im "Unified Process" (nach [JBR99])

In dieser Phase kann ebenfalls bereits eine erste Version des Anwenderhandbuches und der Online-Hilfe erstellt werden.
Die dritte Phase stellt die Konstruktionsphase (engl. Construction) dar. Die Komponenten werden entwickelt, getestet und zu einem fertigen Produkt zusammengefügt. Die Dokumentation wird ständig mit den Änderungen im System angepasst und mit Hilfe eines Integrationstests wird die korrekte Kollaboration der Klassen festgestellt. Der Projektfortschritt wird laufend überprüft, Benutzerhandbuch und Hilfe sind erstellt. Alle Komponenten sind getestet, noch vorhandene Fehler werden in der nächsten und letzten Phase des "Unified Process" ausgebessert.
In der Übergangsphase (engl. Transition) wird das vollendete Produkt in der Zielumgebung des Kunden installiert und letzte Integrationstests werden durchgeführt. Gefundene Fehler werden ausgebessert und das System wird direkt beim Kunden nochmals getestet, sowie die potentiellen Benutzer eingeschult. Die Projektdokumentation wird archiviert und die gesammelte Erfahrung für zukünftige Projekte aufbewahrt [GKZ04].

Abbildung 2.17 zeigt die Phasen, Iterationen und Arbeitsschritte des IBM Rational Unified

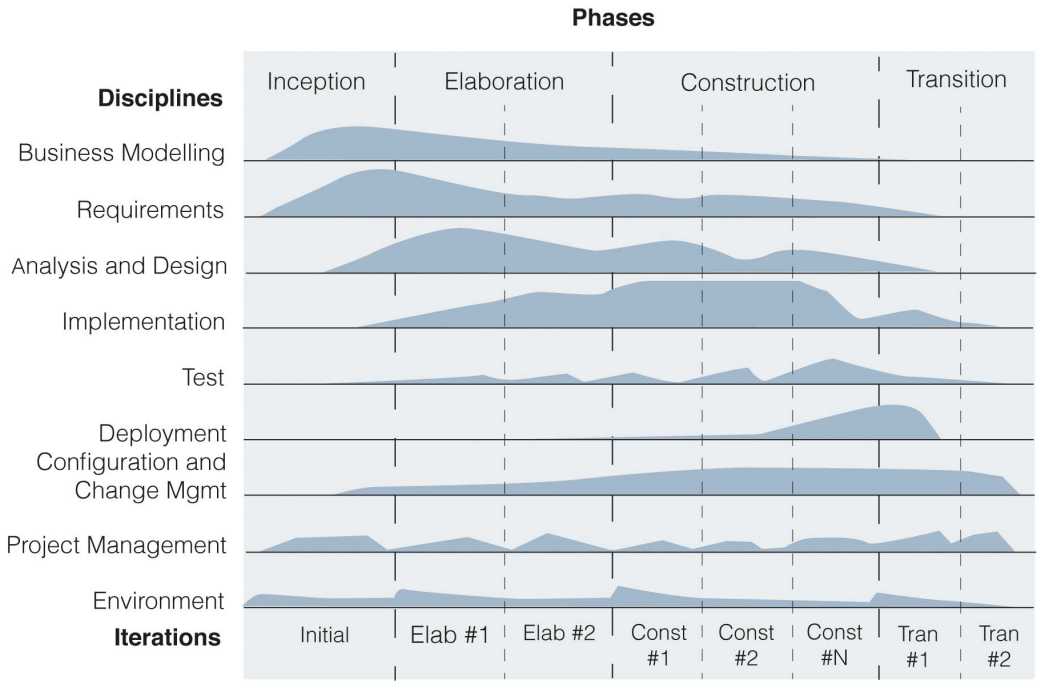

Abbildung 2.17: Phasen, Iterationen und Arbeitsschritte im IBM Rational Unified Process (nach [Kru04])

Process. Bei dieser konkreten Implementierung des UP werden neun Arbeitsschritte definiert, die jetzt näher vorgestellt werden [GKZ04]:

- Geschäftsmodellierung (engl. Business Modeling): Die Struktur, das Vorgehen, sowie die Probleme der Organisation müssen analysiert und verstanden werden. Die Zielorganisation inklusive Prozesse, Rollen und Verantwortlichkeiten wird mit Hilfe eines Geschäftsprozess- und eines Geschäftsobjekt-Modells dargestellt.

- Anforderungsanalyse (engl. Requirements): Mittels eines Anwendungsfalldiagramms werden die ermittelten Anforderungen für das Zielsystem modelliert und die Benutzerschnittstellen definiert. Der Fokus liegt auf der engen Zusammenarbeit zwischen Kunden und Entwicklern zur korrekten Darstellung des Ziels.

- Analyse & Entwurf (engl. Analysis & Design): Die Anforderungen werden in ein konkretes Implementierungsmodell übersetzt d.h. die Architektur der zu entwickelnden Software wird entworfen und das Design der Datenbank wird ermittelt.

- Implementierung (eng. Implementation): Im "IBM Rational Unified Process" gibt es verschiedene Implementierungskonzepte. Der so genannte Build ist eine lauffähige Software, die in bestimmten Zeitabständen immer wieder überarbeitet wird, so dass ein neuer Build entsteht. Im Integrationsschritt werden Artefakte der Entwicklung zusammengefügt. Dies muss zumindest einmal pro Iteration geschehen. Das Konzept der Prototypen dient der Risikominimierung. Änderungen der Architektur oder des Programmcodes können gefahrlos ausprobiert werden.

- Test: Innerhalb dieses Arbeitsschrittes werden die Testabläufe, Testprozesse, Testskripts, Testklassen und Testfälle erstellt. Im IBM Rational Unified Process dienen Tests dem Zweck die Interaktion zwischen Objekten zu verifizieren, die richtige Integration aller Komponenten der Software zu überprüfen, die korrekte Implementierung der Anforderungen zu verifizieren, um Fehler vor der Auslieferung zu identifizieren und zu korrigieren und die fehlerbereinigte Version der Software abermals zu testen. Der IBM RUP folgt einem iterativen Ansatz, das heißt die Software wird laufend während des Projekts getestet. Fehler werden dadurch so früh wie möglich gefunden, was wiederum die Kosten für die Fehlerbereinigung minimiert. Die Tests werden in die vier Kriterien Zuverlässigkeit, Funktionalität, Leistungsstärke der Applikation und Leistung im System unterteilt. Für jede dieser Sektionen definiert der Prozess die Testplanung, das Testdesign, die anschließende Implementierung, die Ausführung und Evaluierung [Kru04].

- Auslieferung (eng. Deployment): Übergebung des fertigen Projektes an den Kunden inkl. Beta-Tests, Installation und Einschulung der Benutzer.

- Konfigurations- und Änderungsmanagement (eng. Configuration & Change Management): Das Konfigurationsmanagement teilt die verschiedenen Artefakte verschiedenen Mitarbeitern zu und identifiziert Abhängigkeiten. Das Änderungsmanagement verwaltet die Änderungswünsche der Kunden.

- Projektmanagement (eng. project management): Verfolgung, Überwachung und Planung des gesamten Projektes.

- Infrastruktur (eng. Environment): Die Entwickler werden mit Prozessen und Tools zur Fehlerminimierung unterstützt.

Abschließend soll ein konkretes Beispiel zur Anwendung des "IBM Rational Unified Process" gegeben werden. Ein Projektteam mittlerer Größe (zwanzig Personen) wird beauftragt eine firmenspezifische Controlling-Lösung zu erstellen und in das Unternehmen zu integrieren. Der Projektleiter entscheidet sich für den Einsatz des "Rational Unified Process" und es wird mit der Geschäftsmodellierung, welche die Organisation in der Gesamtheit erfasst, begonnen. Sind die Geschäftsprozess- und Geschäftsobjektmodelle fertig gestellt, beginnt die Anforderungsanalyse. Zusammen mit den Kunden werden UML-Anwendungsfalldiagramme erstellt, welche die gewünschten Requirements abbilden. Im Anschluss werden die Anforderungen mit Hilfe der Diagramme in eine entsprechende Software-Architektur übersetzt. Es werden erste Prototypen von Software-Artefakten implementiert und ausprobiert. Diese werden später im Integrationsschritt zu größeren Schritten zusammengefügt. Die einzelnen Module werden mit Unit-Tests überprüft,

Integrationstests sorgen für die Zusammenarbeit der abhängigen Teile. Die Implementierung und die Tests sind iterativ, sie werden wiederholt bis das erwartete Ergebnis erreicht wird. Ist dieser Zustand erreicht, wird mit der Auslieferungsphase begonnen: Die fertige Software wird den Kunden näher gebracht und es findet ein Beta-Test statt. Kommt es zu Änderungswünschen der Kunden wird das Configuration & Change Management tätig, die zuvor eingeteilten Zuständigen werden diese Änderungen veranlassen beziehungsweise umsetzen.

Open Unified Process

Der "Open Unified Process" (kurz.: OpenUP) stellt einen Entwicklungsprozess dar, der von der Eclipse Foundation als Open-Source-Projekt als Teil des "Eclipse Process Framework" entwickelt wird und stark am "IBM Rational Unified Process" orientiert ist [Amb02].
Es werden nur fundamentale Konzepte für diesen Softwareentwicklungsprozess zur Verfügung gestellt, für Themen wie die Teamgröße, Sicherheit, vertragliche Gegebenheiten, usw. gibt es keine definierte Methoden, diese können jedoch bei Bedarf jederzeit zum erweiterbaren "Open Unified Prozess" hinzugefügt werden oder auch wieder entfernt werden. Iterative und inkrementelle Ansätze innerhalb eines strukturierten Ablaufs werden mit Hilfe des "Open Unified Prozesses" umgesetzt. OpenUP basiert auf Anwendungsfällen und Szenarien, Risikomanagement und einem Architektur-zentrierten Ansatz für die Entwicklung [Ecl].
Der "Open Unified Process" basiert auf den folgenden vier essentiellen Prinzipien, welche die Grundlage für die Interpretation der Rollen, der Definition der Arbeitspakete und der ausführenden Aufgaben bilden [Ecl]:

- Die Zusammenarbeit des Teams soll gestärkt werden um das Interesse zu fördern und ein Verständnis für das Projekt zu bilden.
- Die Prioritäten werden balanciert um die entwickelte Lösung optimal auf die Kundenwünsche auszulegen.
- Der Fokus wird sehr früh im Verlauf des Projekts auf die Architektur gelegt um Risiken zu minimieren und die Entwicklung effizient zu organisieren.
- Ein ständiges Feedback der Kunden soll eingeholt und eine ständige Verbesserung der Entwicklung durchgesetzt werden.

Die bereits oben erwähnten Rollen des OpenUP werden nun näher erläutert. Die so genannten Stakeholder repräsentieren Interessenten, die sich für den Projektausgang interessieren. Die Analysten holen sich Informationen über das Projekt von den Stakeholders um das zu lösende Problem zu verstehen und Anforderungen definieren zu können. Die Architekten sind für die Architektur der Software verantwortlich, sie legen das Design und die Implementierung fest. Die Entwickler sind zuständig für die Implementierung aller Teile des Systems. Die Tester legen die zentralen Aktivitäten, wie die Definition, Implementierung und Durchführung notwendiger Tests fest, dokumentieren und analysieren die Ergebnisse anschließend. Der Projektmanager leitet die Planung des Projekts in Zusammenarbeit mit den Stakeholdern und dem Team. Er koordiniert die Interaktion zwischen den Stakeholdern und führt das Team zum angestrebten Ziel. Die so

genannte Any Role repräsentiert jeden, der innerhalb des Teams eine allgemeine Aufgabe erfüllen kann [Ecl].
Der "Open Unified Prozess" lässt sich in folgende vier iterative Phasen einteilen [Ecl]:

1. **Beginn des Projekts (Inception Phase):** Das Projekt wird gestartet, die Iterationen werden geplant und angepasst, die Anforderungen werden identifiziert und kontinuierlich verbessert und abschließend folgt der Entwurf beziehungsweise die Verbesserung des technischen Ansatzes. Diese Phase hat das Ziel Verständnis für die Entwicklung zu schaffen, die Kernfunktionalität des Systems zu identifizieren, eine mögliche Lösung des Problems zu entwickeln und die Kosten, den Zeitplan und die Projektrisiken zu erfassen.

2. **Arbeitsphase (Elaboration Phase):** Die Iterationen dieser Phase werden geplant und angepasst, die Anforderungen werden identifiziert und kontinuierlich verbessert, die Architektur wird definiert, die Lösung wird stetig verbessert und anschließend getestet. Diese Phase hat das Ziel ein besseres Verständnis für die Anforderungen der erstellenden Software zu entwickeln, das Design der Architektur festzulegen, diese zu implementieren und zu validieren, die Feststellung erheblicher Risiken und eine akkurate Zeit- und Kosteneinschätzung zu gewährleisten.

3. **Konstruktionsphase (Construction Phase):** Die Iterationen werden geplant und angepasst, die Anforderungen werden identifiziert und adaptiert, die aktuelle Entwicklung wird stetig verbessert und getestet. Diese Phase hat das Ziel ein komplettes Produkt auf iterativer Basis zu entwickeln und die Entwicklungskosten zu minimieren und Arbeitsvorgänge zu parallelisieren.

4. **Übergangsphase (Transition Phase):** Die Iterationen werden geplant und angepasst, die Lösung wird stetig weiterentwickelt und anschließend getestet. Die Phase sieht Beta-Tests vor, wobei die Benutzer feststellen können ob ihre Erwartungen von der Software erfüllt werden. Des Weiteren werden die Stakeholder informiert, dass die Auslieferung der Software abgeschlossen ist.

Diese vier Phasen können beliebig oft iteriert werden und ergeben somit den so genannten "Open Unified Delivery Process", welcher einen kompletten und integrierten Ansatz zur Durchführung eines spezifischen Projekttyps darstellt. Er beschreibt einen vollständigen Projekt Lebenszyklus und wird als Referenz zur Durchführung ähnlicher Projekte verwendet. Die Abbildung 2.18 zeigt diesen Delivery Process: Jede Phase kann, je nach Art und Anforderungen des Projektes, beliebig oft iteriert werden [Ecl].
Der Open Unified Prozess kann auch als Basis Prozess gesehen werden, der durch verschiedene Plug-ins erweitert oder weiter vereinfacht werden kann (beispielsweise kann die Modellgetriebene Entwicklung innerhalb des OpenUP realisiert werden) um den Anforderungen des Projekts zu entsprechen. Neben den Plug-ins kann auch der (im Eclipse Framework mitgelieferte) Eclipse Process Framework Composer eingesetzt werden um den Prozess selbst auf die Bedürfnisse des Projektleiters zuzuschneiden. Organisationsspezifische Vorlagen können in den Prozess integrieren werden und eigene Rollen und Aufgaben können in den OpenUP eingefügt werden [Ecl].

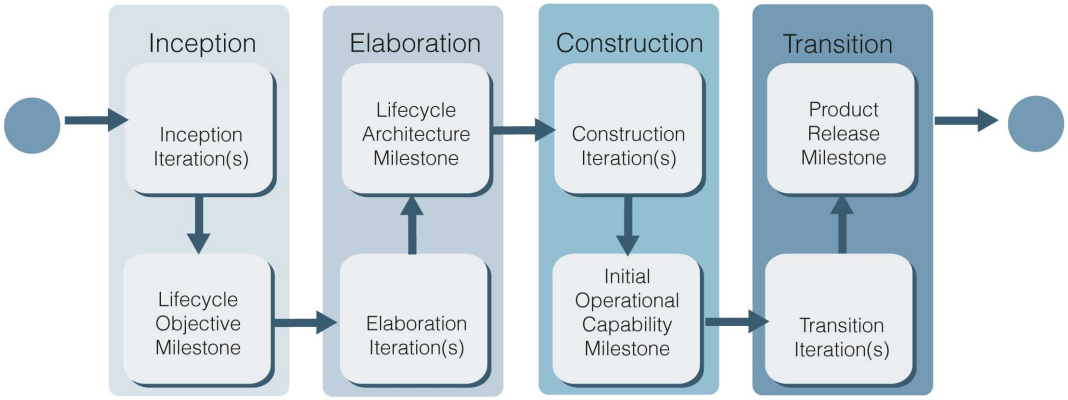

Abbildung 2.18: OpenUP Delivery Process (nach [Ecl])

Zusammengefasst ist der "Open Unified Prozess" ein minimaler, vollständiger und erweiterbarer Prozess. Er forciert den Einsatz agiler Methoden und Prinzipien, welche in einem strukturierten Lebenszyklus des Projekts den Schwerpunkt auf die fortlaufende Auslieferung von Qualitätssoftware, welche die Anforderungen des Kunden befriedigt, setzt [Ecl].
Am Ende dieses Kapitels wird dem Lesenden noch der "Open Unified Prozess" anhand eines Beispiels praktisch näher gebracht: Das Projekt beginnt in der Inception Phase und das Projektmanagement plant die Anzahl der Iterationen dieser Phase. Die Anforderungen an die Software werden analysiert und die technische Planung anschließend umgesetzt. In der Elaboration Phase wird wiederum die Anzahl der Iterationen festgesetzt, aufbauend auf die technische Planung folgt die Planung der Architektur. In der Construction Phase werden abermals die benötigten Iterationen bestimmt, es wird die Implementierung und das gleichzeitige Testen realisiert. Die Übergangsphase legt ebenfalls die Wiederholungen am Beginn fest, es folgen Beta-Tests um die Kundenzufriedenheit zu ermitteln. Diese vier Phasen können innerhalb des Projekts beliebig oft wiederholt werden, es kommt zu Anpassungen bei den Iterationen jeder Phase, zu Verbesserungen der bestehenden Lösungen, zu Veränderungen in der Implementierung und im Testplan und zur Erhöhung der Kundenakzeptanz. Der "Open Unified Process" erlaubt innerhalb dieser Phasen Kreativität und Eigeninitiative, da er nur als Basiskonzept für iteratives Vorgehen gedacht ist.

Extreme Programming (XP)

Der "Extreme Programming" (kurz XP) Prozess verfolgt das Ziel das Projekt auf die Kundenbedürfnisse auszurichten und folgt dem Beispiel eines agilen Prozesses. Auch bei fast fertigen Produkten sollen Änderungswünsche des Kunden noch umsetzbar sein, der enge Kontakt mit dem Kunden steht im Fokus. Die Vorgehensweise ist eher auf kleine Teams ausgelegt, bei größeren Projekten ist die direkte Kommunikation zwischen Entwicklern und Kunden schwer umsetzbar. Des Weiteren sollte der Kunde häufig kontaktiert werden um über Änderungen mitbestimmen zu können [GKZ04].

Zentrale Punkte des Extreme Programming lassen sich nach Kent wie folgt zusammenfassen [Bec99]:

- Permanenter Review-Prozess der Implementierung
- Programmierer und Benutzer müssen ständig Tests durchführen
- Das einfachste Design muss verwendet werden
- Die Architektur muss ständig verfeinert werden
- Der Integrationsprozess muss mehrmals täglich ausgeführt werden
- Sehr kurze Iterationszeiten im Projektverlauf müssen eingehalten werden

Abbildung 2.19 zeigt den Ablauf eines XP-Prozesses. Auch hier finden sich die vier Elemente der klassischen Vorgangsweise zur Erstellung von Software: Planung, Entwurf, Implementierung, Test. Einige Begriffe des Prozesses werden nun näher erläutert [Bec99]:

- Versionsplanung: Mit Hilfe der mit dem Kunden geplanten Anforderungen wird ein Release- und Iterationsplan erstellt, der die Wünsche des Kunden nach Priorität umsetzt.
- Iterationen: Die Iterationen, welche annähernd die gleiche Ausführungsdauer besitzen müssen, werden in einem eigenen Meeting geplant. Wird die geplante Zeit einer Iteration überschritten, müssen weitere Anforderungen auf die nächste Iteration verschoben werden. Deshalb werden in der Versionsplanung die Kundenwünsche (Anforderungen) nach Priorität sortiert. Die Implementierung erfolgt mittels Pair Programming.

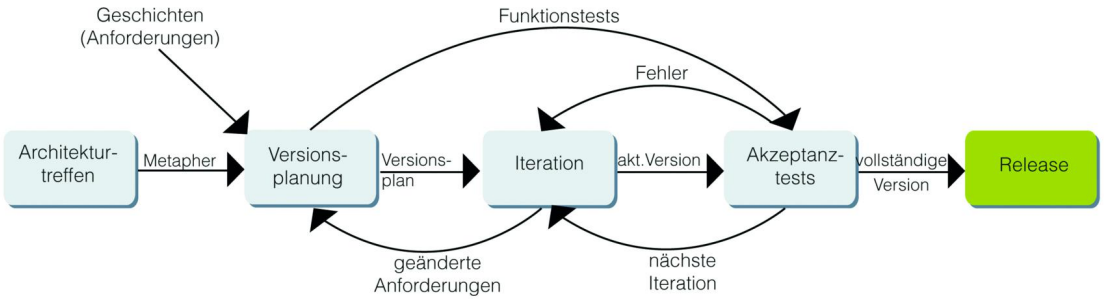

Abbildung 2.19: Extreme Programming Projekt (nach [Wel99])

Die Methoden, die im Extreme Programming Modell verwendet werden sollten, werden in Abbildung 2.20 zusammengefasst und nachfolgend näher beschrieben [Bec99]:

Kommunikation: permanente Gespräche mit dem Kunden

Metapher: entspricht einer User Story um jedem Interessenten das System näher erklären

zu können

Test-Driven Development: zuerst werden die Tests definiert und erst danach implementiert (siehe Kapitel Test-Driven Development)

Team: Permanente Kommunikation innerhalb des Teams, dies funktioniert nur bei kleiner Teamgröße, die Zusammenarbeit wird bei Extreme Programming groß geschrieben

Refactoring: Permanente Anpassungen und Verbesserungen des Programmcodes

Pair Programming: bezeichnet eine Methode aus der agilen Softwareentwicklung, bei welcher zwei Programmierer an einem Computer sitzen: Der Eine programmiert (auch driver genannt), während der Andere den geschriebenen Code überdenkt, verifiziert und Ideen sowie Ratschläge zur Verbesserung gibt (auch observer genannt). Diese zwei Rollen werden oft gewechselt [WK03]

Einfaches Design: Das einfachste funktionierende Software-Design wird verwendet

Continuous Integration: Integration wird mehrmals pro Tag durchgeführt um bestehende Software-Teile zusammenzufügen

Customer Tests: Der Kunde wird in den Entwicklungs- und Testprozess integriert

Coding Standards: Verwendung von Standards für die nahtlose Integration in andere Projekte

Small Iterations: kurze Iterationszyklen sind projektimmanent

Nachhaltiges Tempo: ein hohes Arbeitstempo ermöglicht bereits früh Ergebnisse präsentieren zu können

Collective Ownership: Alle Mitglieder des Teams (inklusive der Kunden) kennen den Code und können selbst Ideen in das Projekt einbringen

Planning Poker: Im Konsens wird der erwartete Aufwand des Projekts und der einzelnen Aufgaben geschätzt

Extreme-Programming basiert auf der testgetriebenen Entwicklung, die noch im Laufe des Kapitels "Bestehende Software-Entwicklungsprozesse" näher erläutert wird.
Abschließend wird noch ein praktisches Beispiel zum Extreme-Programming gegeben. Ausgegangen wird von einer Teamgröße von fünf Personen. Zu entwickeln ist eine Applikation, die Aktienkäufe und -verkäufe automatisiert. Der Auftraggeber ist eine einzelne Privatperson. Das Team bestimmt die Anforderungen im Gespräch mit dem Kunden, die einfachste Architektur wird entworfen und eine Metapher erstellt. Darauf folgt die Versionsplanung, die für jede Version eine

Iteration definiert, nach dieser wird ein Akzeptanztest der aktuellen Version durchgeführt und die nächste Iteration eingeleitet, wobei die Anforderungen zuvor angepasst wurden. Dieser Prozess wird solange wiederholt, bis die letzte und vollständige Version zum Release, das heißt zum produktiven Kundeneinsatz, freigegeben wird. Der Extreme-Programming Prozess endet somit. Wie an diesem Beispiel ersichtlich wird, erfordert das XP Vorgangsmodell äußerst geringen Dokumentationsaufwand.

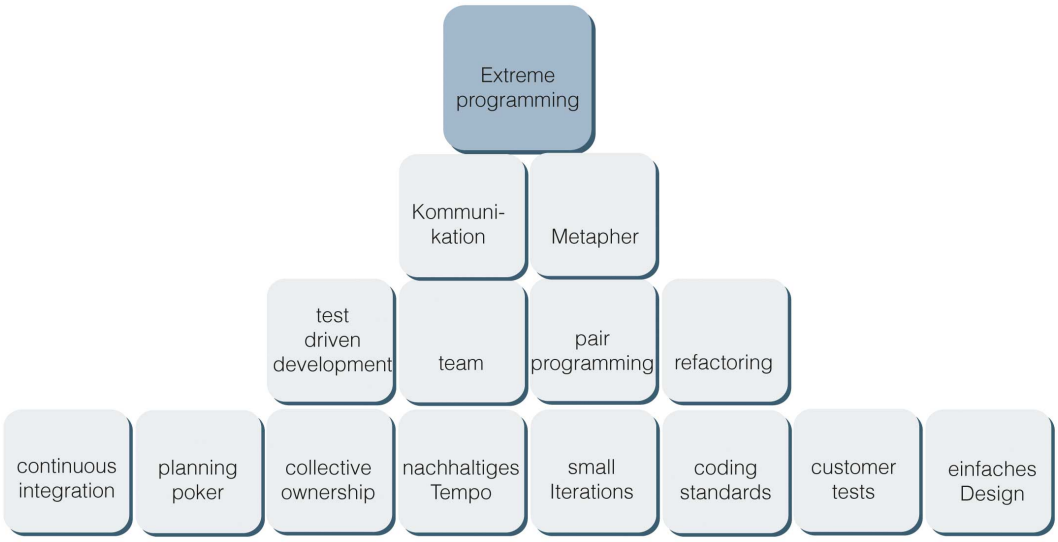

Abbildung 2.20: Verwendete Methoden bei Extreme Programming (nach [Bec99])

Crystal Family

Die Crystal Family ist eine Familie von Prozessen, die sich nahtlos in die Gruppe der agilen Softwareprozesse einreiht. Im Vergleich zu Extreme Programming (siehe 2.9 - Extreme Programming (XP)) sind die einzelnen Crystal Methoden weniger starr und orientieren sich mehr am einzelnen Projekt als an der arbeitszentrierten Umsetzung. Kernprinzipien der Crystal Family sind:

- Testautomatisierung: Die Qualität der Software wird durch Unit-Testing gesichert.
- Releases und Integration in kurzen Zyklen: Alpha-Versionen zur besseren Realisierung der Kundenwünsche werden oft erstellt.
- Häufiger Kontakt zu Kunden: Die ständige Kommunikation mit dem Kunden soll die Qualität der Sofware steigern.

Je nach der Anzahl an beteiligten Personen wird eine andere Methode (benannt nach Farben) der Crystal Family gewählt. Verschiedene Projektrisiken (Gefährdungen für das Leben (Life), das Unternehmen (Essential Money), das Geld (Discretionary Money) und den Komfort (Comfort)) der

-	**Clear**	**Yellow**	**Orange**	**Red**	**Maroon**
Persons	**1-6**	**7-20**	**21-40**	**41-80**	**81-200**
Life	L6	L20	L40	L80	L200
Essential Money	E6	E20	E40	E80	E200
Discretionary Money	D6	D20	D40	D80	D200
Comfort	C6	C20	C40	C80	C200

Tabelle 2.1: Crystal Family (nach [dev])

zu erstellenden Software werden je nach ausgewählter Methode anders bewertet. Tabelle 2.1 zeigt die verschiedenen Varianten inklusive ihrer Risikoeinstufungen. Je nach verwendeter Variante definiert der Crystal Process andere Pakete, welche die Vorgangsweise im Projekt definieren. Jede Variante hat einen anderen Kommunikations-, Dokumentations-, und Implementierungsaufwand [Coc03]. Die einfachste Methode stellt dabei der Crystal Clear Prozess dar, der durch zehnjährige Forschung im Bereich Software-Development entwickelt wurde und für kleine Teams, bestehend aus maximal sechs Projektmitarbeitern, anwendbar ist [Coc05]. Einige Vorteile dieser Methode werden nun vorgestellt [Coc05]:

- Crystal Clear basiert auf Beobachtungen vieler erfolgreicher Entwicklungsteams, andere Softwarentwicklungsmodelle basieren auf wenigen Einzelerfahrungen.
- Crystal Clear setzt den Fokus auf die wesentlichen Meilensteine des Projekts, wobei die verwendete Methodik (wie beispielsweise die Verwendung von exakt definierten Diagrammtypen) in Crystal Clear eher zweitrangig ist.
- Es muss nicht der gesamte Prozess implementiert werden. Teams, die bereits den "IBM Rational Unified Prozess" oder das W-Modell verwenden, können Teile von Crystal Clear sehr wohl in ihr Projekt integrieren.
- Eine Fülle von Informationen über die konkrete Integration des Entwicklungsprozesses in das eigene Unternehmen existiert.
- Die Crystal Clear Methode lässt sich innerhalb weniger Tage selbst erlernen.

Die Vorgangsweise der Crystal Clear Methode wird im folgenden Absatz noch näher beschrieben. Der Prozess definiert einzelne Arbeitspakete: Zu Beginn des Projektes werden in der Aufgabenbeschreibung die Ziele festgelegt. Wie im Extreme Programming erfolgt darauf aufbauend eine Versionsplanung, allerdings mit einer konkreten Festlegung der Dauer der Iterationen. Zusätzlich werden die Risiken des Projekts festgestellt und in einer Liste zusammengefasst. Der Projektstatus wird laufend dokumentiert und kommuniziert. Die Teamstruktur und das Design des Systemes werden basierend auf der Grundlage der bereits ermittelten Anforderungen, welche in einem Dokument festgehalten werden müssen, bestimmt. Des Weiteren werden Anwendungsfälle definiert. Entwürfe über das System und die Oberfläche werden gezeichnet, Notizen darüber gemacht und diese anschließend diskutiert. Danach wird ein allgemeines Objektmodell erstellt und anschließend das zuvor Geplante implementiert. Der Quellcode wird dabei klar in

verschiedene Pakete strukturiert und auf Wiederverwendbarkeit des bestehenden Programmcodes geachtet. Anschließend folgt das Arbeitspaket der Testfälle, wobei mit Hilfe von Unit-Tests automatisiert die erstellte Software auf Fehler überprüft wird. Anschließend wird eine Liste der entdeckten Mängel geführt. Alle diese beschriebenen Arbeitspakete können beliebig oft wiederholt werden. Der Crystal Clear Prozess schlägt auch die Erstellung eines Benutzerhandbuches vor. Sind alle Arbeitspakete abgeschlossen und der Kunde zufrieden mit der entwickelten Lösung, endet die Methode [Coc05].

Scrum

Der "Scrum Prozess" verfolgt die Annahme, dass sich das Projekt auf Grund seiner Komplexität nicht in einzelne Phasen oder Arbeitsschritte teilen lässt. Das Team organisiert sich und erledigt einzelne Umsetzungen der Anforderungen selbständig. Klassische Mittel oder ein zentrales Management werden abgelehnt, der Fokus liegt auf der Zusammenarbeit der Mitarbeiter. Innerhalb des Teams werden aber sehr wohl Rollen definiert, die folgend näher beschrieben werden [Sch07]:

- ScrumMaster: Er überwacht die korrekte Durchführung des Scrum Prozesses für das Projekt.
- Product Owner: Anhand der mit dem Kunden festgelegten Anforderungen definiert diese Rolle das Ziel (Product Backlog) und setzt die Priorität der einzelnen Programmteile fest.
- Team: Die Größe des Projektteams beim Scrum sollte zwischen fünf und neun Personen liegen.

Die einzelnen Teile des Scrum Prozess lassen sich wie folgt zusammenfassen [Sch07]:

- Product Backlog: Dieses wird vom Product Owner definiert und umfasst alle Merkmale der zu erstellenden Software, welche nach Priorität geordnet werden.
- Sprint Backlog: Der sogenannte Sprint, welcher eine einzelne Iteration im Entwicklungsprozess darstellt, wird geplant. Jede Aufgabe sollte dabei nicht länger als einen Tag dauern.
- Selected Backlog: Eine Liste aller Product Backlog Einträge, die zur definierten Zielsetzung eines Sprints benötigt werden, wird erarbeitet.
- Burndown Chart: Eine graphische Repräsentation des zu benötigten Restaufwands pro Sprint wird einmal täglich erstellt.
- Impediment Backlog: Die potentiellen Projektrisiken werden erfasst.

Es folgt nun eine Beschreibung des Ablaufs des Scrum-Prozess [Sch07]:

1. Täglicher Scrum: Täglich wird ein kurzes Treffen zur Besprechung des Projektfortschritts abgehalten.

2. Sprint-Planungsmeeting: Im ersten Treffen werden alle für den nächsten Durchgang erforderlichen Backlog Einträge durch den Product Owner für das Team erläutert und das Ziel des Sprints definiert.

3. Sprint-Planungsmeeting: Die Selected Backlog Einträge werden in einzelne Arbeiten aufgeteilt und verschiedenen Mitarbeitern zugewiesen.

4. Sprint: Hier wird eine einzelne Iteration umgesetzt, die idealerweise maximal ein Monat dauern sollte. Als Abschluss muss eine funktionsfähige, voll getestete und bei weiteren Iterationen verbesserte Version des Produkts vorliegen.

5. Rückblick: Nach jedem Sprint wird durch den Product Owner geprüft ob die Software mit den Anforderungen übereinstimmt.

6. Retroperspektive: Die wichtigsten Ereignisse des Sprints werden diskutiert und eventuelle Fehler analysiert.

Die Abbildung 2.21 zeigt den typischen Ablauf im Scrum-Prozess.
Scrum definiert einen reinen Projektmanagement-Prozess, das heißt es gibt keine Vorgaben zur

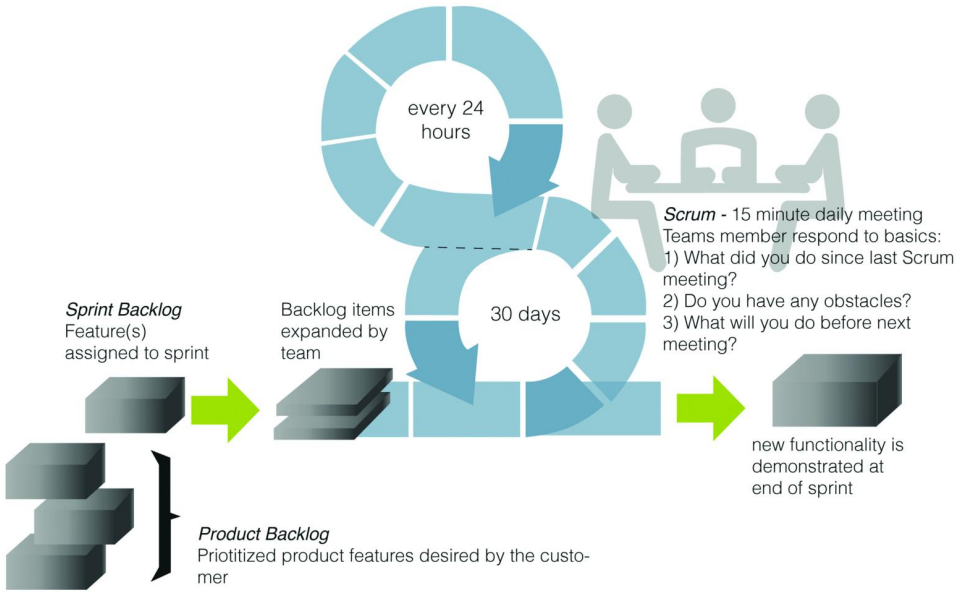

Abbildung 2.21: Der Scrum-Prozess (nach [Sch07])

Implementierung und den zu verwendenden Testmethoden. In der Praxis wird Scrum häufig mit Extreme Programming kombiniert, was zu einer Testgetriebenen Entwicklung innerhalb der Methode führt: Die vorgegebenen Methodik, beispielsweise der Daily Scrum oder der Sprint, wird plangemäß durchgeführt, für die Implementierung werden andere Vorgangsmodelle in Betracht gezogen. Das Ergebnis ist ein agiler und sehr flexibler Entwicklungsprozess, der an das

jeweilige Unternehmen ideal angepasst werden kann [Sch07].

Kanban

Kanban kann als Grundgerüst für den Scrum-Prozess betrachtet werden, jedoch werden keine fix vorgegebenen Zeiten für eine Iteration (Sprint) festgesetzt. Will ein Team den Entwicklungsprozess noch agiler gestalten, ist Kanban die mögliche Antwort. Die einzelnen Arbeitsschritte (Definition der Anforderungen, Entwicklung, Dokumentation, Testphase, ...) werden auf einem so genannten Kanban-Board in Spaltenform dargestellt. Jede Anforderung wird als Ticket einer Spalte zugeordnet und die einzelnen Stationen werden durchlaufen. Die parallele Bearbeitung mehrerer Tickets wird eingeschränkt. Damit wird gewährleistet, dass in jeder Phase alle Aufgaben fertig gestellt wurden, bevor die nächste Phase startet. In der Familie der agilen Software-Prozesse kann Kanban beispielsweise das Scrum-Modell auflockern: Die Durchlaufzeiten für die Iterationen können weiter minimiert werden und ein höherer Fokus kann auf die einzelnen Anforderungen gelegt werden [Lad08].
Kanban führt das System von Service Level Agreements ein, welches verschiedene Klassen für die Tickets definiert. Diese werden nachstehend erläutert [Lad08]:

- Standard-Klasse: Diese Tickets werden nach dem First in, First out Prinzip behandelt. Die Reihenfolge, in der Aufgaben in einer Station abgeschlossen wurden, wird in der Nächsten ebenfalls eingehalten.
- Klasse der fixen Termine: Die Aufgabe wird wenige Tage vor dem festgesetzten Termin erledigt.
- Bevorzugte Tickets: Diese Tickets haben die höchste Priorität und müssen möglichst schnell erledigt werden, andere Aufgaben können hierfür später abgeschlossen werden.
- Klasse der vagen Tickets: Diese Tickets sind von niedriger Priorität.

Verschiedene Diagramme helfen in Kanban die Ergebnisse fest zu halten. Das Flussdiagramm zeigt den jeweiligen Status der Tickets in jeder Phase des Kanban-Prozesses. Ein Work-In-Progress Diagramm präsentiert die Entwicklung der einzelnen Aufgaben und User-Stories. Das Durchsatz-Diagramm stellt die fertiggestellten Tickets pro Woche dar. In einem Bug-Diagramm werden die ermittelten Fehler im System festgehalten und wie sich diese im Laufe der Zeit entwickeln. Mit Hilfe dieser graphischen Repräsentationen kann man dann aus Fehlern lernen und Projekte in der Zukunft besser abwickeln [Lad08].
Kanban ist wie schon Scrum als reiner Projektmanagement-Prozess definiert, es gibt auch hier keinerlei Vorgaben, wie die Implementierung oder die Tests zu gestalten sind. Es bietet sich ebenso an als Entwicklungsprozess Extreme Programming zu verwenden.

Modellgetriebene Softwareentwicklung

Die modellgetriebene Softwareentwicklung definiert bestimmte abstrahierte Modelle mit Hilfe von eigenen Sprachen, die Zielumgebung inkl. Programmiersprache ist hierbei nicht von Bedeu-

tung. So genannte Generatoren übersetzen dann die mit der Modellierungssprache erzeugten Umgebungen für die jeweilige Zielplattform. Es wird aber auch eine Vorgehensweise definiert, welche die sinnvolle Integration von Modellen und Generatoren in den Entwicklungsprozess ermöglicht [SVEH07].
Die hohe Abstraktion erleichtert das Verständnis für das zu entwickelnde Produkt für alle Mitglieder des Projekts, jedoch ist die Entwicklung und Validierung des Metamodells oft mit sehr hohem Aufwand verbunden und die Fehlersuche im Modell kann sich als äußerst schwierig erweisen [SVEH07].

Es gibt verschiedene Tools für die modellgetriebene Softwareentwicklung, welche sich in

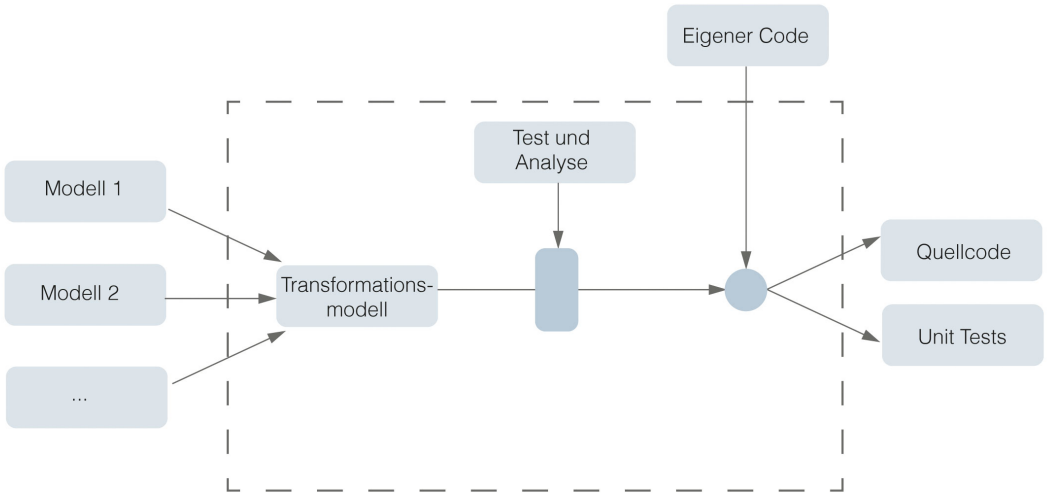

Abbildung 2.22: Modellgetriebene Entwicklung - Vom Modell zum Code (nach [SVEH07])

folgende Kategorien einteilen lassen [PTN+07]:

- Tools zur Modellierung: Sie dienen nur der graphischen Repräsentation der abstrahierten Domäne.
- Tools zur Transformation: Das bestehende Modell (welches beispielsweise in XMI definiert wurde) wird importiert, transformiert und anschließend wieder exportiert.
- Kombinierte Tools: Das Modell, die Transformation und die Codeerzeugung wird mit Hilfe einer einzigen Umgebung erstellt. Diese Tools helfen Fehler zu vermeiden.

Ein sehr gutes und bekanntes Tool ist das Eclipse Modelling Framework (EMF), welches sich nahtlos in Eclipse einfügt. Es unterstützt die Generierung von Java-Code und zusätzlich werden automatisch jUnit-Tests erstellt, die den automatisch erstellten Code testen.
Die Abbildung 2.22 zeigt den Ablauf von Modellerstellung, Transformation und Codegenerierung

im modellbasierten Entwicklungsprozess.
Die geplanten Tests bei der Modellgetriebenen Entwicklung lassen sich ebenso modellbasiert

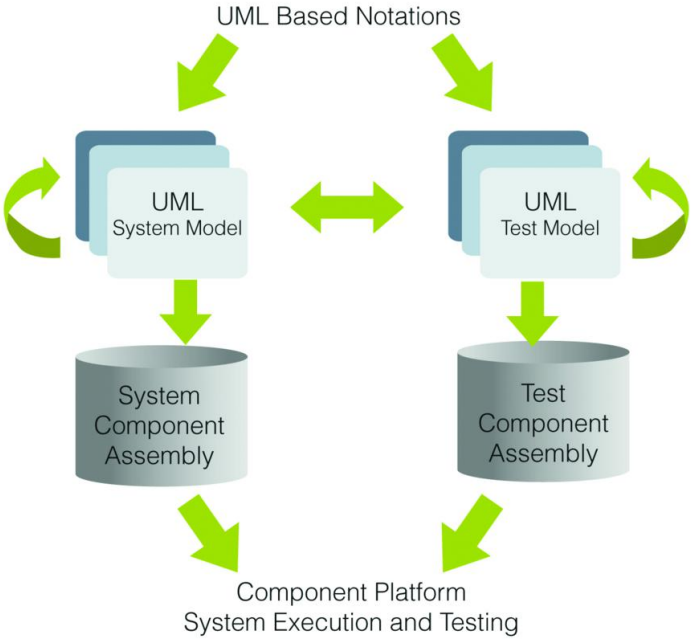

Abbildung 2.23: Modellgetriebene Entwicklung - System- und Testentwicklung (nach [BSGS04])

generieren. Das "UML Testing Profile" (kurz U2TP) ist eine Erweiterung von UML 2.0 und spezifiziert Testkonzepte innerhalb der Modellbasierten Entwicklung. Eine Modellierungssprache zur Visualisierung, Spezifikation, Analyse, der Konstruktion und Dokumentation von Artefakten eines Testsystems wird definiert. Das Testprofil unterstützt die Spezifikation und Modellierung einer Infrastruktur zum Software Testen: Die Testkomponenten, der Text-Kontext, die Schnittstellen des Testsystems, die Testprozeduren, das Setup, die Ausführung der Tests und eine anschließende Evaluierung werden definiert [BSGS04].
Die Abbildung 2.23 veranschaulicht die simultane Entwicklung des Systems und der dazugehörigen Tests im modellbasierten Entwicklungsprozess.

Feature Driven Development

Die merkmalsgetriebene Softwareentwicklung (engl.: Feature-Driven Development) wurde in den neunziger Jahren von Peter Coad und Jeff De Luca im Rahmen eines großen Java-Projekts entwickelt, die Verwendung einer objektorientieren Programmiersprache und der "Unified Modelling Language" sind für diesen Entwicklungsvorgang Voraussetzung [CLL99].
"Feature Driven Development" stellt die einzelnen Features der Software in den Fokus der En-

twicklung und definiert Rollen und einen Prozess, der sich gut in bestehende Projekte integrieren lässt. Ein Feature-Plan wird vom Chief Architect erstellt, diese Person definiert auch die Architektur der Software. Die Beschreibung von Feature Driven Development umfasst nur zehn Seiten, sie ist sehr einfach gehalten [PFP02].
Die Kundenorientierung steht im Vordergrund, es werden laufend implementierte Features vorgestellt. Durch die Aufspaltung der Software in kleine Einzelteile wird der Fortschritt leichter feststellbar, die Entwickler sind motivierter und der Aufwand pro Feature lässt sich leichter erfassen, was eine bessere Kontrolle des Projektes ermöglicht. Die Bestimmung des Fortschritts basiert aber letztendlich wiederum auf Erfahrungswerten, wird vom Chefprogrammierer laufend durchgeführt und in der Dokumentation aktualisiert [CLL99].
Es werden im Feature-Driven Development verschiedene Personenrollen definiert, welche nachfolgend beschrieben werden [CLL99]:

- Der Chefprogrammierer (Chief Programmer): Er trägt die Verantwortung für die Erstellung eines Features von der Implementierung bis zum abschließenden Test.
- Der Klassenverantwortliche (Class Owner): Er ist für den Entwurf und die Implementierung einer einzelnen Klasse verantwortlich.
- Das Feature Team: Während der Implementierung eines Features sind alle Klassenverantwortlichen, die an einem gemeinsamen Merkmal arbeiten, teil des Feature Teams.

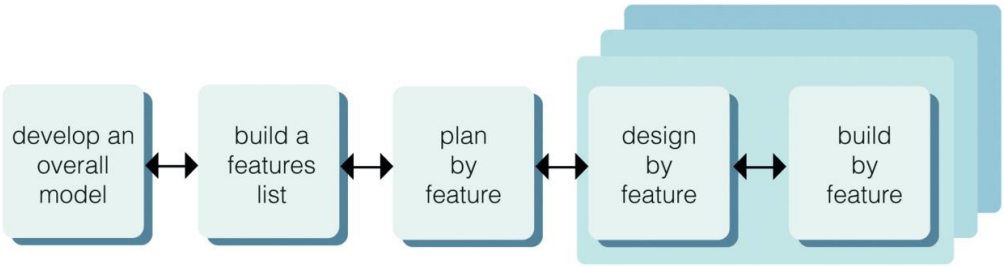

Abbildung 2.24: Feature-Driven Development Prozessmodell (nach [CLL99])

Abbildung 2.24 zeigt den fünfteiligen Entwicklungsprozess der "Feature Driven Development" Methode, die einzelnen Schritte werden nun detailliert beschrieben [PFP02] [CLL99]:

1. **Entwicklung des gesamten Modells:** Experten, Entwickler und Chief Architekt definieren den Inhalt des Systems und das zugrunde liegende Kernmodell. Ein Team bestehend aus Fachexperten wird hierbei zur Modellierung des Systems bestimmt und während des Projekts gegebenfalls um erweitert. Die vorhandenen Dokumente werden von diesem Team dann inspiziert. Es folgt ein Rohentwurf der ersten Feature-Liste und Teilmodelle der erwarteten Software werden bereits erstellt. Darauf aufbauend folgt die Gestaltung eines Gesamtmodells und mögliche Alternativen werden bestimmt und dokumentiert. Die Fachexperten verifizieren die erstellten Klassendiagramme am Ende dieses ersten Schritts.

2. **Erstellung einer Liste der Features:** Der Inhalt des Systems wird in Features eingeteilt, wobei eine Implementierung eines Merkmals maximal vierzehn Tage dauern darf. Ist ein Merkmal zu groß, wird es weiter unterteilt. Es kommt auch zur Festlegung der Prioritäten jedes Features in die folgenden Kategorien: Must have (vom Kunden gefordert), nice to have (falls noch Zeit übrig bleibt, sollte man diese Idee unbedingt noch implementieren), add it if we can (Das Feature ist nicht unbedingt notwendig, könnte aber bei verbleibender Zeit noch hinzugefügt werden) und future (Umsetzen eines Merkmals in unbestimmter Zukunft). Als Ergebnis des zweiten Schrittes wird eine komplette, nach Prioritäten sortierte Feature Liste, welche ebenfalls von den Fachexperten geprüft wird, erwartet.

3. **Die Reihenfolge der Implementierung der Features wird im so genannten "Plan by Feature" festgelegt:** Für jedes Merkmal der Software wird ein konkretes Datum zur Fertigstellung gesetzt und der/die Verantwortlichen bestimmt.

4. **Die einzelnen Features werden einzelnen Entwicklerteams zugewiesen und mit Hilfe von Sequenzdiagrammen detailliert dargestellt:** Der Chief Architect verfeinert diese Diagramme weiter nach seinen Vorstellungen. Die Diagramme werden anschließend vom Team verifiziert. Dieser Prozess wird als Design by Feature (kurz: DBF) bezeichnet.

5. **Die Features werden implementiert und getestet:** Dieser finale Schritt ist in einschlägiger Fachliteratur auch unter dem Begriff "Build by Feature" (kurz: BBF) zu finden. Das Team verifiziert den Code: Der Verantwortliche überprüft die korrekte Funktionalität anhand eines Unit Tests, es folgen Integrationstests innerhalb des Teams. Sind alle Klassen und das gesamte System getestet und werden als fehlerfrei befunden, endet dieser Schritt.

6. **Wiederholung von Schritten:** Schritt Nummer Vier und Schritt Nummer Fünf werden für jedes einzelne Feature der Gesamtliste aller Merkmale wiederholt [CLL99].

Das Testen jedes Merkmals erfolgt, wie bereits weiter oben erwähnt, bei der Feature-getriebenen Entwicklung nach der Implementierung der Klasse mit einem Modultest und bei Komplettierung aller Klassen des Features mit Hilfe von Integrationstests.

Test Driven Development

Bei der "Test Driven Development" Methode werden vor der Erstellung des Programmcodes immer die Tests für die Komponenten erstellt und erst danach kommt es zur Implementierung dieser. Bei der klassischen Software-Entwicklung werden Tests immer parallel oder nachträglich ausgeführt. Teilweise fehlen die Tests sogar komplett, weil sie von der Unternehmensführung nicht erwünscht sind oder die Zeit für das Testen nicht ausreicht. White-Box Testing führt des Weiteren dazu, dass der Entwickler das System kennt und manchmal unbewusst unzureichende Tests entwickelt. Die testgetriebene Entwicklung versucht diese Probleme zu beheben [Bec02]. Die in [Bec02] beschriebene Unit-Test Entwicklung hat folgenden Ablauf:

1. **Erstellung der Tests** für das gewünschte Verhalten der einzelnen Komponenten, ein Programmcode existiert noch nicht.

2. **Schreiben des Programmcodes:** Die in Schritt eins erstellten Tests müssen ohne Fehler terminieren.

3. **Refactoring-Prozess:** Das Entfernen von Wiederholungen, Abstraktion und Einhaltungen von Programmcode-Konventionen mit erneutem Testlauf erfolgt zum Abschluss.

Test Driven Development lässt sich auch zum Testen des gesamten Systems verwenden, hierfür werden Systemtests vor Erstellung der Software entwickelt.
Große Systeme, bestehend aus mehreren Komponenten, die unabhängig voneinander implementiert werden und eventuell noch Fremdkomponenten enthalten, können mit Hilfe von eigens erstellten Mock-Objekten, die das erwartete Verhalten der benötigten Komponenten simulieren, ebenfalls getestet werden [Bec02].
Die Abbildung 2.25 zeigt ein Ablaufdiagramm der testgetriebenen Entwicklungsmethode. Betrachtet man diese Illustration wird die Vorgangsweise der Test Driven Development deutlich: Änderungen am Programmcode werden solange wiederholt, bis der Test fehlerfrei verläuft. Ist die Entwicklung des gesamten Systems noch nicht abgeschlossen, wird ein neuer Test erstellt und der Prozess beginnt wiederum von Neuem. Sind alle benötigten Klassen implementiert und fehlerfrei endet die Test Driven Development Methode.
Abschließend werden noch einige Vor- und Nachteile der testgetriebenen Entwicklung diskutiert [GW03]:
Das Programmverständnis wird gefördert: Studien besagen, dass die Hälfte der Zeit der Programmierer bei der Wartung bestehender Software zum Verständnis des Programmcodes verwendet wird. "Test Driven Development" regt die Entwickler an ihre Implementierung nicht mit Worten zu beschreiben. Stattdessen werden Programmcode und Test zur Erklärung verwendet. Überdies wird garantiert, dass die Testfälle immer am aktuellsten Stand sind.
Ein weiterer wesentlicher Vorteil liegt in der gesteigerten Effizienz bei der Programmierung. Fehler im Programmcode können sehr schnell gefunden werden, jedoch wird diese gesparte Zeit durch die Erstellung der Tests wieder kompensiert.
Durch die Verwendung der testgetriebenen Entwicklung ist die Testbarkeit der erstellten Klassen automatisch gegeben. Fehler bei einer Programmänderung werden durch die bestehenden Testfälle auch sofort gefunden und ausgebessert.
Die Nachteile der Test Driven Development Methode werden in den folgenden Zeilen erläutert [GW03]:
Mangelndes Design ist bei dieser Vorgangsweise oft zu finden, weil die Architektur von der Vorstellungskraft des Programmierers begrenzt ist. Arbeiten am Design sind im Prozess nicht vorgesehen. Dies induziert ein hohes Projektrisiko, allerdings kann, wenn die Methode einwandfrei funktioniert, auch sehr viel Entwicklungszeit und somit Geld gespart werden.
Die Grenzen zwischen Designerstellung, Implementierung und dem Testen werden unklar, was wiederum zu einer klaren Verfehlung des Gesamtziels führen kann, weil sich die testgetriebene Entwicklung mehr auf die Erstellung der einzelnen Elemente als auf die logische Struktur des Programmaufbaus konzentriert.
Manche Tests, wie beispielsweise die Überprüfung einer graphischer Benutzeroberfläche, sind äußerst schwierig zu realisieren. Außerdem müssen für viele Tests so genannte Dummy Objekte erstellt werden, welche als Hilfsmittel zur Überprüfung von Modulen mit Abhängigkeiten zu

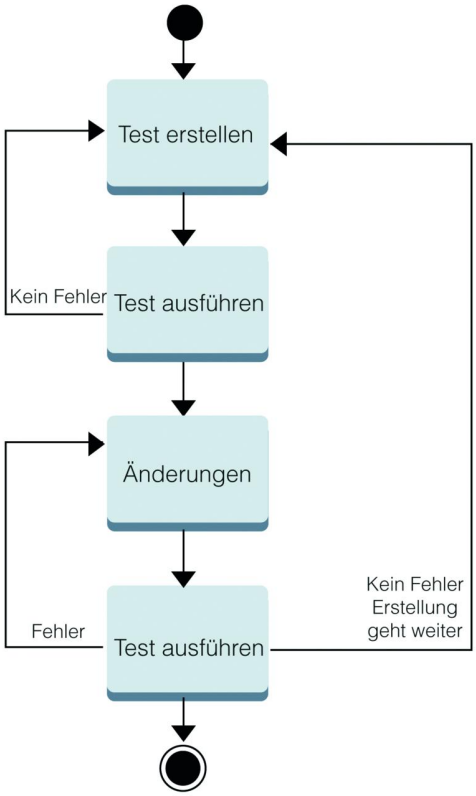

Abbildung 2.25: Test Driven Development - Ablauf (nach [Bec02])

nicht implementierten Klassen dienen. Dies bringt einen erheblichen Entwicklungsaufwand mit sich. Wichtige Entscheidungen werden im Test Driven Development ebenfalls nicht dokumentiert und können in Vergessenheit geraten.
Die testgetriebene Entwicklung stützt sich auf permanente Änderungen des Programmcodes, diese können in großen Software-Projekten aber die Komplexität erheblich steigern.
Schwer-testbarer Code erfordert viel Erfahrung und setzt hohe Ansprüche an die Fähigkeiten der Programmierer. Einige Personen im Team sind möglicherweise diesen Anforderungen nicht gewachsen und als Resultat werden Testfälle einfach ausgelassen.

KAPITEL 3

Related Work

Diese Arbeit inkludiert auch eine Recherche nach bestehenden Testkonzepten einer Service-orientierten Architektur. Die Ergebnisse werden in diesem Kapitel zusammengefasst und auf ihre Relevanz für den eigenen Entwicklungsprozess untersucht.

3.1 Analyse bestehender Testkonzepte für Service-orientierte Architekturen

SOA Test Methodik

Beim Testen einer SOA kommt es nicht nur auf das Überprüfen der Funktionalität und der Performance an, auch das Testen der Schnittstellen und der einzelnen Services, welche die Erstellung vieler verschiedener heterogener Systemlandschaften ermöglichen, sind von großer Bedeutung. Andere Aspekte, wie Latenz oder Sicherheit sollten ebenfalls berücksichtigt werden. Ein Business-Prozess kann aus vielen unabhängigen Services bestehen, wodurch Integrationstests zwischen den Services eine wesentliche Rolle spielen. Der Begriff SOA Governance ist für das Testen auch von großer Bedeutung: Es wird überprüft ob jedes einzelne Service während der gesamten Einsatzzeit der unternehmenseigenen Richtlinie und dessen festgesetzten Standards entspricht. Dies erhöht die Flexibilität, die Implementierungsgeschwindigkeit, die Qualität, die Konsistenz und die Kommunikation der Services enorm. [Har07].
Die Implementierung einer Service-orientierten Architektur durchläuft die Phasen Anforderungsanalyse, Festlegung des Designs und Implementierung. [Har07] schlägt für die Umsetzung das bereits in den Grundlagen detailliert erläuterte V-Modell vor, weil es das Projektteam laufend an das Testen jedes Services & der gesamten SOA erinnert. Des Weiteren verfolgt es einen Top-Down Ansatz für die Definition der Anforderungen des Business Prozesses und der Festlegung der funktionalen und technischen Architektur. Nach der Implementierung wird dann mit Hilfe eines Bottom-up Ansatzes zuerst ein einzelnes Service getestet, dann das Zusammenspiel mehrerer Services und am Ende der komplette Businessprozess. Tests werden während des gesamten Software-Entwicklungsprozesses auf unterschiedlichen Leveln durchgeführt.

Einige Herausforderungen beim Testen Service-orientierter Architekturen im Vergleich zu klassischer Software lassen sich wie folgt definieren [Har07]:

- Es gibt zur normalen Funktionalität noch eine zusätzliche Business-Logik, die getestet werden muss.
- Services existieren auch außerhalb des Unternehmens.
- Die Qualität der einzelnen Services muss für eine potentielle Wiederverwendung sehr hoch sein, es ist mit einem wesentlich höheren Testaufwand zu rechnen.
- SOA basiert auf heterogenen Technologien.
- Standards wie WSDL oder SOAP als Teil einer Service-orientierten Architektur sind nur ein kleines Fragment des zu testenden Bereiches.

Um mit diesen Herausforderungen fertig zu werden muss das Test-Team ein hohes technisches Know-How mitbringen (verwendete Technologien, Netzwerksicherheit, Latenz, ...). Darüber hinaus werden einige zusätzliche Tools gebraucht um SOA Governance, Qualität, Konfiguration und die Sicherheit zu gewährleisten. Einzelne Testteams müssen unabhängig von der eingesetzten Technologie einzelnen Businessprozessen zugeordnet werden um durch agile Entwicklung nicht die Qualität zu beeinträchtigen. Unternehmen müssen auch in die Entwicklung und die Wartung von Tests für die wichtigsten Services investieren um eine gute Performance und die Sicherheit für die gesamte Service-orientierte Architektur zu garantieren [Har07].
Das Testen einer Service-orientierten Architektur wird durch das Aufspalten in kleine Artefakte realisiert: die SOA wird in verschiedene Bereiche aufgeteilt und separat getestet: Die Abbildung 3.1 zeigt die einzelnen Komponenten mit ihren spezifischen Testlevels. Die Testlevels lassen sich wie folgt zusammenfassen [Har07]:

- Governance Tests: Während des gesamten Projektes wird überprüft ob die vom Unternehmen definierten Richtlinien eingehalten werden.
- Service-Komponenten-Level Tests: Dies entspricht normalem Unit-Testing, einzelne Module werden, idealerweise mit Hilfe zuvor spezifizierter Qualitätskriterien und einer "Pair Reviewing" Strategie, auf korrekte Funktionalität überprüft.
- Service-Level Tests: Ein ganzes Service wird auf Funktionalität, Sicherheit, Performance und auf korrekte Implementierung der Anforderungen mit Hilfe von Test-Tools (auch automatisiert) überprüft. Code-Reviews sollten ebenfalls mit einer "Pair Reviewing" Strategie behandelt werden und die Qualitätskriterien müssen auch klar definiert sein.
- Integration-Level Tests: Die Interfaces werden getestet, hierbei wird auf die korrekte Umsetzung der Schnittstellen-Definition geachtet.
- Prozess & Orchestrierungs-Level Tests: Die Funktionalität des Zusammenspiels einzelner Services wird getestet und die Business-Logik wird hierbei ebenfalls überprüft.

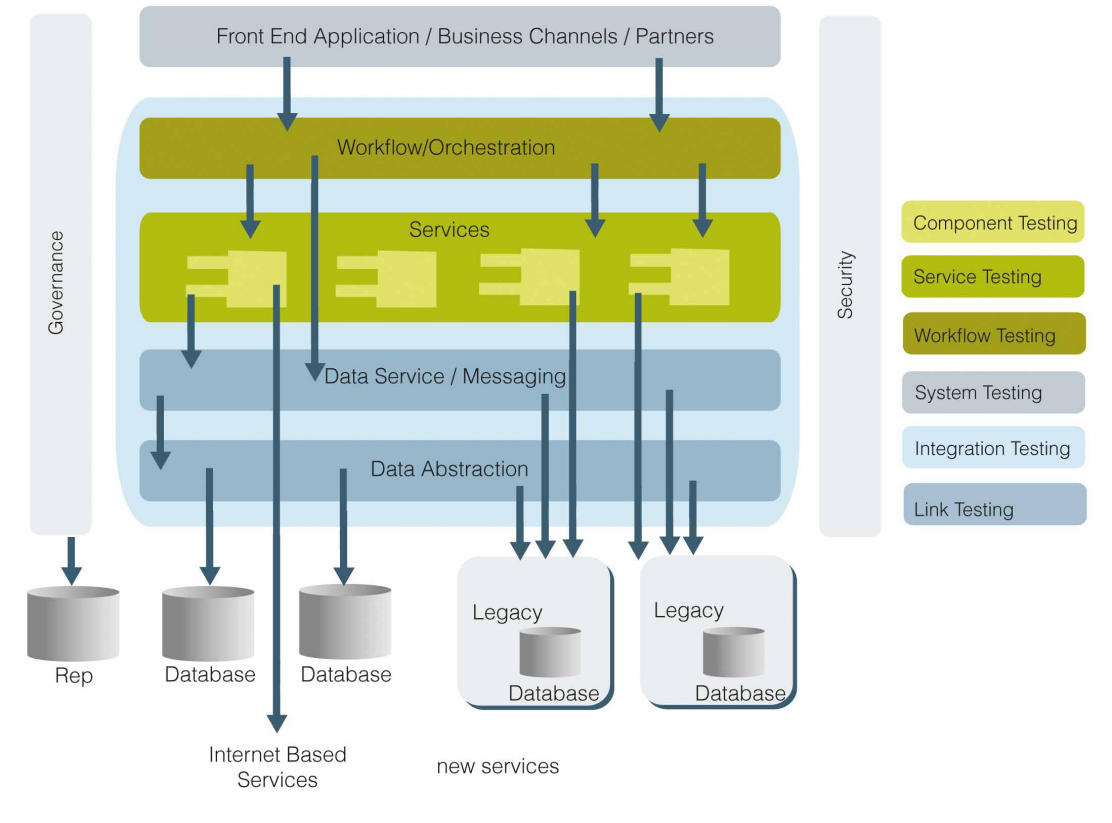

Abbildung 3.1: SOA Komponenten inkl. Test-Levels (nach [Har07])

- System-Level Tests: Die Erfüllung der zuvor definierten Business Anforderungen wird überprüft. Diese Tests werden in Zusammenarbeit mit den endgültigen Benutzern durchgeführt.
- Sicherheits Tests: Während der gesamten Projektlaufzeit müssen Sicherheitstests durchgeführt werden, da dies auch gesetzlich (zum Beispiel im Sarbanes-Oxley-Act) gefordert wird. Die Business-Anforderungen werden um Sicherheitsaspekte erweitert. Eine Risikoidentifikation inklusive -beurteilung und ein Review bezüglich der Einhaltung definierter Sicherheitsstandards sind durchzuführen. Geplante periodische Penetrationstests sollen Schwachstellen in der Service-orientierten Architektur aufzeigen.

Abschließend werden noch die verschiedenen Testphasen mit ihren durchzuführenden Testtypen definiert [Har07]:

Komponenten-Level-Test Phase: In dieser Phase sollte die Funktionalität, Performance, Interoperabilität, Rückwärtskompabilität, Compliance zu den Unternehmensrichtlinien und die Security mit Hilfe der Dokumentation des technischen Designs und der Programmspezifikation

getestet werden.

Service-Level-Test Phase: In dieser Phase sollte die Funktionalität, Performance, Compliance zu den Unternehmensrichtlinien und die Security mit Hilfe der Dokumentation der Business Anforderungen, des technischen Designs und der Unternehmensrichtlinien überprüft werden.

Integrations- und Orchestrations-Test Phase: In dieser Phase sollte die Funktionalität, Interoperabilität, Rückwärtskompabilität und die Security mit Hilfe der Dokumentation der Business Anforderungen, des technischen Designs und der Unternehmensrichtlinien validiert werden.

System Test Phase: In dieser Phase sollte die Funktionalität, Performance, Interoperabilität, Rückwärtskompabilität, Compliance zu den Unternehmensrichtlinien und die Security mit Hilfe der Dokumentation der Business Anforderungen, des technischen Designs und der Unternehmensrichtlinien analysiert werden.

Zum Überprüfen der Funktionalität wird mittels Black Box Testtechnik bestimmt ob eine Komponente, ein Service oder ein ganzes System wie spezifiziert arbeitet. Die Business Anforderungen und das technische Design werden hierbei für das Testdesign verwendet.
Die Performance eines Service/Komponente kann mit Hilfe von einigen SOA Test-Hilfswerkzeugen festgestellt werden. Es soll nicht nur das ganze System gemessen, sondern die Leistungsfähigkeit einzelner Services ebenfalls dokumentiert werden.
Die Sicherheitstest wurden bei den Testlevels bereits definiert und werden nicht nochmals beschrieben.
Service-Interoperabilität wird entweder durch striktes Implementieren einzig und alleine von veröffentlichten Interface-Standards oder durch die Verwendung eines Broker Service, der die Daten für das andere System zur Verfügung stellt, erreicht. Die Interoperabilität von SOA muss auch während der Laufzeit getestet werden.
Die Rückwärtskompabilität testet ob Änderungen bei einer Schnittstelle auf vorhandene Service-Konsumenten dieses Interfaces Auswirkungen hat. Jede Änderungen an Interfaces müssen diesem Test unterzogen werden.
Der Compliance Test überprüft ob die SOA Implementierung den Unternehmensstandards und -richtlinien genügt und muss während der gesamten Projektlaufzeit immer wieder durchgeführt werden. Ob die Software nicht gegen gesetzliche Bestimmungen verstößt wird ebenfalls durch diese Tests überprüft.

SOA Testing Governance

Der Begriff SOA Testing Governance beschreibt ein Konzept, welches gemeinsame Tests der Service-orientierten Architektur Umgebung empfiehlt. Eine SOA ist unabhängig von Organisation und Technologie, was das Testen allgemein erschwert, weil im Gegensatz zum klassischen Testen von Software der ausführende Tester keinen direkten Zugang zum System hat, die einzelnen Fragmente können während des Tests weder eingesehen noch verändert werden [BP09].
Bei Service-orientierten Architekturen können einzelne Teile der Software von anderen Unternehmen verwaltet werden, d.h. der Zugriff ist beschränkt oder es stehen nur Teile der Tes-

tumgebung zur Verfügung. Die SOA Testing Governance empfiehlt alle Beteiligten in den Testprozess einzubinden, es werden zu verwendende Standards und gemeinsame Regeln festgelegt, an die sich alle in den Vorgang inkludierten Organisationen halten müssen. [BP09].
Das Konzept der "SOA Testing Governance" wird als wichtiger organisatorischer Aspekt im eigenen Softwareentwicklungsprozess berücksichtigt.

Testen von Mobile Devices

Die hohe Rechenleistung und die Kommunikationsmöglichkeiten (Wireless-Lan, UMTS/LTE, Bluetooth, ...) von Smartphones machen diese Geräte auch für die Verwendung in verteilten Systemen interessant. Beispielsweise können (in Hinblick auf eine Service orientierte Architektur) Dienste auf diesen Smartphones ausgeführt werden.
Das Testen von Mobile Devices wie beispielsweise einem Mobiltelefon mit Betriebssystem Android gestaltet sich ebenfalls herausfordernd: Debugging und Testen ist auf vielen Geräten nur beschränkt realisierbar. Mobilfunkapplikationen haben bestimmte Anforderungen an die Ausführungszeit, unterschiedliche Verbindungsarten wie Wireless-Lan, 3G/HSDPA und Bluetooth steigern die Komplexität der Testumgebung weiter. Auch die Akkubelastung ist bei Tests von mobilen Geräten mit einzurechnen. Eine mögliche Lösung ist die Einführung einer zusätzlichen Abstraktionsebene zwischen den Test Cases und deren Ausführung [ZfXp09].
Die Ausführung von Tests auf mobilen Geräten soll ebenfalls im Softwareentwicklungsprozess berücksichtigt werden. Die Einführung eines leichtgewichtigen standardisierten Interfaces soll das Monitoring auf beliebigen Betriebsumgebungen ermöglichen.

Integration von White- und Blackbox Testen

Die Kombination von White- und Blackbox-Tests in einem Projekt bringt einige Vorteile: Dem Testleiter reicht es ein einziges Testkonzept zu kennen, die Wartung wird durch Zentralisierung somit stark vereinfacht. [BGS01] stellt eine Lösung mittels graphischer Repräsentation der Tests vor: Jede implementierte Klasse wird aus einer Spezifikationssicht und einer Implementationssicht betrachtet, welche mit Kontrollflussgraphen visualisiert werden. Sie ermöglichen ein Testen mit strukturbasierten Testmethoden. Dieses Konzept sieht keine Implementierung auf Basis der Service-orientierten Architektur vor, nur klassische Software soll getestet werden.
Das Konzept ist trotzdem für diese Arbeit relevant, weil es das Ziel verfolgt das komplexe Testen von Software zu vereinfachen und zu zentralisieren. Dieser wichtige Ansatz wird auch bei der Erstellung des in dieser Arbeit präsentierten Softwareentwicklungsprozess berücksichtigt und umgesetzt.

Reflektives Architektur-basiertes Software Test Management Modell

Eine Eigenschaft von Service-orientierten Architekturen ist die wachsende Komplexität, die sich durch immer größer werdende Netzwerke heterogener Natur erklären lässt. Das in [JfSJb+06] vorgestellte "Reflective Architecture Based Software Testing Management Model" (abgekürzt: RATMM) vereint die Theorie der Reflektion mit Methoden des Software-Architektur Design und des Managements von Softwaretests. Der Prozess des Software-Testens wird durch die

Verwendung von Meta-Informationen über die verwendete Umgebung gestützt und der Test einzelner Komponenten wird forciert. Im Gegensatz zum V-Modell wird hier die Annahme getroffen, dass die einzelnen Testphasen nicht der Reihe nach abgearbeitet werden können, sondern das Testen während der Implementierungsphase jederzeit begleitend durchgeführt werden muss. Die Umgebung wird in ein Basis- und ein Metalevel geteilt. Das letztere Level enthält Meta-Informationen um ein effizientes Monitoring für die Basis zu schaffen (einem Basisobjekt ist immer genau ein Metaobjekt mit zusätzlichen Daten zugeordnet).
Die Notwendigkeit eines Monitoring-Systems für ein effektives Testmodell für heterogene Systemlandschaften wurde erkannt und wird in den eigenen Softwareentwicklungsprozess inkludiert. Zusätzliche Informationen über den Zustand eines Service werden ebenfalls beim Entwurf berücksichtigt.

BPA Simulierte Ereignis Proxies

Ein vorhandenes Testframework für Service-orientierte Architekturen stellt das Business Process Activity Simulated Event Proxies System dar. Die Umgebung wird in drei Layer unterteilt: einem Business Process Layer, einen Service und einen Computing Resource Layer. Einige Aktivitäten werden simuliert um über den Business Process Layer (funktioniert dieser, ist anzunehmen, dass auch die darunter liegenden Layer korrekt arbeiten) den Geschäftsprozess testen zu können. Diese Simulation hilft bei der Implementierung durch die Schaffung der Möglichkeit wiederholter Tests bereits während der Entstehung [Lee09].
Das in dieser Arbeit beschriebene Prozessmodell unterstützt die Entwicklungsarbeit ebenfalls mit häufigen Testiterationen, jedoch werden durch die testgetriebene Entwicklung der einzelnen Services mit anschließenden Integrations- und Business-Requirement Abnahmetests alle Layer berücksichtigt.

Fortlaufende Integrationstests von Service-orientierten Architekturen

Ein weiteres Framework stellt das Konzept des "Continuous Integration Testing" (CIT) zur Verbesserung der Qualität von Service-orientierten Architekturen vor. Nach jeder Implementierung einer neuer Komponente wird ein Integrationstest durchlaufen, der die Akzeptanz der neuen Komponente in der Umgebung sicher stellen soll. Hier existieren ebenfalls zwei verschiedene Layer: Der Behaviour Layer ist unabhängig von der Plattform und beinhaltet die Logik eines Testcases. Die umgebungsspezifischen Daten sind im Configuration Layer gespeichert, welcher eine Anpassung an unterschiedliche Programmierungsmodelle ermöglicht. Die Problematik verschiedener Aufrufmechanismen für verschiedene Technologien wird somit beseitigt [LLZ+09].
Durch den Einsatz von standardisierten Schnittstellen, spezifiziert durch das Prozessmodell dieser Diplomarbeit, sollen ebenfalls bestimmte Technologien wie Web-Services oder Enterprise Java Beans gemeinsam ohne große Anpassungen der Testumgebung verwendet werden.

Architektur für Monitoring von Laufzeitumgebungen

In einer Laufzeitumgebung unterliegen Webservices ständigen Veränderungen, die beispielsweise durch die Einbindung von externen Webservices, welche die Funktionalität der bereits laufenden Services beeinträchtigen kann, verursacht werden. Wenn zusätzlich noch Daten über den aktuellen Zustand vom Service gespeichert werden, ist die Implementierung einer leicht skalierbaren Umgebung für Monitoring und Validierung noch schwieriger. Sie wird mittels Ausführung unterschiedlicher Testfallsequenzen, welche auf realistischen Beispieldaten basieren, realisiert [BDS10].
Änderungen, die durch äußere Einflüsse verursacht wurden, sollen bei der Validierung erkannt werden, dies wird ebenfalls im Prozessmodell berücksichtigt.

Testen mit Hilfe von negativen Testcases

Testcases, die bewusst Fehler produzieren, werden negative Testfälle genannt. Es werden bei diesem Lösungsansatz nicht alle Services validiert, sondern fehlerhafte Services, die nicht gemäß der Spezifikation arbeiten, ausgeschlossen. Dies geschieht durch automatisiertes Testen. Schlägt eine gewisse Anzahl von Tests fehl, wird das Service ausgeschlossen und dem Prozess diese Informationen mitgeteilt [ZQ06].
Dieser Ansatz soll ebenfalls beim Abnahmetest im eigenen Entwicklungsprozessmodell Verwendung finden.

Testwerkzeuge für SOA

In diesem Kapitel werden einige unterstützende Tools für das Testen von Service-orientierten Architekturen vorgestellt.

- PushToTest Testmaker [Pus]: Dieses Tool hilft beim Finden von Performance Bottlenecks und funktionellen Problemen bei Web-Anwendungen, Rich Internet Applikationen (Ajax, Flex, Flash), Service-orientierte Architekturen und Business Prozess Management Anwendungen. Des Weiteren kann die Skalierbarkeit der Anwendung gemessen werden.
- Borland SilkPerformer [Bor]: Dieses Tool ermöglicht automatisierte Performance- und Funktionstest von Web Services und Interoperabilitätstests zwischen einzelnen Services.
- Eviware soapUI [Evi]: Einfache Erstellung und Ausführung automatisierter Funktions-, Regressions-, Sicherheits- und Load-Tests für SOAP und REST-basierte Web-Services, Java Messaging Service Enterprise Layers und Rich Internet Anwendungen. Es können auch Services simuliert werden um frühzeitig Integrationstests durchführen zu können.
- Greenhat GH Tester [Gre]: Eine grafische Anwendung zum Testen von Message-basierten Systemen (JMS, SOAP, und weitere).
- Parasoft SOAtest [Par]: Eine automatisierte Web-Testing Lösung, die wesentliche Aspekte des SOA-Testens implementiert. Die Interoperabilität, die Sicherheit und die Skalier-

barkeit der Service-orientierten Architektur können beispielsweise vergleichsweise einfach ermittelt werden.

3.2 Analyse bestehender Schnittstellentests für SOA

In weiterer Recherche wurde nach bestehenden Lösungen für das Monitoring und Reporting von Interfaces bei Service-orientierten Architekturen gesucht. Nützliche Konzepte wurden identifiziert, die in den folgenden Sektionen vorgestellt werden.
Im folgenden Kapitel wird auf den "seekda!" Suchdienst eingegangen, welcher öffentlich erreichbare Web-Services im Internet anhand ihrer WSDL Definitionen testet und anschließend "Quality of Service" Informationen zur Verfügung stellt. Außerdem wird der WSDL Standard, der ebenfalls validiert werden kann, als Mittel zur Überprüfung betrachtet. Abschließend wird das Konzept "jUnit" zur Überprüfung von Schnittstellen analysiert.

seekda! Suchdienst

Der "seekda" Suchdienst stellt einen direkten Zugang zu einer Vielzahl öffentlich verfügbarer Web-Services, welche den WSDL-Standard [Eri] oder Web-APIs (wie REST) nutzen, zur Verfügung. Diese Dienste können mit Hilfe der bereitgestellten Schnittstellen als neue Komponenten in bestehende Software-Architekturen eingefügt werden [BSS11].
"Seekda" durchsucht das Web nach verwendbaren Web-Services und speichert neben den Inhalten auch RDF (Resource Description Framework, eine Sammlung von W3C Standards zur Objektschreibung) - Metadaten über die Services. Ein Tool namens "Seekda! Enriched Services" sammelt automatisch WSDL- und REST-basierte Informationen über die öffentlich zugänglichen Web-Services, des Weiteren können die Benutzer manuelle Beschreibungen dieser Dienste hinzufügen [BSS11].
Der "Seekda!" Suchvorgang muss bei der Verifikation auch überprüfen ob der gefundene Dienst wirklich einem verwendbaren Web-Service entspricht. Hierbei spielt die manuelle Bewertung durch die Benutzer eine wichtige Rolle [BSS11].
Der Annotationsprozess gliedert sich in einen generischen und je nach gefundener Technologie WSDL-spezifischen oder Web-API-spezifischen Teil. Diese werden nachstehend erläutert [BSS11]:

- **Generische Merkmale:** Die Links zu Service-Beschreibungen und weiterführenden Informationen werden über das Annotationstools bereitgestellt und die Informationen im RDF-Standard exportiert. Bestehende Annotationen von Benutzern können eingesehen werden und ein Assistent hilft bei der manuellen Bewertung der Webservices.

- **WSDL-basierte Dienste:** Das WDSL-Abstract und eine Liste von zugehörigen Dokumenten wird angezeigt. Ein Link zum Provider des Services sowie verfügbare Detailinformationen werden präsentiert (beispielsweise die Service Kategorie, gratis oder Preisinformationen). Auch Ratings von anderen Benutzern können bewertet werden.

- **Web-APIs:** Die Annotation von Web Application Programming Interfaces stellt eine große Herausforderung dar, da die Unterscheidung zwischen einer reinen HTML-Seite und einer

Beschreibung eines Web-API schwer automatisch erkennbar ist. Benutzer müssen die Suchergebnisse bewerten um die gefundenen Ergebnisse zu validieren.

Das "Seekda Enriched Services" Tool verwendet folgende Systemarchitektur für den Umgang mit einer großen Datenmenge [BSS11]:

- Java-Server-Pages (JSP) Frontend: für das web-basierte User Interface
- INSEMTIVES Services: Annotationsdienst & Dienst zur Einordnung in Kategorien & Dienst für das Hinzufügen von Tags & Dienst für die semantische Suche
- Services-Index mit allen gefundenen Informationen gespeichert mit Hilfe des RDF-Standards unter Benutzung verschiedener definierter Ontologien (für Detailinformationen über die Ontologien siehe [BSS11])
- Komprimierte Archivdateien: Physische Repräsentation der gefundenen Dokumente durch den Dienst für die semantische Suche

Wie man am "Seekda" Beispiel sehen kann, ist WSDL als Standard für die Bewertung von Web-Services geeignet. Außerdem kann die Web-Service Description Language validiert werden, dies entspricht somit im weiteren Sinne einer automatischen Überprüfung von Schnittstellen der Web Services und ist ein Grund zur näheren Betrachtung dieses Konzeptes in der nachfolgenden Sektion.

WSDL

Die "Web-Services Description Language" (kurz: WSDL) beschreibt netzwerkfähige Dienste als eine Gruppe von Endpunkten, welche mit Hilfe von Nachrichten miteinander kommunizieren, und nutzt dabei die "Extended Markup-Language" (XML). Die "Messages" werden abstrahiert beschrieben und einem konkreten Netzwerkprotokoll sowie Nachrichtenformat zugeordnet um einen Endpunkt zu definieren. Mehrere konkrete Endpunkte werden zu abstrakten Endpunkten (Services) zusammengefasst. Die WSDL lässt sich um eine Beschreibung der Endpunkte und der Nachrichten (unabhängig vom Netzwerkprotokoll und Nachrichtenformat) erweitern, die W3C-WSDL-Referenz beschreibt allerdings nur eine Nutzung in Kombination mit HTTP GET und POST, SOAP in Version 1.1 und dem MIME-Datenformat [Eri].
Nachdem Kommunikationsprotokolle und Nachrichtenformate zunehmend standardisiert im Web zu finden sind, wird die strukturierte Beschreibung der Kommunikation immer wichtiger. WSDL bietet Service-Definitionen für verteilte Systeme und kann die Details der Kommunikation zwischen Systemen automatisieren [Eri].
Durch die abstrakte Definition von Endpunkten und Nachrichten, unabhängig von eingesetzten Technologien, können diese Definitionen wiederverwendet werden. Ein WSDL-Dokument nutzt folgende Elemente zur Spezifikation von netzwerkfähigen Diensten [Eri]:

- **Typen:** ein Container für Datentypdefinitionen, welche mit Hilfe des "XML-Schema Definition" (XSD) beschrieben werden
- **Nachrichten:** eine abstrakte Definition der gesendeten Daten

- **Aktion:** eine abstrakte Beschreibung der Operation, welche vom Service unterstützt wird
- **Port-Typ:** eine abstrakte Menge von Aktionen, welche von einem oder mehreren Endpunkten unterstützt werden
- **Bindung:** eine konkrete Kommunikationsprotokoll- und Datenformatspezifikation für einen bestimmten Porttyp
- **Port:** ein einzelner Endpunkt wird durch die Bindung und den Porttyp definiert
- **Service:** eine Sammlung von zusammenhängenden Endpunkten

Neue Typdefinitionen werden in WSDL mit Hilfe der "XML-Schema Specifikation" (XSD) umgesetzt. Durch den Umstand, dass in einem großen System eine Typdefinition nicht ausreicht, unterstützt WSDL auch ein Konzept für Erweiterungen [Eri].
Zusätzlich definiert WSDL einen einheitlichen Mechanismus für die Bindung von einem spezifischen Protokoll oder von einem Datenformat an eine abstrakte Nachricht, eine Aktion oder einen Endpunkt. Abstrakte Definitionen können hierbei wiederverwendet werden. Wie bereits am Anfang dieses Kapitels angedeutet, enthält das Service Definitionsframework auch eine Unterstützung für die Erweiterung der Bindung auf folgende Protokolle und Nachrichtenformate: SOAP 1.1, HTTP GET/POST, MIME [Eri].
Für weitere Informationen über die Verwendung der genannten Protokolle und Nachrichtenformate sowie einem praktischen Anwendungsbeispiel sei auf die W3C WSDL Spezifikation ([Eri]) verwiesen.
Die Validierung von WSDL garantiert, dass das eigene Dokument dem WSDL Standard entspricht. Diese Analyse kann beispielsweise mit dem Eclipse Plugin "WTP's" durchgeführt werden. Wie bereits im "Seekda" Abschnitt erwähnt, kann diese Überprüfung auch für den Test der Schnittstellendefinitionen des Web Services verwendet werden. Aus dem validierten WSDL Dokument können automatisch Teile der Software generiert werden [BL06]. Die Validierung unterstützt die Qualitätssicherung der eigenen Entwicklung.

jUnit-Tests

Das Test-Framework jUnit, welches die Erstellung und Ausführung von Testfällen zur Überprüfung der korrekten Funktionalität von Java-Klassen ermöglicht, wurde ebenfalls auf seine Einsatzfähigkeit zum Testen einer Service-orientierten Architektur und ihrer Schnittstellen untersucht. Die Ergebnisse werden in den folgenden Absätzen präsentiert.
Einzelne Komponenten (in der Programmiersprache Java entspricht dies dem Klassenkonzept) können einfach mit der Erstellung von spezifischen Testfällen überprüft werden. Abhängigkeiten zu anderen Klassen werden mit "Dummy-Objekten" (siehe Kapitel "Testgetriebene Entwicklung" in den Grundlagen) simuliert [MWW05].
Die Verifikation einzelner Komponenten einer Service-orientierten Architektur garantiert keinesfalls die korrekte Funktionalität des gesamten Systems beziehungsweise einzelner Services. Integrationstests mehrerer Komponenten können mit jUnit auch umgesetzt werden, in der abhängigen Klasse wird der Konstruktor zur Erzeugung eines Test-Objektes aufgerufen (statt des

Dummy-Objekts wie beim Komponententest). Bei einer großen Anzahl von Klassen müssen jedoch unzählige Testfälle erstellt werden um die korrekte Kollaboration der Komponenten garantieren zu können [MWW05].
Interessant ist die Überprüfung von "Use-Cases" mittels jUnit-Tests, da damit der funktionale Systementwurf (unter der Annahme vollständiger Dokumentation) validiert werden kann [MWW05].
Zusammengefasst kann man sagen, dass das jUnit-Testframework sich gut zur Überprüfung von Komponenten und des funktionalen Systementwurfs eignet. Um die Komplexität der Integrationstests von Komponenten und der Services sowie deren Schnittstellen bei einer SOA kontrollieren zu können, soll im nächsten Kapitel ein neues Konzept vorgestellt werden.

KAPITEL 4

Prozess zur Qualitätssicherung von Schnittstellen

Dieses Kapitel definiert einen Softwareentwicklungsprozess, der die Qualität von heterogenen Systemlandschaften, mit besonderem Fokus auf Service-orientierte Architekturen, erheblich verbessern soll. Monitoring und Reporting von Interfaces in SOA sollen ein fester Bestandteil der Testumgebung werden, weil somit die Fehleridentifizierung und -behebung wesentlich leichter umgesetzt werden kann. Weitere Vorteile sind effiziente Feststellung der Performance des Systems und durch die testgetriebene Entwicklung kann von einer schnellen, aber dennoch qualitativ hochwertigen Implementierung ausgegangen werden.
Im ersten Kapitel folgt eine Definition des Sensorenkonzeptes, welches vom Entwicklungsprozess für das Monitoring der Betriebs- und Testumgebung der Service-orientierten Architektur verwendet wird.
Im folgenden Abschnitt werden zuerst die qualitativen Anforderungen an den Softwareentwicklungsprozess definiert um eine Konkretisierung der Zielsetzung zu erreichen und ein möglichst effektives Modell zum Testen von heterogenen Systemlandschaften entwickeln zu können. Es werden hierbei die im Kapitel Related Work erfassten Anforderungen explizit berücksichtigt.
Außerdem werden qualitätsfördernde Maßnahmen dokumentiert, die im zu realisierenden Modell inklusive Prozessbeschreibung den reibungslosen Projektablauf unterstützen sollen. Dabei wird zwischen konzeptuellen und technischen Qualitätssicherungsmaßnahmen unterschieden.
Anschließend wird der Softwareentwicklungsprozess mit seinen einzelnen Phasen und Schritten der Vorgehensweise eingeführt und im kleinsten Detail beschrieben: Ein neues Modell wird vorgestellt und graphisch repräsentiert um ein einfaches Bild der Entwicklungsmethode zu schaffen und ein intuitives Verständnis für die von diesem Prozess empfohlene Vorgangsweise zu vermitteln. In der ersten Phase folgt eine Beschreibung des Ablaufs, wie Business Requirements ermittelt, analysiert und dokumentiert werden müssen. Danach wird ein möglicher Ablauf zur Definition des funktionalen Systementwurfs des zu implementierenden Businessprozesses vorgeschlagen. Der darauf folgende technische Entwurf gibt einen Überblick über die einzelnen Services und deren Zusammenspiel, auch hierfür wird dem Leser eine konkrete Vorgehensweise

näher gebracht. Die Definition und die Überprüfung der Schnittstellen wird im vorgestellten Softwareentwicklungsprozess in einem eigenen Schritt behandelt: Das Konzept von Sensoren wird für jedes Service eingeführt, auf welches im entsprechenden Kapitel noch detailliert Bezug genommen wird. Ein Konzept zur exakten Definition der einzelnen Services/Komponenten wird präsentiert und in der zweiten Phase die Implementierung mit Hilfe der Methode der testgetriebenen Entwicklung für die Services, Interfaces und Sensoren behandelt. Die dritte Phase beschreibt verschiedene Testkonzepte und die Durchführung von notwendigen Änderungen im Programmcode während einer Iteration im vorgestellten Prozessmodell dieser Arbeit.
Des Weiteren wird in dieser Sektion eine mögliche Architektur für ein das Prozessmodell unterstützendes Framework vorgestellt, die Bestandteile graphisch veranschaulicht und die Zusammenhänge beschrieben. Diese Architektur soll als Basis für eine Referenzimplementierung des Frameworks in einer anderen Diplomarbeit dienen.
Um effizientes Reporting zu ermöglichen, wird in diesem Abschnitt auch eine Auswahl möglicher Metriken zur Qualitätssicherung und -verbesserung aufgezeigt.
Den Abschluss dieses Kapitels bilden eine Aufzählung von möglichen Beispielen für die Einsatzfähigkeit des im Laufe dieses Abschnitts vorgestellten Sensorenkonzeptes und darauf aufbauend eine Machbarkeitsstudie anhand einer Analyse des WASI-Modells für die Praxis.

4.1 Definition des Sensorenkonzeptes

Bevor die Anforderungen und die Qualitätsmaßnahmen für den zu entwickelnden Softwareprozess ermittelt werden können, muss das Konzept des Sensors präsentiert werden.
Ein Sensor dient dem Monitoring eines Services und soll sich in jeder Sprache realisieren lassen. Die Anpassung an die jeweiligen Bedürfnisse wird mit Hilfe einer Konfigurationsdatei (beispielsweise im XML-Format) gewährleistet. Standardisierte Interfaces sorgen für die Verbindung zwischen den Komponenten des Services und dem eigentlichen Sensor, welcher relevante Daten der Komponenten während der Ausführung überwacht und an einen Server zum effizienten Monitoring während der Laufzeit weiterschickt. Der Server wertet dann die Monitoring-Daten aus und generiert Reports, welche bei der Fehlersuche und der Steigerung der Qualität (Aufspüren von Bottlenecks, Messung der benötigten Zeit, u.v.m.) von großem Nutzen sein können.
Die Verwendung mehrerer verschiedener Sensoren für eine Komponente ist ebenfalls ohne großen Aufwand möglich. Wird allerdings ein Sensor von mehreren Komponenten verwendet, müssen die Monitoring-Daten den eindeutigen Namen der Komponente enthalten, damit die Identifikation gewährleistet ist. Dies ist bei der Spezifikation der Services, Komponenten und Sensoren zu berücksichtigen.
Der Einsatz von Sensoren wird im Kapitel "Architektur eines Frameworks" in einer Übersicht graphisch veranschaulicht und detailliert beschrieben.

4.2 Anforderungen an den Entwicklungsprozess

Durch detaillierte Recherche und auch mit Hilfe der ausführlichen Artikel, welche im Kapitel Related Work vorgestellt wurden, konnten die Anforderungen an einen Entwicklungsprozess zum Testen von Service-orientierten Architekturen, welcher einen besonderen Fokus auf das

Testen der Schnittstellen und das Monitoring aller bestehenden Services legt, identifiziert werden. Diese ermittelten Requirements wurden in einzelne Kategorien eingeordnet, sie werden in den nachfolgenden Unterkapiteln präsentiert. Manche Anforderungen sind mehreren Kategorien zugeordnet, weil sie für mehrere Bereiche entscheidend sind, jedoch werden die unterschiedlichen Perspektiven jedes Requirements berücksichtigt. Die Abbildung 4.1 zeigt eine kategorisierte Übersicht der wichtigsten Anforderungen an den Softwareentwicklungsprozess.

Abbildung 4.1: Identifizierte Anforderungen an den Entwicklungsprozess

Anforderungen an die Dokumentation

Dieser Abschnitt fasst alle erforderlichen Dokumente und alle Projektinitiierungsschritte zusammen, die für den Entwicklungsprozess essentiell sind.

Business Requirements Analyse inklusive Risikomanagement: Die Anforderungen des Busi-

nessbereichs, der abgebildet werden soll, müssen analysiert werden. Gleichzeitig ist eine Risikoanalyse und eine qualitative und quantitative Risikobeurteilung durchzuführen. Die ermittelten Projektrisiken müssen während der gesamten Projektlaufzeit überwacht werden.

Testplan: Der Testplan wird zur gleichen Zeit wie die Requirementsanalyse, das heißt zu Beginn des Projektstarts, erstellt.: Er enthält eine Dokumentation über die geplanten Testdaten, Testfälle und Testscripts inklusive einer zeitlichen Planung der Durchführung der Tests mit anschließender Auswertung der Log-Dateien. Ein weiterer wesentlicher Faktor ist die Auswahl der zu verwendenden Testwerkzeuge (Tools) für die Testexekution inklusive Automatisierung. Bei Service-orientierten Architekturen ist nicht nur die Implementierung einzelner Services zu testen, sondern die gesamte Business-Logik auf korrekte Funktionalität hinsichtlich der Spezifikation zu prüfen.

Zeitplan: Im Zeitplan werden alle Fertigstellungstermine für die zu implementierenden Komponenten und Services spezifiziert, die Durchführung der Tests definiert und bedeutende Schritte der Implementierung als Meilensteine festgelegt.

Dokumentation des Businessprozesses (Funktionaler Entwurf): Die relevante Businesslogik wird mit Hilfe der "Business Process Modeling Notation" (BPMN) abgebildet und ein detaillierter Entwurf des Businessprozesses geschaffen.

Dokumentation des Systementwurfs: Dieser Entwurf enthält alle zu erstellenden Komponenten, Services und Schnittstellen aus technischer Sicht. Er soll aus lose gekoppelten Services bestehen und die Erstellung darf technische Barrieren keinesfalls berücksichtigen. Hilfsmittel für die Dokumentation einzelner Services/Komponenten können UML-Klassendiagramme darstellen.

Dokumentation über Sensoren: Die Verwendung der Sensoren zum Monitoring jedes Services soll ausreichend dokumentiert werden. Dies soll durch UML-Anwendungsfalldiagramme unterstützt werden.

Konfigurationsdokumentation: Die verwendeten Technologien und Standards zur Implementierung des Businessprozesses sowie deren Zusammenspiel müssen nachvollziehbar festgelegt werden. Auch die unternehmensinternen Richtlinien für die SOA Test Governance werden hier beschrieben.

Reporting: Die Metriken zur Überwachung des laufenden Projektstatus werden dokumentiert und eine Zusammenfassung der Fehler gegeben. Ein Status Report gibt zusätzlich Auskunft über den aktuellen Projektstatus.

Dokumentation zu Projektabschluss: Am Ende des Projektes soll ein Dokument erstellt werden, welches die "Lessons Learned" zusammenfasst und potenzielle Services für eine Wiederverwendung vorschlägt.

Anforderungen an den Testprozess

Diese Sektion beschreibt die Anforderungen an den eigentlichen Testprozess: Die Fragestellungen welche Tests wann und wie zu implementieren sind werden in diesem Kapitel aufgeführt.

Testaktivitäten: Die Testaktivitäten sollen den in den Grundlagen beschriebenen PDCA (Plan - Do - Check - Act) Zyklus während der gesamten Projektlaufzeit wiederholt durchlaufen um die Qualität der Software garantieren zu können, ständige Wiederholung, Anpassung und Verfeinerung der Tests auf allen Ebenen sind erforderlich.

Sicherheitstests während der gesamten Projektlaufzeit: Der Entwicklungsprozess soll ebenfalls die Sicherheit der Software im Produktiveinsatz gewährleisten. Service-orientierte Architekturen speichern oft Daten, die absolutem Firmengeheimnis unterstehen (beispielsweise Kostenvoranschläge für Projekte), und arbeiten oft plattformübergreifend, was zusätzliche Herausforderungen an die Sicherheit mit sich bringt, weil ein effektives Management der Rechte inklusive deren Vergabe implementiert werden muss. Sicherheitstests während der Entwicklung sind notwendig und bei Fertigstellung der Implementierung müssen mit so genannten Security Penetration Tests die verbleibenden sicherheitsbedrohenden Schwachstellen gefunden werden.

SOA Test Governance: Die unternehmensinternen Standards und Richtlinien, welche für die Service-orientierte Architektur festgesetzt wurden, müssen zwingend eingehalten werden. SOA Governance Testing garantiert diese Anforderung und muss deshalb im Entwicklungsprozess dringend berücksichtigt werden.

Zentrierung auf Business mit Services, verschiedenste Technologien: Eine weitere Herausforderung stellt die Vielzahl von verschiedenen Technologien bei der Umsetzung einer Service-orientierten Architekturen dar. Der Fokus soll aber auf lose-gekoppelten Services, die sich zu unterschiedlichen Business-Prozessen kombinieren lassen, liegen und nicht auf den eingesetzten Technologien. Der Testvorgang muss alle vorhandenden Technologien abdecken und der Entwicklungsprozess muss dies explizit für den funktionalen und technischen Systementwurf definieren.

Service-, und Schnittstellentests: Service-orientierte Architekturen bestehen aus einzelnen Services, welche über (meist) standardisierte Interfaces miteinander verknüpft werden. Die Tests müssen die korrekte Funktionalität jedes einzelnen Services überprüfen und deren Zusammenspiel über die Schnittstellen garantieren (wird über Integrationstests realisiert).

Berücksichtigung externer Services: Dienste, die von außerhalb des eigenen firmeninternen Systems kommen, können nicht vollständig getestet werden. Viel mehr muss dann mittels Integrationstests (siehe Punkt weiter unten) die Zusammenarbeit garantiert werden.

Wiederverwendung von Services: Um eine Wiederverwendung der eingesetzten Dienste zu gewährleisten, müssen diese vermehrt getestet werden. Bei mangelhafter Qualität ist eine Wiederverwendung dieser Services wenig wahrscheinlich.

Service-Komponenten-Level Tests & Service-Level Tests: Sowohl die einzelnen Komponenten als auch die Services müssen getestet werden. Hierbei ist zu berücksichtigen, dass manche dieser Dienste mit graphischer Benutzeroberfläche und manche komplett ohne im System vorhanden sind, das heißt es müssen verschiedene Testmethoden implementiert werden. Des Weiteren reicht das einfache Testen von Web-Services (wie bspw. SOAP) nicht aus um die gesamte Architektur zu erfassen, weil SOAs um einiges komplexer aufgebaut sind (Nachrichten am Bus, Datenbanken, ...). Nach Implementierung aller Services können diese getestet werden, bei einer bestimmten Fehleranzahl ist von einer Verwendung/Wiederverwendung abzuraten.

Definition von Key-Services: So genannte Schlüsselservices müssen definiert werden, sie genießen eine besondere Priorität bezüglich laufender Tests während des Projektes, da sie mit hoher Wahrscheinlichkeit zur Wiederverwendung herangezogen werden. Zusätzlich muss die Qualität jedes einzelnen Services nach jeder Überprüfung dokumentiert werden.

Integrationstests: Der Entwicklungsprozess fordert häufige Integrationstests, welche das korrekte Zusammenspiel einzelner Services über die standardisierten Schnittstellen verifiziert, die klare Kommunikation zwischen einzelnen Services muss einwandfrei möglich sein.

Prozess- und Orchestrationstests: Die Prozess- und Orchestrationstests müssen ebenfalls fixer Bestandteil des Testprozesses sein. Hier werden Teile oder sogar die gesamte Service-orientierte Architektur einem Test unterzogen. Es muss besonders auf Fälle wie dem gleichzeitigen Zugriff auf gemeinsame Ressourcen geachtet werden, es darf in keinem Fall zu einem Deadlock kommen.

System-level testing: Funktions- und Performance-Tests der implementierten SOA müssen ebenfalls Bestandteil des Entwicklungsprozesses sein. Die gesamte Business-Logik muss einer Prüfung auf Korrektheit unterzogen werden.

Regressionstests: Die Ausbesserung von Fehlern kann neue Fehler im System produzieren. Um diesem Missstand entgegen zu wirken, werden für den Entwicklungsprozess so genannte Regressionstests gefordert. Diese testen die gesamte bestehende Software und analysieren die Fehleranzahl, welche nachher zwingend kleiner als zuvor sein muss. Dies wird am leichtesten mittels Testautomation erreicht.

Zentralisiertes Testen: Eine weitere Anforderung an den Testprozess stellt das Thema Zentralisierung dar. Das System soll von zentraler Stelle getestet, überwacht und analysiert werden können.

Automatisierung: Der Testprozess fordert eine Testautomatisierung auf Service-Level-Ebene um Regressionstests effizient realisieren und Fehler rasch zu finden und ausbessern zu können.

Versionierung: Das Prozessmodell fordert eine Dokumentation der Änderungen an Tests und empfiehlt den Einsatz eines Versionierungssystems um Änderungen an Testfällen besser zu überblicken/rekonstruieren zu können.

Auswahl der Testwerkzeuge: Eine weitere Anforderung stellt die Festlegung der zu verwendenden Testwerkzeuge dar, damit alle Mitarbeiter auch wirklich einen standardisiertes Testablauf einhalten müssen.

Aufspaltung der gesamten SOA in überschaubare Testeinheiten: Diese Anforderung ist bei diesem Testprozess implizit durch die Aufteilung in Komponenten-, Service-, Integrations-, Orchestrations- und Systemlevel-Tests realisiert, wird aber zur Vollständigkeit trotzdem angegeben.

Monitoring für jedes Service zu jeder Zeit: Der Entwicklungsprozess muss ein effizientes Monitoring der einzelnen Services und Schnittstellen zur Laufzeit garantieren. Dies wird mittels des Konzeptes der Sensoren realisiert, welches im Kapitel "Prozess zur Qualitätssicherung: Das WASI-Modell" realisiert wird.

Test von mobilen Endgeräten: Der Testprozess muss auch das Testen für mobile Endgeräte berücksichtigen. Um dies zu bewerkstelligen müssen Sensoren leichtgewichtig implementiert werden.

Testgetriebene Entwicklung der Services: Eine weitere Anforderung stellt die Verwendung der testgetriebenen Entwicklungsmethode auf Service- und Sensorenebene dar, weil diese eine hervorragende Qualität der einzelnen Dienste der Service-orientierten Architektur sicher stellt.

Anforderungen an das Test-Team

Hier werden explizit die Anforderungen an das Test-Team aufgelistet, da diese sehr wesentlich für erfolgreiches Testen sind, aber oft nicht beachtet werden.

Komplexität von Test- und Zeitplan: Bei Service-orientierten Architekturen gestaltet sich der Aufbau sowie der Test des Systems als hochgradig komplex. Der Entwicklungsprozess muss das Expertenteam bei dieser Aufgabe durch feste Ablaufsvorgaben unterstützen.

Tiefes Verständnis des Geschäftsbereichs: Das Test-Team der Architektur muss nicht nur ein tiefes technisches Verständnis haben, sondern auch vom zu implementierenden Geschäftsbereich. Die Dokumentation des Business-Prozesses innerhalb des Entwicklungsprozesses muss also dem interessierten Entwickler hinreichendes Verständnis für die Gesamtabläufe bieten.

Test-Teams für jeden Businessbereich: Der Entwicklungsprozess sieht die Aufteilung in kleine Testteams vor, welche den einzelnen Businessbereichen zugeordnet sind. Welche Technologien diese einzelnen Domänen verwenden ist dabei nicht entscheidend für die Zuteilung.

Zentrierung auf Business mit Services: Der Fokus des Testprozesses liegt auf dem Businessbereich mit seinen Services, welcher mit unterschiedlichsten Technologien realisiert sein kann.

Aufspaltung der SOA in kleine Teile zum erleichterten Testen: Dies wird vom Prozess ebenfalls implizit realisiert und bedarf keiner weiteren Betrachtung.

Anforderungen an den Aufbau des Prozesses

Diese Informationen dienen zur Konzeption des eigenen Entwicklungsprozesses, der auf einem bestehenden Modell aufbauen soll. Zur Auswahl dieses Modells müssen diese Requirements berücksichtigt werden.

PDCA-Zyklus: Der Plan - Do - Check - Act Zyklus soll durch häufige Iterationen den Prozess der Dokumentation, der Implementierung und der Überprüfung maßgeblich verbessern. Der Entwicklungsprozess muss diese Anforderungen in seiner Architektur umsetzen können.

Top-Down-Vorgehen für Spezifikation: Das System inklusive dessen Dokumentation soll zuerst gesamt erfasst werden, bevor mehr Details ausgearbeitet werden. Dies garantiert, dass das Gesamtziel einer funktionierenden, skalierbaren Service-orientierten Architektur nicht verfehlt wird.

Bottom-Up für den Testvorgang: Hingegen soll der Testprozess bei den kleinsten Einheiten (Komponenten) beginnen und sich sukzessive von den Services bis zum System-Level-Test steigern.

Testgetriebene Entwicklung: Der Prozess soll so aufgebaut sein, dass ein Großteil der Service-orientierten Architektur mittels der Methode der testgetriebenen Entwicklung realisiert wird um die Qualität der einzelnen Services der SOA zu steigern und für die Wiederverwendung zu sichern.

Anforderungen an die Sicherheit

Je größer die Service-orientierte Architektur wird (zum Beispiel durch Einbindung externer Services externer Unternehmen) desto wichtiger wird das Thema Sicherheit für das Projekt.

Sicherheitstests während der gesamten Projektlaufzeit: Diese stellen eine notwendige und absolut essentielle Anforderung dar. Durch frühzeitiges und vollständiges Security Testing in allen Phasen des Projekts soll die Sicherheit gewährleistet werden. Am Projektende müssen regelmäßig "Security Penetration Tests" durchgeführt werden um Schwachstellen zu finden und die laufende Sicherheit zu garantieren.

SOA Governance: Ein weiteres wesentliches Sicherheitskriterium stellt die Einhaltung der zuvor festgesetzten unternehmensinternen Spezifikationen und Standards dar. Kommt es zu Abweichungen können diese potentielle Schwachstellen für Angreifer darstellen.

Anforderungen an die Qualität

Dieser Abschnitt definiert die Qualitätsanforderungen an den Entwicklungsprozess. Aus diesen Anforderungen werden im Kapitel Qualitätssicherungsmaßnahmen für heterogene Systeme qualitätssichernde Maßnahmen abgeleitet.

Risikomanagement: Ein optionales Risikomanagementsystem minimiert die Gefahr des Scheitern des Projektes enorm und ist somit als fixe Anforderung für den Entwicklungsprozess vorgesehen.

PDCA-Zyklus: Der Entwicklungsprozess wird den Gedanken des "Plan - Do - Check - Act Zyklus" umsetzen, weil häufige Neuplanung beziehungsweise Anpassung ein entscheidender Faktor für die Qualität ist.

Integrations- und Interoperabilitätskriterien: Die Entwickler müssen sich bei der Konzeption und Implementierung an Standards orientieren um eine einfache Integration und eine große Interoperabilität zu gewährleisten. Dies muss vom Prozess vorgeschlagen werden.

Quality of Service (QoS): Jeder Dienst und der gesamte Prozess muss den zuvor spezifizierten Requirements entsprechen. Diese Anforderung wird in den Entwicklungsprozess aufgenommen.

Konsistenz: Die Implementierung, die Tests und die Dokumentation sollen eine konsistente Darstellung inklusive leicht zu verstehenden Kommentaren besitzen für maximale Nachvollziehbarkeit.

Effizienz: Wie in jedem existierenden Entwicklungsmodell soll die Effizienz im eigenen Prozess gesteigert werden. Dieses Qualitätskriterium wird bei der Konstruktion des Modells in den folgenden Kapiteln berücksichtigt.

Zuverlässigkeit: Die durch den Entwicklungsprozess entstehende Software soll ausfallsicher und korrekt bezüglich der geforderten Funktionalität funktionieren. Diese Anforderung wird im Modell implizit durch häufiges Testen implizit realisiert und wird zur Vollständigkeit hier als wesentliche Qualitätsanforderung angeführt.

Flexibilität: Änderungen wie auch Anpassungen des Programmcodes sollen jederzeit während des laufenden Projektes möglich sein. Dies stellt eine weitere Anforderung an den Entwicklungsprozess dar.

Agilität: Der Entwicklungsprozess soll eine agile Softwareentwicklung fördern und unterstützen.

Verfügbarkeit: Ein Ausfall von "Key Services" oder anderer für die Service-orientierte Architektur notwendige Artefakte kann zu enormen finanziellen Verlusten wegen eingeschränkter oder völliger Handlungsfähigkeit für ein Unternehmen führen. Die ständige Verfügbarkeit soll deshalb als essentieller Faktor für die Entwicklung des Prozessmodells eingeführt werden.

Kundenorientierung: Durch eine kundennahe Entwicklung und die Einbeziehung der Endbenutzer in die Erstellung der Service-orientierten Architektur inklusive Abnahmetests soll die Qualität maximiert werden. Diese Anforderung wird in den Entwicklungsprozess aufgenommen.

Zeit bis zur Marktreife durch Wiederverwendung: Durch Wiederverwendung von Schlüsselkomponenten kann die Zeit bis zur Marktreife einer neuen Service-orientierte Architektur reduziert werden.

SOA Governance: Mittels SOA Governance soll überprüft werden ob die unternehmensinternen Standards und Richtlinien eingehalten werden, daher stellt diese eine unverzichtbare Anforderung an das zu entwickelnde Modell dar.

Wiederverwendung von Services und Sensoren: Ist die Qualität der entwickelten Services und Sensoren hochwertig genug, können sie für zukünftige Projekte wiederverwendet werden, sonst ist die Wahrscheinlichkeit eines erneuten Einsatzes durch das Entwicklungsteam gering. Dieses Maß an Qualität wird vom neuen Entwicklungsmodell gefordert.

Skalierbarkeit der SOA: Die fertige Service-orientierte Architektur soll vollständig skalierbar sein, das heißt das Hinzufügen oder Entfernen von Services soll mit vertretbarem Aufwand ermöglicht werden. Dieses Kriterium kann durch den strikten Einsatz von Standards bei Service-orientierten Architekturen bewältigt werden.

Definition von Schlüsselservices: Diese Anforderung garantiert die Aufmerksamkeit auf besonders wichtige Services. Laufende Tests sollen die Qualität dieser "Key Services" sichern und eine anschließende Dokumentation der Qualität dieser Services soll eine Wiederverwendung ermöglichen.

Laufende Qualitätskontrollen: Durch häufige Qualitätsüberprüfungen der Services und der Orchestration sowie des gesamten Businessprozesses während der gesamten Projektlaufzeit soll die Qualität gesichert werden. Dies entspricht einer weiteren Anforderung, die durch das Prozessmodell umgesetzt werden muss.

Kosteneinsparung durch Qualität: Der Entwicklungsprozess soll eine Kostenreduktion durch erhöhte Qualität der Software bezwecken. Der Prozess soll explizit darauf hinweisen.

Anforderungen an die Projektleitung

Auch die Projektleitung und die Kunden werden in den Entwicklungsprozess aktiv integriert um die Qualität der Service-orientierten Architektur zu sichern.

Gemeinsames Erarbeiten der Business-Requirements: Die Anforderungen sollen gemeinsam mit den Endbenutzern und der verantwortlichen Management-Ebene definiert werden. Dies wird im Prozessmodell dem Projektleiter explizit vorgeschlagen.

Aufmerksamkeit schaffen: Das Management soll stets über die Projektfortschritte informiert werden um Aufmerksamkeit für das Projekt zu schaffen. Des Weiteren soll das Verständnis der Auftraggeber und der Projektleitung über die Wichtigkeit der Einhaltung der Governance der Service-orientierten Architektur gesichert werden. Der Prozess empfiehlt diese Vorgehensweise.

4.3 Qualitätssicherungsmaßnahmen

Dieses Kapitel hebt explizit Maßnahmen hervor, welche der Sicherung der Qualität bei der Erstellung von heterogenen Systemen mit besonderem Fokus auf Service-orientierte Architekturen

dienen. Diese Aktionen werden in konzeptuelle und technische Qualitätssicherungsmaßnahmen unterteilt, die in den Entwicklungsprozess aktiv integriert werden müssen um die hoch angelegten Qualitätsziele zu erreichen. Die Abbildung 4.2 zeigt eine Übersicht der Qualitätssicherungsmaßnahmen, welche in den Entwicklungsprozess integriert werden. Es folgt eine Beschreibung der konzeptuellen Maßnahmen.

Abbildung 4.2: Qualitätssicherungsmaßnahmen für den Entwicklungsprozess

Konzeptuelle Qualitätssicherungsmaßnahmen

Diese Maßnahmen betreffen hauptsächlich den Bereich Dokumentation im Entwicklungsprozess, aber sie bestimmen auch die Gestaltung des Aufbaus des Entwicklungsprozesses mit.

Die Verwendung eines PDCA-Zyklus für die Dokumentation soll häufige Anpassungen/Än-

derungen zur Steigerung der Qualität garantieren. Die initiale Planung, die Ausführung der geplanten Tätigkeiten (das heißt u.a. die Umsetzung des funktionalen Entwurfs), die laufende Überprüfung und Anpassung der Dokumente inklusive aller UML- und Business Process Modelling Notation (BPMN) Diagramme während des gesamten Projektverlaufs werden im Entwicklungsprozess elementare Bestandteile sein.

SOA Governance bezeichnet die Umsetzung von unternehmensinternen Richtlinien und Standards in der Service-orientierten Architektur. Die Festlegung dieser Regeln und deren laufende Überwachung sollen im Testplan fest verankert werden und sind Teil des Entwicklungsprozesses.

Aktives Risikomanagement im Projekt soll Risiken vermeiden und helfen die Kosten bei eingetretenen Risiken zu reduzieren. Alle projektgefährdenden Risiken sollen zu Beginn identifiziert und analysiert werden. Eine qualitative und quantitative Risikobewertung durch das gesamte Entwicklungsteam soll eine objektive Einschätzung der zuvor gefundenen Risiken liefern. Die anschließende Risikosteuerung trifft Entscheidungen über den Umgang mit den Risiken, kommt es zur Akzeptanz ohne weiterem Handlungsbedarf, soll man das Risiko unbedingt vermeiden, die Auswirkungen von eingetretenen Risiken vermindern oder das Problem auf eine andere Division verlagern. Die Risikoüberprüfung kontrolliert anschließend die Wirksamkeit der zuvor durchgeführten Risikosteuerung. Das gesamte Risikomanagement soll mit Hilfe des PDCA-Zyklus im Entwicklungsprozess implementiert werden.

Ein Testplan mit häufigen Iterationen soll die Durchführung alle möglichen Arten von Tests, dazu zählen Komponenten-, Service-, Sensoren-, Integrations-, Orchestrations-, System-, Regressions-, SOA Governance-, Sicherheits- und Abnahmetests, planen und deren Ergebnisse dokumentieren. Die Häufigkeit der Tests und der Einsatz einer testgetriebenen Entwicklung sollen die Qualität im Entwicklungsprozess sichern.

Aufspaltung der SOA in kleine Teile Der funktionale und technische Systementwurf muss die Service-orientierte Architektur in kleine Teile aufspalten, damit eine Übersicht über die gesamte SOA leicht möglich ist, die Implementierung einfacher auf verschiedene Verantwortliche aufgeteilt werden kann und die durchzuführenden Tests überschaubar bleiben. Der Entwicklungsprozess berücksichtigt dies in den Phasen des funktionalen und technischen Systementwurfs.

Die Definition von Schlüsselservices wird ebenfalls in den Entwicklungsprozess integriert. Services, welche von höchster Bedeutung für die Service-orientierte Architektur als auch für eventuelle nachfolgende Projekte sind, werden besonders genau getestet und auf korrekte Funktionalität bezüglich der Spezifikation überprüft. Die Fehlerfreiheit dieser Dienste betrifft auch das wichtige Qualitätskriterium der Wiederverwendbarkeit.

Eine häufige Analyse der Reporting Daten soll vorhandene Bottlenecks und/oder sonstige Probleme in der gesamten heterogenen Systemlandschaft ausfindig machen. Die zuvor aus dem Monitoring gewonnenen Daten werden ausgewertet und durch die Methode des Reportings in Berichte, welche relevante Informationen für das Unternehmen enthalten,

transformiert. Qualitätssichernde Metriken werden in einem späteren Kapitel dieser Arbeit behandelt. Dieses Qualitätskriterium wird im Prozess berücksichtigt.

Technische Qualitätssicherungsmaßnahmen

Die technischen Qualitätssicherungsmaßnahmen sind Aktionen, die während der Implementierung und der Tests des gesamten Projektes umgesetzt werden müssen um eine effiziente und fehlerfreie Erstellung der Service-orientierten Architektur gewährleisten zu können.

Sensoren sollen als Konzepte für permanentes Monitoring der einzelnen Services der Service-orientierten Architektur eingeführt werden. Sie werden über ein Interface in jedes Service eingebunden und geben Rückmeldung über den jetzigen Zustand des Services inklusive verfügbarer Daten. Diese Technologie in Kombination mit Reporting erleichtert die Fehlersuche und die Lokalisierung von Schwachstellen für den Entwickler enorm und wird als komplett neue Idee Teil des zu entwickelnden Prozesses.

Eine testgetriebene Entwicklung für Komponenten, Services und Sensoren soll Einzug in das Prozessmodell finden. Diese Methode des Testens garantiert eine weitgehend fehlerfreie Implementierung und die Einhaltung der zuvor festgelegten Anforderungen bei der Erstellung der heterogenen Systemlandschaft.

Monitoring und Reporting sollen weitere technische Kernkonzepte des Entwicklungsprozesses darstellen. Monitoring des Status der einzelnen Services zur Laufzeit soll mittels des Konzeptes der Sensoren, welche noch vorgestellt werden, realisiert werden. Die gewonnenen Daten werden mittels zuvor definierter Reporting-Metriken aufbereitet.

Ein PDCA-Zyklus für die Implementierung steigert die Qualität bei der Dokumentation. Nach der anfänglichen Planung kommt es zu einer Implementierung mit Hilfe der testgetriebenen Entwicklungsmethode auf unterster Ebene, die erstellten Services werden dann zu einem Business-Prozess zusammengeschlossen. Die anschließenden Tests auf allen Ebenen sollen Fehler aufzeigen, die schließlich verbessert werden. Dieser Zyklus soll mehrmals innerhalb des Projektes durchlaufen werden und genießt somit hohe Priorität im Entwicklungsprozess.

SOA Governance Testing: Die zuvor im SOA Governance ermittelten Standards und Richtlinien des Unternehmens werden auf Korrektheit überprüft. Kommt es zu Abweichungen, müssen diese umgehend beseitigt werden um die Sicherheit und die standardisierten Abläufe in der Service-orientierten Architektur nicht zu gefährden. SOA Governance Testing wird im Prozessmodell ebenfalls implementiert. Zum besseren Verständnis soll noch ein Beispiel für die Überwachung der SOA Governance gegeben werden: Ein Unternehmen setzt für die Berechnungszeit einer Aufgabe eines Services eine bestimmte Obergrenze, die nicht überschritten werden darf. Ein Test soll die Einhaltung prüfen und bei Regelverstoß eine Optimierung der Laufzeit des Dienstes veranlassen.

Security Testing oder auch Sicherheitstests genannt, überprüfen die Sicherheit der einzelnen Services/der gesamten Architektur. Die Sicherstellung der unternehmensinternen Daten

wird fokussiert, dies bedeutet das Sicherheitstest während der Entwicklung und "Security Penetration Tests" bei und nach Fertigstellung der Software im Prozess inkludiert werden.

Häufige Integrationstests sollen die weitgehend fehlerfreie Zusammenarbeit der einzelnen Services garantieren. Sie werden mit hoher Iterationszahl als wichtiger Bestandteil in den Entwicklungsprozess aufgenommen.

Regressionstests verhindern, dass neue Fehler unabsichtlich durch Ausbesserung der Alten in das System eingeschleust werden. Dieser Test soll im Entwicklungsprozess immer durchgeführt werden, wenn Fehler beseitigt wurden.

Testautomation soll den Gesamtaufwand für das umfangreiche Testen im Prozess auf ein vertretbares Maß reduzieren.

4.4 Prozess zur Qualitätssicherung: Das WASI-Modell

Nach der Identifikation aller notwendigen Anforderungen und der anschließenden Festlegung der qualitätssichernden Maßnahmen, welche im entwickelten Prozess absolute Priorität genießen, folgt nun ein Abschnitt, der das Vorgehen im Prozessmodell im kleinsten Detail unter Berücksichtigung der oberhalb angeführten Punkte und aus den Kenntnissen intensiver Recherche der Thematik beschreibt. Das in den Grundlagen erwähnte V-Modell eignet sich gut für die durch ein Prozessmodell unterstützte Implementierung einer Service-orientierten Architektur, weil es den in den Anforderungen verlangten Top-Down Ansatz für die Dokumentation und das Bottom-up Vorgehen für die Implementierung fordert. Ein großer Nachteil ergibt sich bei diesem Entwicklungsprozess jedoch aus dem Umstand, dass keine nachträglichen Änderungen im Vorgehen möglich sind. Das ebenfalls betrachtete W-Modell behebt diesen Missstand glücklicherweise, so dass es für das eigene Prozessmodell als Grundlage dienen kann: Fast alle benötigten Tests werden schon während der Dokumentationsphase vorbereitet und nach der Implementierung sofort ausgeführt, nur SOA Governance Tests, Sicherheitstests und Regressionstests wurden nicht explizit berücksichtigt. Auch fehlt eine Unterstützung für ein Monitoring der einzelnen Services, die Schnittstellen lassen sich somit nicht effizient testen. Für das eigene Modell muss somit das W-Modell passend erweitert werden. Das Hinzufügen von Sensoren und einer testgetriebenen Entwicklung für die Komponenten und Services scheint die benötigten Anforderungen zur Gänze zu erfüllen.

Die Abbildung 4.3 illustriert das WASI (Well Assisted SOA Implementation) Modell, welches den Prozess in vier Phasen unterteilt:

- **Dokumentationsphase:** In dieser Phase werden die Anforderungen ermittelt und ein funktionaler und technischer Systementwurf inklusive passender Dokumentation angefertigt. Gleichzeitig werden Abnahmetest, Systemtest und Integrationstest vorbereitet.

- **Detailspezifikation und Testgetriebene Entwicklung:** Hier werden die Sensoren, Services und Komponenten spezifiziert und anschließend die Tests bezüglich der Anforderungen erstellt, bevor noch Programmcode geschrieben wurde. Die Methode des Test Driven Development wird angewendet.

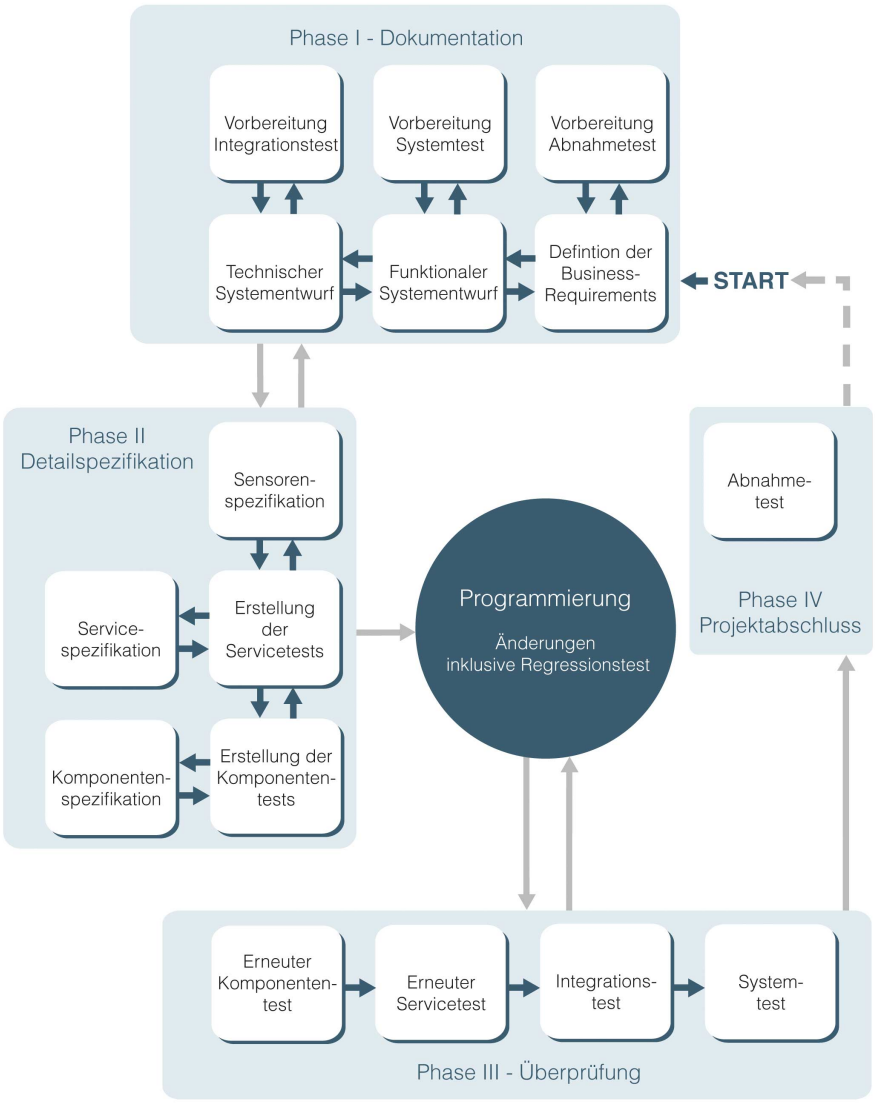

Abbildung 4.3: Well Assisted SOA Implementation Model (WASI)

- **Durchführung aller Tests:** Nach der Programmierung der Komponenten (verbunden mit den fertigen Sensoren) und Services werden die im vorherigen Punkt implementierten Tests nochmals ausgeführt. Es wird solange programmiert, bis die Tests fehlerfrei terminieren. Dann folgt die Integration der einzelnen Komponenten und Services mit abschließendem Integrationstest. Wurden alle Services integriert, folgt der Systemtest, der den gesamten Business-Prozess überprüft. Sollten Fehler bei diesen Tests auftreten, wird der bestehende Code verändert und Regressionstests durchgeführt.

- **Projektabschluss mit Abnahmetest:** Ist die Service-orientierte Architektur vollständig

implementiert und fehlerfrei, wird zusammen mit dem Kunden die letzte Testphase durchlaufen. Sollte der Kunde Änderungswünsche haben wird wieder bei der Dokumentationsphase begonnen.

Jedes Rechteck dieser Grafik zeigt dabei zentrale Aktivitäten bei der Entwicklung, die Pfeile bedeuten, dass die nächste Aktion stattfinden kann. Ein Paar Richtungspfeile mit entgegen gesetzter Anordnung, bedeutet, dass jederzeit wieder zum vorherigen Schritt gewechselt werden kann. Der Kreis mit dem Inhalt "Programmierung & Änderungen" steuert den verteilten Implementierungs- und Testprozess. Die Aktivitäten werden in den folgenden Unterkapiteln angeführt und das Vorgehen und ihre erwarteten Ergebnisse erläutert.

Business-Requirements & Zeitplan

Diese initiale Phase des Projektes muss zusammen mit den Endbenutzern und Experten der jeweiligen Business-Domain durchlaufen werden. Folgende Dokumente sollen Ziel dieser Aktivität sein:

1. **Überblick verschaffen:** Zuerst soll ein Überblick über den abzubildenden Geschäftsbereich erstellt werden, die grobe Funktionalität soll graphisch illustriert werden (zum Beispiel mittels Skizze).

2. **Identifikation und Analyse der Anforderungen:** Die funktionalen und technischen Anforderungen an das System sollen gemeinsam mit dem Kunden identifiziert werden. Diese werden dann in einem Dokument aufgelistet und analysiert. Den Anforderungen sollen verschiedene Prioritäten zugeteilt werden, die mittels einer zuvor definierten Metrik (zum Beispiel die Zahl 1 bedeutet geringe Priorität und die Zahl 10 maximale Priorität) eine quantitative Bewertung dieser Requirements erlauben. UML-Anwendungsfalldiagramme sollen als Hilfsmittel anschließend die gewünschte Funktionalität jeder Anforderung zusätzlich dokumentieren.

3. **Zeitplan**: Nach Erhebung der Anforderungen muss ein grober Plan die benötigte Projektzeit und erforderlichen Projektressourcen einschätzen. Meilensteine in Form von fertiggestellten Services zu einer bestimmten Zeit sollen definiert werden.

4. **Testplan:** Ein grober Testplan soll sämtliche durchzuführenden Testaktivitäten planen, dokumentieren und die Ergebnisse festhalten. Der Testplan muss unbedingt mit dem Zeitplan abgeglichen oder kombiniert werden, weil der gesamte Entwicklungsprozess auf Tests aller Art (Abnahmetest, Systemtest, Integrationstest (Orchestrationstests), Servicetests, Komponententests, Regressionstests, SOA Governance Testing, Security Testing, Security Penetration Testing) aufbaut.

5. **SOA Governance:** Im ersten Teil des Service-orientierten Architektur-Governance sollen die unternehmensinternen Richtlinien hinreichend protokolliert werden um sie später bei der Vorbereitung diverser Tests gleich einsetzen zu können. Die Regel wird eingeführt, dass Verstöße gegen Richtlinien in unternehmensinternen SOA Diensten nicht geduldet werden.

6. **Risikomanagement (Risikoidentifikation und Risikobeurteilung):** Auch das in den Prozess integrierte Risikomanagement startet schon zu Projektbeginn. Alle Risiken sollen identifiziert und ebenfalls mit Hilfe der potentiellen Schadenshöhe und ihrer Eintrittswahrscheinlichkeiten priorisiert werden (zum Beispiel 10.000 Euro Schadenshöhe bei einem Projektvolumen von 100.000 und einer Eintrittswahrscheinlichkeit von 90% erhält eine sehr hohe Priorität innerhalb des Projektes). Die Sinnhaftigkeit des Risikomanagements darf nicht unterschätzt werden, sie beugt Fehlern vor und hilft die Gesamtkosten zu reduzieren.

Guideline für Sicherheitstests

Für die Vorbereitung aller nachfolgenden Sicherheitstests wird die folgende Vorgehensweise vorgeschlagen:

1. **Identifikation von Schwachstellen:** Eine Analyse durch Sicherheitsexperten ohne und mit Hilfe von Werkzeugen zur Identifikation von Schwachstellen (beispielsweise automatisierte Suche nach Buffer-Overflow Möglichkeiten in einem bestehenden "C"-Programm) soll die Schwachstellen im entwickelten Fragment (Komponente oder Service) aufzeigen.

2. **Regelmäßige Überprüfung:** Die gefundenen "Security-Holes" sind im Testplan zu dokumentieren und entsprechende Sicherheitstests zu implementieren. Durch eine regelmäßige, zentralisierte Ausführung dieser Überprüfungen soll die Sicherheit garantiert werden.

3. **Sofortige Ausbesserung bei Fehlern:** Werden Fehler durch die Sicherheitstests aufgedeckt, müssen diese durch die Verantwortlichen der Komponente oder des Services sofort ausgebessert werden.

Vorbereitung Abnahmetest

Der **Abnahmetest** entspricht dem letzten Stadium im Testprozess. Gemeinsam mit den Endbenutzern/Kunden wird die implementierte Service-orientierte Architektur bezüglich der Erfüllung aller ermittelten Anforderungen überprüft. Sollten hier grobe Mängel festgestellt werden, müssen Änderungen dokumentiert werden und der gesamte Entwicklungsprozess ein weiteres Mal durchlaufen werden, da Anpassungen oft nicht ohne elementare Änderungen einzelner Services durchgeführt werden können.
Im Testplan wird eine eigene Sektion für den Abnahmetest mit Hilfe einer Checklist für alle Anforderungen und einer Bewertung der Zufriedenheit der Kunden/Benutzer geschaffen.
Des Weiteren wird im Testplan ein weiterer Abschnitt angelegt: **"Security Penetration Testing"**, das bedeutet die Service-orientierte Architektur soll am Projektende auf Sicherheit gegen Hacker-Angriffe und unberechtigter Datenzugriffe überprüft werden.
Es wird dem Unternehmen auch empfohlen diese Tests auch nachher laufend fort zu führen und in regelmäßigen Zeitabständen die Sicherheit des Systems zu überprüfen um die Chancen für den Verlust oder Betriebsspionage zu minimieren.
Auch eine endgültige Überprüfung der Maßnahmen des Risikomanagements (speziell der Risikos-

teuerung während der Implementierung) soll vorbereitet werden.
Alle hier genannten Elemente sollen im Testplan entsprechend dokumentiert werden.

Funktionaler Systementwurf

Das Ziel des funktionalen Systementwurfs ist ein kompletter Überblick über den Business-Prozess. Der Geschäftsbereich, der abgebildet werden soll, wird abstrahiert. Der Ablauf zwischen den einzelnen Business Requirements, die im initialen Schritt des Entwicklungsprozesses identifiziert wurden, soll erfasst und dokumentiert werden. Hier wird folgendes Vorgehen vorgeschlagen:

1. **Einsatz der Unified Modelling Language:** Die Zusammenhänge und Abläufe zwischen den Anforderungen sollen mit Hilfe von mehreren UML Aktivitäts- und Sequenzdiagrammen dokumentiert werden.

2. **Business Process Modelling Notation (BPMN):** Mit Hilfe dieser graphischen Spezifikationssprache kann der gesamte Businessprozess detailliert abgebildet werden. Die Interaktion Mensch-Prozess kann abgebildet werden und die Orchestrierung von Prozessen wird unterstützt. Seit der Version 2.0 können bestehende Modelle auch leicht abgeändert oder erweitert werden.

3. **Business Process Execution Language (BPEL):** Verzichtet man auf die testgetriebene Entwicklung im WASI-Modell, kann als Alternative auch die Business Process Execution Language zur automatischen Code-Generierung eingesetzt werden. Der Prozess muss dann allerdings stark adaptiert werden, weil er zur Qualitätssteigerung der gesamten Entwicklung auf der "Test-driven Development" Methode aufbaut.

4. **Analyse des Domainenmodells des funktionalen Systementwurfs:** Aus dem bestehenden BPMN-Domainenmodell sollen alle Services identifiziert und in einem Dokument aufgelistet werden.

5. **Definition von Schlüsselservices:** So genannte Key Services sollen identifiziert werden, sie sind besonders wichtig im Businessprozess und für eine eventuelle Wiederverwendung in anderen Service-orientierten Architekturen. Diese werden im Laufe des Prozesses vermehrt getestet, weil deren Qualität von äußerster Priorität sein muss.

Das Ergebnis der Phase des funktionalen Systementwurfes soll also ein komplettes Domainenmodell sein, welches alle Abläufe für die Service-orientierte Architektur spezifiziert.

Vorbereitung Systemtest

Der Systemtest soll die vorletzte Testphase des Projektes darstellen, er wird direkt im Anschluss an die Dokumentation des funktionalen Systementwurfs vorbereitet. Die fertige Service-orientierte Architektur wird vom Entwicklungsteam auf Erfüllung aller Business Requirements getestet. Werden in dieser Phase Fehler gefunden, werden diese auf der jeweiligen identifizierten Ebene ausgebessert und mit Regressionstests wird überprüft ob durch die Änderungen nicht neue Fehler in der SOA entstanden sind.

Zusätzlich soll das ganze System auf Einhaltung der SOA Governance zum letzten Mal überprüft werden. Abweichungen dieser Richtlinien werden nicht toleriert. Abschließende Security Tests sollen die Sicherheit der gesamten Service-orientierten Architektur gewährleisten. Im Systemtest dürfen auf keinen Fall Fehler in Key Services gefunden werden, ohne deren Fehlerfreiheit darf das Projekt nicht abgeschlossen werden.
Sämtliche genannten Artefakte sollen im Testplan dokumentiert werden.

Technischer Systementwurf

Der dritte Teil der Dokumentationsphase wird durch die Spezifikation des technischen Systementwurfs eingeleitet. Es werden die verwendeten Technologien festgesetzt und anschließend die einzelnen Services der SOA in einzelne Komponenten unterteilt um die Komplexität der Architektur unter Kontrolle zu bringen und den Testvorgang zu erleichtern. Folgende Punkte definieren im Modell den technischen Systementwurf:

1. **Analyse des Domainenmodells des funktionalen Systementwurfs:** Die Dokumentation der Services aus dem funktionalen Systementwurf wird verwendet, die Kooperation und unerwünschte Abhängigkeiten sollen gefunden und ebenfalls dokumentiert werden.

2. **Spezielle Kennzeichnung der identifizierten Key Services:** Diese Services sollen im Programmcode mit Hilfe von Kommentaren als besonders wichtige Teile der Implementierung markiert werden.

3. **Festlegung der Technologien und Standards:** Nach der Analyse sollen die zu verwendenden Programmiersprachen (beispielsweise Java, C#, ...), Technologie für Datenbanken (Beispiel: MySQL), Kommunikationstechnologien (beispielsweise SOAP) und die Verwendung von Standards (beispielsweise XML) spezifiziert werden. Hierbei gilt folgende Regel: Sind Standards verfügbar sollen diese zur Steigerung der Interoperabilität und Integrationsfähigkeit verwendet werden, außer es bestehen zwingende Gründe dies zu unterlassen. Des Weiteren sollen für jedes Service zwingende SOA Governance Richtlinien definiert werden.

4. **Aufspaltung der Services in Komponenten:** Die einzelnen Services aus der Analyse sollen weiter aufgespaltet werden, ein Dienst soll aus mehreren Komponenten bestehen. Diese Komponenten sollen mit Hilfe von UML-Klassendiagrammen dokumentiert werden.

5. **Schnittstellen zwischen den Services:** Die Interfaces zwischen den einzelnen Services sollen klar definiert sein. Die Verwendung von standardisierten Interfaces fördert die Unabhängigkeit einzelner Services und die Wiederverwendung bestehender Dienste.

6. **Integrationspunkte für Sensoren:** Prinzipiell soll das Konzept der Sensoren bei jedem Service angewendet werden, weil es effizientes Testen der Schnittstellen und das Monitoring zur Laufzeit der Service-orientierten Architektur ermöglicht.

Vorbereitung Integrationstests

Nach der Spezifikation des technischen Systementwurfs sollen die Integrationstests vorbereitet werden. Sie überprüfen die korrekte Funktionalität der Kooperation einzelner Services über Schnittstellen. Auch hier gilt: Werden Fehler gefunden, werden diese auf der jeweiligen Ebene ausgebessert und mit Hilfe von Regressionstests sicher gestellt, dass die Ausbesserungen keine neuen Fehler in das bestehende System bringen.
Zusätzlich wird auch hier die SOA Governance mit Tests überprüft und Abweichungen von den zuvor festgelegten unternehmensinternen Richtlinien und Standards festgestellt. Änderungen werden wieder mit anschließenden Regressionstests durchgeführt. Die Sicherheit der miteinander kombinierten Services soll auch abschließend noch geprüft werden. Falls bei der Kombination der einzelnen Services Schlüsseldienste dabei sind, muss absolute Priorität auf komplette Fehlerfreiheit gelegt werden.
Alle in diesem Abschnitt genannten Punkte sollen im Testplan ausreichend dokumentiert werden.

Spezifikation der Sensoren

Die Spezifikation der Sensoren soll folgendem Ablauf folgen:

1. **Einsatz der Sensoren bestimmen:** Im ersten Schritt soll für jeden aus dem funktionalen Systementwurf ermittelten Dienst bestimmt werden, welche Daten für das Monitoring von Relevanz sind. Aus dieser Zuordnung ergeben sich dann verschiedene Kategorien von Sensoren.

2. **Festlegung der Sensoren für die Services:** Jedem Service soll dann eine Sensor-Kategorie zugewiesen werden, damit ein effizientes Monitoring der gesamten Service-orientierten Architektur vollzogen werden kann. Des Weiteren ist auch eine Nutzung mehrerer Sensoren für ein Service möglich (beispielsweise ein generischer und ein Datenbanksensor).

3. **Programmiersprache festlegen:** Idealerweise sollte die verwendete Programmiersprache für alle einzelnen Komponenten die Gleiche sein. Für den Sensoreinsatz kann es jedoch aus verschiedenen Gründen notwendig sein eine andere Sprache zu verwenden (beispielsweise Java für höhere Portabilität auf andere Systeme oder C für die Leichtgewichtigkeit des Sensors für mobile Endgeräte).

4. **Standard für Interfaces festlegen:** Für jede Kategorie von Sensoren sollen Schnittstellen entworfen werden, die dann eine standardisierte Verwendung garantieren. Dies soll leichte Änderungen und eine einfache Erweiterung der Service-orientierten Architektur ermöglichen.

5. **Anpassung der Sensoren mittels Konfigurationsdatei:** Reicht für eine Service-Kategorie die implementierte Funktionalität nicht, so sollen sich gewisse Eigenschaften mit Hilfe einer Konfigurationsdatei (bspw. im XML-Format) anpassen lassen (Beispiel: Erhöhe die Output-Rate für das Monitoring für ein Service, diese Daten werden dann zum Monitoring Backend geschickt).

6. **SOA Governance**: Die unternehmensinternen Richtlinien und Standards sollen bereits bei der Spezifikation der Sensoren berücksichtigt werden. Zum besseren Verständnis wird hier ein einfaches Beispiel präsentiert: Für die Bearbeitungszeit einer Aufgabe wurde für ein Service eine zeitliche Obergrenze von einer Organisation definiert. Der Sensor muss diese Ausführungszeit messen können, das heißt diese Monitoring-Komponente muss bei der Sensorenspezifikation berücksichtigt werden.

Service-Spezifikation

In diesem Schritt des Entwicklungsprozesses werden die einzelnen Services näher spezifiziert: Eine technische Beschreibung bis ins kleinste Detail mit Hilfe von Diagrammen und ein erneutes Überprüfen der Anforderungen, welche der Dienst abdecken muss, sollen durchgeführt werden. Hierfür wird folgende Vorgehensweise vom Prozessmodell empfohlen:

1. **Erneute Überprüfung der Anforderungen der Services:** Die im Schritt Business Requirements ermittelten und im funktionalen Systementwurf den einzelnen Services zugeteilten Anforderungen sollen nochmals auf korrekte Spezifikation und Zuordnung überprüft werden. Werden Fehler in der Dokumentation gefunden, sollen diese umgehend ausgebessert und alle erforderlichen Schritte erneut durchlaufen werden.

2. **SOA Governance**: Die unternehmensinternen Richtlinien und Standards sollen bereits bei der Spezifikation der Services berücksichtigt werden. Zur Förderung des Verständnisses wird auf das Beispiel, welches bereits im Kapitel "Spezifikation der Sensoren" beschrieben wurde, zurückgekehrt: Bereits bei der Spezifikation und Entwicklung des Services soll die zeitliche Obergrenze berücksichtigt werden (Messung der Berechnungszeit bereits während der Entwicklung, eventuelle Optimierung des Dienstes).

3. **Unabhängigkeit und Standards:** Die Services sollen unabhängig voneinander funktionieren. Kann dies nicht garantiert werden, muss der funktionale und technische Systementwurf für eine korrekte Implementierung nachträglich angepasst werden. Des Weiteren sollen die festgesetzten technologischen Standards eingehalten werden.

4. **Unterteilung in Komponenten:** Die einzelnen unabhängigen Services sollen zur weiteren Vereinfachung in einzelne, voneinander abhängige Komponenten unterteilt werden. Dies ermöglicht einfache automatisierte Unit-Tests auf Komponentenebene.

5. **Integration der Sensoren:** Die zuvor spezifizierten Sensoren werden dem Service zugeordnet und müssen bei der technischen Spezifikation des Dienstes berücksichtigt werden.

6. **Technische Spezifikation mit UML-Klassendiagrammen:** Jedes Service soll mit Hilfe der zuvor identifizierten Komponenten graphisch in der Form von Klassendiagrammen der "Unified Modeling Language" abgebildet werden. Die einzelnen Komponenten mit ihren Klassenvariablen und Methoden werden erst im nächsten Schritt detailliert spezifiziert werden, jedoch sollen die Schnittstellen zwischen den Komponenten bereits klar festgelegt werden. Im nächsten Schritt wird das Klassendiagramm dann vervollständigt.

Erstellung der Service-Level-Tests

Nach der Spezifikation werden die Tests geschrieben, welche die identifizierten Anforderungen für jedes Service auf korrekte Funktionalität überprüfen sollen. Dies geschieht mit Hilfe der testgetriebenen Entwicklungsmethode: Zuerst sollen die Tests erstellt werden, erst danach wird programmiert, bis alle zuvor entwickelten Tests fehlerfrei durchlaufen werden. Dies soll die Qualität der Services steigern. Die Integration der benötigten Sensoren muss bei der Erstellung dieser Tests ebenfalls beachtet werden.
Auch die SOA Governance Level Tests müssen für die einzelnen Services vor der Implementierung erstellt werden. Des Weiteren muss die Sicherheit für jeden Dienst überprüft werden. Diese Tests sollen ebenfalls der testgetriebenen Methode folgen.
Die oben genannten Punkte im Entwicklungsprozess sollen im Testplan entsprechend dokumentiert werden.

Komponentenspezifikation

Hier werden die einzelnen Komponenten im Detail mit ihren Klassennamen, Klassenvariablen und statischen, öffentlichen und privaten Methoden genau beschrieben. Der Entwicklungsprozess schlägt hierfür folgende Verfahrensweise vor:

1. **SOA Governance:** Die unternehmensinternen Richtlinien und Standards müssen auch auf Komponentenebene berücksichtigt und eingehalten werden. Dies ist bei der Spezifikation der einzelnen Komponenten zu beachten.

2. **Namenskonventionen:** Sofern im technischen Systementwurf näher spezifiziert, müssen die Namenskonventionen für Klassen, Variablen und Methoden eingehalten werden.

3. **Standards:** Die im technischen Systementwurf festgelegten technologischen Standards (Programmiersprache, Aufbau von Interfaces, ...) müssen berücksichtigt werden.

4. **Komponentenspezifikation:** Unter strikter Einhaltung der oben genannten Punkte werden die einzelnen Komponenten mit Namen und Funktionsweise spezifiziert.

5. **Schnittstellenspezifikation:** Des Weiteren werden die einzelnen Interfaces zwischen den Komponenten festgelegt, die zur Komposition des jeweiligen Services dienen.

6. **Integration der Sensoren:** Die auf Service-Level Ebene zugeteilten Sensoren werden nun der jeweiligen Komponente zugeordnet um eine korrekte Monitoring-Funktionalität zu garantieren.

Erstellung der Komponententests

Die spezifizierten Komponenten werden anschließend durch Tests, welche die korrekte Funktionalität überprüfen sollen, repräsentiert. Dies geschieht wiederum vor der eigentlichen Implementierung, es soll genau wie bei den Services auf den Einsatz einer testgetriebenen Entwicklungsmethode geachtet werden. Kommt es zu Anomalien zwischen Komponenten- und Dienstspezifikation müssen diese in der jeweiligen Ebene beseitigt und der gesamte Prozess ab

diesem Level erneut durchlaufen werden. Die Implementierung der Sensoren muss von den Komponententests ebenfalls berücksichtigt werden.
Auch auf dieser Ebene sollen SOA Governance Tests, welche die unternehmensinternen Richtlinien und Standards überprüfen, und Security Tests zur Sicherung der Unternehmensdaten durchgeführt werden. Zusätzlich soll die Funktionsweise zwischen einzelnen Komponenten (das heißt die Schnittstellen) getestet werden.
Alle Testaktivitäten sollen im Testplan dokumentiert werden. Die Sensoren sollen in einer Tabelle den einzelnen Testfällen zugeordnet werden.

Programmierung & Änderungen

Dieser Schritt ist eng mit den einzelnen Testvorgängen verknüpft. Werden Fehler im Programmcode durch einzelne Tests aufgedeckt, sollen Änderungen jederzeit möglich sein. Deswegen definiert das Prozessmodell die Testaktivitäten in unmittelbarer Abhängigkeit mit der Programmierung: Werden Fehler gefunden, können diese unmittelbar behoben werden.
Änderungen am Programmcode sollen stets nach dem Pair Reviewing Prinzip durchgeführt werden. Um dem Risiko der Entstehung neuer Fehler durch Ausbesserungen alter Fehler vorzubeugen, wird das Konzept des Regressionstests angewendet. Sobald der Programmcode fehlerbedingt geändert wurde, wird erneut die jeweilige Ebene komplett auf Fehler überprüft. Werden hier Fehler gefunden, müssen diese wiederum ausgebessert und der Regressionstest erneut ausgeführt werden bis die zu entwickelnden Teile der Service-orientierten Architektur fehlerlos terminieren.
Auch die Steuerung des Risikomanagements ist in dieser Phase integriert. Die im Schritt der Business Requirements gefundenen und bewerteten Projektrisiken werden hier periodisch überwacht. Die Risikosteuerung soll je nach Priorität helfen Risiken zu vermeiden, eingetretene Risiken zu vermindern, zu verlagern oder zu akzeptieren (d.h.: Das eingetretene Risiko wird ohne weiterer Konsequenzen als gegeben hingenommen). Auch die Risikosteuerung ist eng mit den Testvorgängen verknüpft, welche bei der Minimierung von Projektrisiken behilflich sind.
Bevor jeder Änderung am System soll mit Hilfe eines Versionierungssystems (beispielsweise Subversion) der momentane Implementierungsstatus gesichert werden und geänderte Klassen mit aussagekräftigen Kommentaren in das Versionierungssystem übertragen werden.
Während die Spezifikation nach dem Top-Down Prinzip erstellt wurde, erfolgt die Erstellung der Service-orientierten Architektur nach der Bottom-Up Vorgehensweise. Der verteilte Implementierungs- und Testprozess gliedert sich in folgende Punkte, welche in mehreren Iterationen bis zum gewünschten Endergebnis durchlaufen werden:

1. **Implementierung der Komponenten:** Zuerst werden die einzelnen Komponenten für ein einzelnes Service implementiert. Diese werden wieder strikt nach der Testgetriebenen Methode entwickelt. Die einzelnen Komponententests werden an alle Entwickler verteilt, welche die Implementierung mit Hilfe von Unit-Tests und Dummy-Objekten (siehe Kapitel 3.2 "jUnit-Tests") unabhängig voneinander durchführen. Die Programmierer informieren den Projektleiter in einem zentralen Verwaltungssystem über den Implementierungsstatus der Komponenten. Erst wenn alle Komponententests eines Dienstes fehlerfrei abschließen (siehe "Erneuter Komponententest") darf zum nächsten Punkt übergegangen werden. Werden jedoch Fehler gefunden, müssen diese ausgebessert und die gesamte Komponente

neu getestet werden. Die Definitionen der jeweiligen Komponentenspezifikation und der Vorbereitungen für alle Komponententests sind in der Programmierung zu berücksichtigen.

2. **Implementierung eines Service:** Sind alle Komponenten eines Service vollständig implementiert, werden die Sensoren gemäß ihrer Spezifikation implementiert und an die jeweiligen Komponenten gebunden. Für diese Aufgabe werden wiederum Entwickler bestimmt, welche auch die einzelnen fehlerfreien Komponenten zu Diensten kombinieren. Die zuvor erstellten Service-Tests werden dann vom Testleiter zentralisiert ausgeführt, sie nutzen die Sensoren zur Überprüfung der korrekten Funktionalität. Durch das Monitoring können Fehler im Zusammenspiel der Komponenten identifiziert, lokalisiert und ausgebessert werden. Die Verantwortlichen werden vom Testsystem oder vom Projektleiter über die gefundenen Fehler informiert. Weichen die Testresultate von den erwarteten Werten ab, müssen alle zugehörigen Komponenten- und die Servicetests (mit Hilfe der Sensoren) erneut durchlaufen werden. Die Fehlerfreiheit des gesamten Dienstes wird somit garantiert. Erst dann kann der nächste Punkt im Implementierungsprozess behandelt werden. Auch hier sind die Definition der Servicespezifikation und der Vorbereitungen für den Servicetest in der Programmierung zu beachten. Besonderes Augenmerk erhalten hier die Dienste, welche als "Key Service" definiert sind.

3. **Orchestration der Services:** Dieser Schritt kombiniert die komplett implementierten, fehlerfreien Services miteinander. Auch hier sind die Spezifikationen des technischen Systementwurfs und die Definitionen der Vorbereitungen für die Integrationstests zu berücksichtigen. Der Projektleiter bestimmt für diesen Schritt ebenfalls zuständige Entwickler. Die Integrationstests stützen sich erneut auf den Einsatz der Sensoren, welche die Fehlerlokalisierung erleichtern, und sind, wie auch die Service-Tests, von zentraler Stelle ausführbar. Die Verantwortlichen werden über das Testsystem oder vom Projektleiter über gefundene Fehler informiert. Ausbesserungen führen zu einem erneuten Test aller betroffenen Komponenten und Services. Verlaufen die Tests fehlerfrei, startet die nächste Iteration im Implementierungsprozess. Externe Services werden bei Bedarf oder am Ende des Orchestrationsvorgangs integriert, diese sollten aber ebenfalls an Sensoren gebunden und Verantwortliche bestimmt werden. Sind alle Dienste komplett funktionsfähig und miteinander kombiniert, folgt ein Sprung auf den vorletzten Schritt des Programmierprozesses.

4. **Letzte Ausbesserungen an der Service-orientierten Architektur:** Wurden alle Services und deren Orchestration durch die Tests für fehlerfrei befunden, folgt der Systemtest. Der Projektleiter bestimmt einen Verantwortlichen zur Entwicklung des Systemtests (eine Kombination aus Unit- und Sensorentests), welcher die Service-orientierte Architektur bezüglich der spezifizierten Business Requirements überprüft. Hierbei sind die Spezifikationen des funktionalen Systementwurfs und die Definitionen der Vorbereitungen für den Systemtest zu beachten. Verläuft dieser ebenfalls fehlerlos, kommt es zum Abnahmetest. Werden jedoch Anomalien im Testlauf entdeckt, müssen die Fehler identifiziert und die Verantwortlichen verständigt werden. Dies kann wiederum automatisiert durch das Testsystem oder durch den Projektleiter/Testleiter erfolgen.

5. **Letzte Änderungswünsche durch den Kunden:** Im nachfolgenden Abnahmetest überprüft der Projektleiter gemeinsam mit dem Kunden ob die durch den Auftraggeber festgelegten Anforderungen an die Service-orientierte Architektur eingehalten wurden. Kommt es zu Abweichungen oder kleineren Änderungswünschen, werden die Verantwortlichen über die Anpassungen informiert, diese in den Komponenten/Services implementiert und die benötigten Testphasen erneut durchlaufen. Sind jedoch große Veränderungen/Erweiterungen am System durchzuführen, startet der Prozess wiederum in der Dokumentationsphase bei der Festlegung der Business Requirements. Die neue Iteration sichert die Aktualität der gesamten Dokumentation und Spezifikation, des Weiteren bleibt die strukturierte Architektur des Systems erhalten.

Erneuter Komponententest

Dieser Schritt folgt dem Konzept der testgetriebenen Entwicklung und ist eng mit dem ersten Punkt des Implementierungsvorgangs (Implementierung der Komponenten) verknüpft. Es soll neben korrekter Funktionalität bezüglich der Anforderungen an die einzelnen Komponenten auch SOA Governance und Security Tests durchgeführt werden.
Wie bereits im letzten Kapitel (Programmierung & Änderungen) erwähnt, müssen hier die Komponentenspezifikationen und Definitionen für die Vorbereitung der Komponententests berücksichtigt werden. Neben der Fehlerlosigkeit der einzelnen Komponenten muss auch deren Zusammenspiel garantiert werden. Hierfür werden zusammenhängende Komponenten nach und nach in der nächsten Phase des Testprozesses auf korrekte Funktionalität bezüglich ihrer Zusammenarbeit überprüft. Es werden so viele Testiterationen durchlaufen, bis das Ergebnis fehlerfrei ist. Änderungen verlangen einen Regressionstest.
Verlaufen alle Tests fehlerfrei ist auch eine Automation der Tests möglich, da Komponententests einfachen Unit-Tests entsprechen und sich hierfür eine Testautomation anbietet.

Erneuter Servicetest

Alle benötigten Komponenten und deren Sensoren stellen nach der Orchestration ein idealerweise fehlerfrei laufendes Service dar.
Die absolute Fehlerfreiheit eines Dienstes wird durch Tests mit Hilfe der integrierten Sensoren garantiert. Relevante Testdaten für das Service werden von den Sensoren überwacht und an das Monitoring-Backend weitergeleitet. Der Test vergleicht und verifiziert anschließend die erhaltenen Daten mit den zuvor spezifizierten, erwarteten Testresultaten. Der Vergleich darf keine Abweichungen zur Spezifikation finden.
Wie beim erneuten Komponententest wird auch hier die Methode der "Test-Driven Development" eingesetzt. Sie ist somit ein Teil des Programmierungspunktes (Implementierung eines Service) und erfüllt ebenfalls den Zweck die korrekte Funktionalität des Dienstes festzustellen.
Auch hier werden zusätzlich SOA Governance Tests und Sicherheitstests durchlaufen, welche die Einhaltung der unternehmensinternen Richtlinien und die Sicherheit der Daten für den jeweiligen Dienst garantieren sollen. Key Services werden vermehrt getestet, da ihre mögliche Wiederverwendung gesichert werden muss, eine hohe Qualität dieses Schlüsseldienstes wird gefordert. Die Spezifikation des Services und die Vorbereitungsdefinitionen für den Service Test müssen auch

hier berücksichtigt werden. Veränderungen am Service verlangen einen Regressionstest.
Eine Testautomation ist mit Hilfe der richtigen Testwerkzeuge und mittels Einsatz des Sensorenkonzeptes auch zu bewerkstelligen, deren Verwendung bleibt aber der ausführenden Testperson überlassen.

Integrationstest

Der Integrationstest ist mit dem fünften Punkt der Implementierung (Orchestration der Services) verknüpft. Sind alle Services korrekt miteinander verknüpft muss deren korrekte Zusammenarbeit überprüft werden. Die Spezifikationen des technischen Systementwurfs sind hierbei zu kontrollieren und zu überprüfen.
Sollten sich Schnittstellen ändern, müssen Integrationstests erneut ausgeführt werden. Dies entspricht im weitesten Sinne ebenfalls dem Konzept des "Regression-Testings".
Die Einhaltung der Unternehmensstandards (SOA Governance) und der Security Tests sollen die Sicherheit der Unternehmensdaten garantieren.
Integrationstests sind schwer zu automatisieren, es wird ein manueller Testvorgang empfohlen.

Systemtest

Wurden alle Services mit ihren Sensoren korrekt integriert, wird ein Systemtest durchgeführt, der die Service-orientierte Architektur mit dem funktionalen Systementwurf vergleicht. Werden hier Abweichungen entdeckt, führt dies zu Ausbesserungen auf der jeweiligen Ebene mit anschließendem Regressionstest. Dies wird im Programmiervorgang unter Punkt sechs (Letzte Ausbesserungen an der SOA) behandelt.
Neben abschließendem Überprüfen der Einhaltung der Unternehmensrichtlinien müssen auch hier Security Tests durchgeführt werden. Des Weiteren müssen Security Penetration Tests die Qualität der Sicherheit der Service-orientierten Architektur feststellen.
Auch hier wird ein manuelles Fortschreiten bei Tests empfohlen.

Abnahmetests

Beim Abnahmetest wird gemeinsam mit dem Kunden geprüft ob die Anforderungen an die Service-orientierte Architektur wunschgemäß erfüllt wurden. Der siebte und letzte Punkt des Programmiervorgehens (Letzte Änderungswünsche durch Kunden) beschreibt detailliert die Vorgehensweise bei Änderungen: Große Änderungen/Erweiterung müssen mit einer erneuten Iteration des Prozessmodells, beginnend bei der erneuten Identifikation der Business Requirements, umgesetzt werden.

Abschließende Bemerkungen

Der Prozess empfiehlt dem Kunden periodische "Security Penetration Tests" um auch im laufenden Betrieb und bei der Entdeckung neuer Sicherheitslücken die Sicherheit der Daten des Unternehmens garantieren zu können. Aus dem Monitoring der Daten sollen periodisch

Berichte generiert werden, die dem Management die optimale Funktionalität garantieren oder Schwachstellen (Bottlenecks, fehlerhafte Verarbeitungen) im System aufzeigen.

4.5 Architektur eines Monitoring-Frameworks

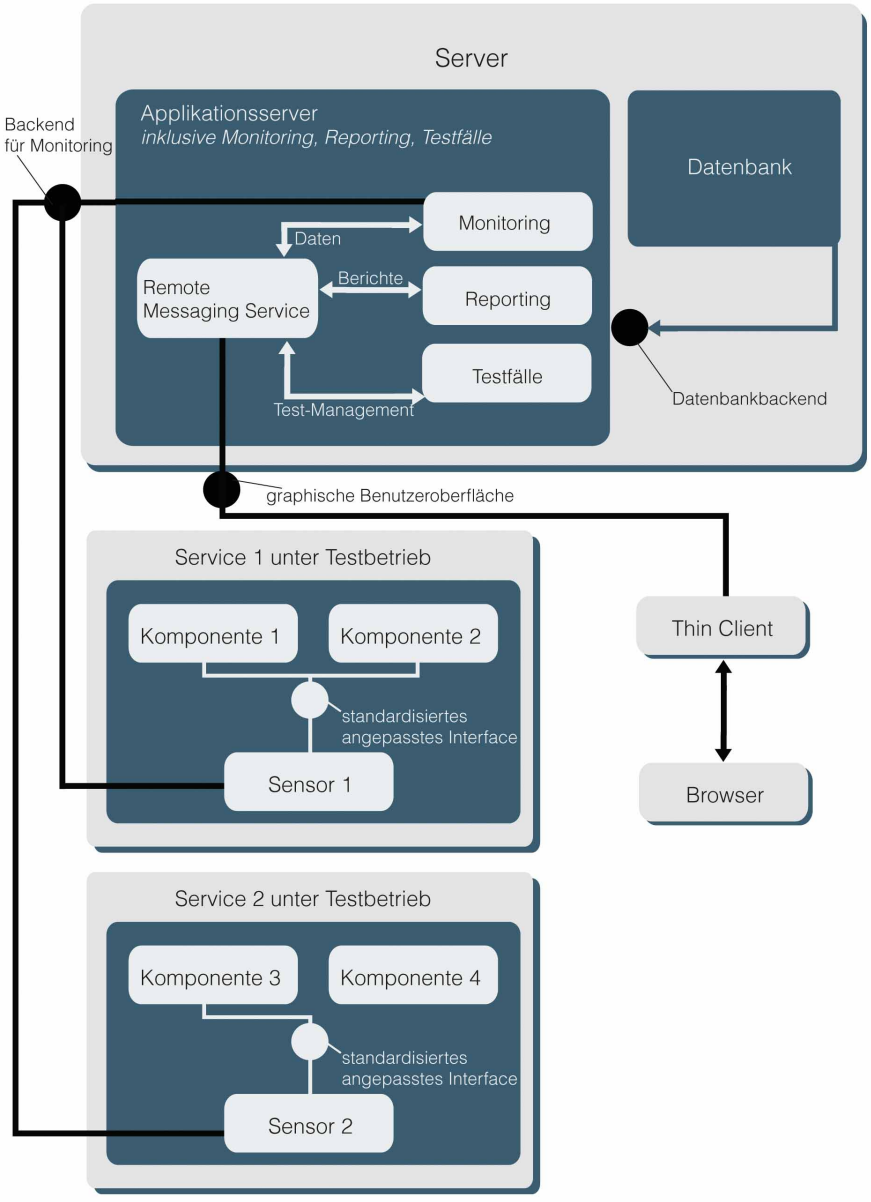

Abbildung 4.4: Architektur für ein Framework für effizientes SOA Monitoring

Die Abbildung 4.4 zeigt den beispielhaften Aufbau einer Architektur für effizientes Monitoring der Implementierung der SOA. Der Einsatz des Konzeptes, welches einen elementaren Bestandteil der Ergebnisse dieser Arbeit darstellt, soll beschrieben und veranschaulicht werden. Die einzelnen Komponenten der Abbildung 4.4 werden nachfolgend beschrieben und ihre Zusammenarbeit über die jeweiligen Backends detailliert erläutert:

- Server: Der Server besteht in diesem exemplarischen Aufbau aus zwei Bestandteilen, dem Applikationsserver und der Datenbank (inklusive Backend zum Application Server). Außerdem wird die Kommunikation zwischen den Services inklusive der Anbindung zum Monitoring mit Hilfe eines Backends ermöglicht. Des Weiteren bietet er für Clients eine graphische Benutzeroberfläche an.

- Applikationsserver: Der Applikationsserver übernimmt den Empfang der Daten für das Monitoring und generiert daraus Berichte für das Reporting. Dies geschieht mit Hilfe eines "Remote Messaging Services" (Beispiel: Web-Service implementiert mit Spring), der die Kommunikation mit den einzelnen Diensten gewährleistet. Auch die Ausführung der Testfälle kann bei Testautomation vom Applikationsserver durch Zuhilfenahme des Remote Messaging Services gelöst werden. Zur permanenten Speicherung relevanter Datensätze wird eine Datenbank verwendet. Dies geschieht über das Backend für die Datenbank.

- Datenbankbackend: Dies entspricht der Schnittstelle zwischen Applikationsserver und der Datenbank in der Illustration und ermöglicht somit die Kommunikation zwischen diesen beiden Komponenten.

- Datenbank: Die Datenbank (beispielsweise MySQL Server) ist für die permanente Speicherung der Monitoring-Daten, Reports und der applikationsbezogenen Daten verantwortlich.

- Services unter Testbetrieb: Die Services unter Testbetrieb sind in dieser Abbildung mehrfach (Service 1 und Service 2) vorhanden um zu demonstrieren, dass völlig unterschiedliche Services mit der Hilfe von gänzlich ungleichen Sensoren in der Service-orientierten Architektur existieren und überwacht werden können. Ein Service besteht hier aus mehreren Komponenten, welche über ein standardisiertes und für den jeweiligen Einsatz entsprechend angepasstes Interface mit dem Sensor verbunden sind. Dieser sendet die relevanten Daten, welche durch die Sensorenspezifikation gegeben und durch die Konfigurationsdatei angepasst sind, an das Monitoring Backend. Jeder gesendete Datensatz enthält auch den eindeutigen Komponentennamen zur Identifikation in der Monitoring-Umgebung. Dies gewährleistet die mehrfache Verwendung eines Sensors für mehrere Komponenten. Natürlich kann auch eine Komponente mehrere Sensoren durch Verwendung der Schnittstellen verwenden. "Service 2" zeigt außerdem, dass nicht alle Komponenten der Service-orientierten Architektur zwingend mit einem Sensor verknüpft werden müssen, das Sensorenkonzept ist flexibel einsetzbar.

- Backend für Monitoring: Das Monitoring-Backend empfängt die Daten und leitet sie direkt an die verarbeitende Monitoring-Komponente des Applikationsservers mit Hilfe des "Remote Messaging Service" weiter.

- Graphische Benutzeroberfläche: Eine graphische Benutzeroberfläche ist eine mögliche Schnittstelle für Benutzerein- und ausgabe für die Service-orientierte Architektur. Sie soll mit Hilfe eines ThinClients erreicht werden.

- Thin-Client: Dieser Client besteht nur aus den unbedingt notwendigen Systemdiensten um die Kommunikation zwischen dem Applikationsserver und einem Browser oder einer lightweight Applikation zu gewährleisten. Es sollen die verfügbaren Ressourcen schonend verwendet werden, denn somit werden minimale Anforderungen an die Hardware der Clients gestellt.

4.6 Auswahl von Dimensionen für SOA Reporting

Es wurde bereits definiert, wie Daten für ein Monitoring der Service-orientierten Architektur vom Service über den Sensor zum Applikationsserver gelangen. Die Frage, wie der Application Server aus diesen Datensätzen relevante Berichte erstellt, wird nun im Laufe dieses Kapitels geklärt.
Für effizientes Reporting von Service-orientierten Architekturen müssen Metriken passend für die Testdimensionen definiert werden, die für das jeweilige Zielsystem als sinnvoll anzunehmen sind. Beispiele solcher Dimensionen inklusive ihrer Metriken für das Reporting beim WASI-Modell sind:

- Definition und Identifikation eines Fehlers: Ein Fehler in der Service-orientierten Architektur entspricht einem unerwarteten Systemverhalten. Die Identifikation dieser Anomalien erfolgt durch Tests (mit und ohne Hilfe der Sensoren), welche zuvor spezifizierte Testwerte mit den durch die Überprüfungen ermittelten Ergebnissen des Systems vergleichen.

- Metrik für alle Dimensionen: Die Zahl "0" bedeutet, dass keine Fehler in der jeweiligen Dimension existieren, Die Zahl (hier in Variablenform) x bedeutet, dass es x Fehler im jeweiligen Bereich gibt, wobei x mathematisch größer 0 definiert ist. Die Zahl "-1" definiert die identifizierte Fehlerfreiheit obwohl gleichzeitig durch manuelle Überprüfung Probleme in der Komponente/Service/System festgestellt wurden.

- Identifizierte Fehler im System absolut: Es soll die momentane Gesamtfehleranzahl der gesamten Service-orientierten Architektur dokumentiert werden. Werden keine Fehler durch die Tests gefunden, wird manuell die Erreichbarkeit des Systems (beispielsweise durch den simplen Aufruf des Web Services und Ausführung eines Anwendungsfalls aus dem funktionalen Systementwurf) überprüft und zwischen den zwei folgenden Fällen unterschieden:

  - Fehlerfreiheit der Service-orientierten Architektur: Der Anwendungsfall funktioniert, die Fehlerzahl beträgt "0", es wird von absoluter Fehlerfreiheit ausgegangen.
  - Unerreichbarkeit des Systems oder Scheitern des Anwendungsfalls: Ist das System nicht erreichbar oder scheitert der Anwendungsfall, wird die Fehleranzahl auf "-1" gesetzt. Das ist ein Indiz, dass die erstellten Testfälle unzureichend für die Überprüfung der Service-orientierten Architektur sind, beziehungsweise die bestehenden

Tests überarbeitet werden muss um den Fehler auch automatisiert entdecken zu können.

- Durchschnittliche Fehlerrate je Service pro Tag/Woche/Monat: Die Fehleranzahl für ein bestimmtes Service soll für einen gewissen Zeitraum festgelegt werden. Diese Analyse soll Services, die vermehrt fehlerhaft arbeiten, identifizieren.

- Durchschnittliche Fehler im System pro Tag/Woche/Monat: Aus den historischen Daten der durchschnittlichen Fehlerraten je Service soll für eine gegebene Periode die Gesamtfehleranzahl in der SOA angegeben werden. So kann auf die Effizienz und den Nutzen des Testens/des Änderungsmanagements geschlossen werden.

- Festgestellte Fehler pro Sensor: Hiermit kann die Effizienz eines Sensors überprüft werden. Auch hier soll ein aktueller Report mit Hilfe von historischen und/oder aktuellen Monitoring-Daten möglich sein.

- Security Leaks: Die Sicherheitslöcher werden mit der in der Sektion "Guideline für Sicherheitstests" des Kapitels "Prozess zur Qualitätssicherung: Das WASI-Modell" vorgestellten Vorgehensweise identifiziert und dokumentiert. Diese Dimension gibt die Anzahl der gefundenen Sicherheitsfehler an.

- Zeit bis zur Behebung eines Fehlers: Diese Dimension gibt die Zeit im Format dd:hh:mm (d.h. sie folgt nicht der oben definierten Standardmetrik!) bis zur Behebung eines Fehlers an. Auch hier kann wieder die durchschnittliche Zeit aller Fehlerbehebungszeiten ermittelt werden.

- Status der einzelnen Services: Überprüfung der Lauffähigkeit der einzelnen Services - beispielsweise kann die Frage beantwortet werden wieviel Prozent der Services in einem Cloud-System fehlerfrei arbeiten.

Diese Liste entspricht nur einer kleinen Auswahl von möglichen Dimensionen, der Kreativität beim Reporting sind kaum Grenzen gesetzt. Nur auf die Erfüllbarkeit durch die Auswertung der verfügbaren Monitoring-Daten sollte bei der Bereichsdefinition für das Berichtswesen geachtet werden.
Der Application-Server hat dann mit Hilfe einer entsprechenden Implementierung die Aufgabe aus den Monitoring-Daten diese (und weitere sinnvolle) Dimensionen in einem Bericht zusammenzufassen.

4.7 Beispiele für den Einsatz der Sensoren

Als Abschluss des Kapitels über den Entwicklungsprozess sollen potenzielle Einsatzmöglichkeiten für das im WASI-Modell vorkommende Sensorenkonzept beschrieben werden.
In den folgenden Sektionen werden Beispiele angeführt, wie durch die Verwendung von Sensoren bestehende Geschäfts- und Industrieumgebungen nachhaltig verbessert werden können. Hierfür wird in jedem Unterkapitel die jetzige Situation im jeweiligen Umfeld erläutert, eine mögliche

Integration in die Betriebsumgebung erklärt und die sich daraus ergebenden Verbesserungen dokumentiert.

Amazons EC2 Cloud Service

Die "Amazon Elastic Compute Cloud" (kurz: Amazon EC2) ist ein über ein Web-Interface erreichbares Service, welches sich je nach Rechenleistungsbedarf frei skalieren lässt. Innerhalb weniger Minuten lassen sich Server konfigurieren und verteilte Systeme realisieren. Wird für komplexe Kalkulationen mehr Rechenleistung benötigt, werden einfach über die Web-Oberfläche neue Server eingebunden [Ama].
Amazon wirbt mit hoher Zuverlässigkeit, kompletter Kontrolle über die Server (inklusive root-Rechten), Flexibilität und hoher Sicherheit für das Cloud Service. Die Verfügbarkeit der eigenen Umgebung wird durch die Verteilung der Server auf verschiedene geographische Standorte garantiert und Unternehmen können ihre bestehenden Infrastrukturen über "Virtual Private Networks" (VPN) von Zugriffen aus dem Internet geschützt erweitern [Ama].

Beschreibung der momentanen Situation: Amazon bietet zusätzlich zum EC2 Service einen Dienst zur Überwachung der Ressourcen und Applikationen innerhalb der eigenen Umgebung an. Beispielsweise können die Performance sowie die CPU- und Speicherauslastung mit Hilfe der "Amazon CloudWatch" für jeden Rechner überwacht werden [Ama]. Lösungen für effizientes Testen und Monitoring des eigenen verteilten Systems existieren nicht.

Integration eines Sensors: Standardisierte Sensoren, welche sich über Konfigurationsdateien beliebig an die eigenen Anforderungen anpasssen lassen, können in bestehende Anwendungen mit Hilfe von Schnittstellen in die Cloud integriert werden. Darüber hinaus wäre eine Kooperation der "Amazon Cloudwatch" und der Sensoren inklusive Reporting-Funktionen über ein einziges Web-Services ebenfalls realisierbar. Java wäre auch hier aus Interoperabilitätsgründen eine passende Lösung bei der Auswahl einer Programmiersprache.

Verbesserungen durch den Sensoreneinsatz: Die Cloud-Services wurden um Monitoring- und Testmöglichkeiten für die Anwendungen der Kunden erweitert. Des Weiteren wird dem Unternehmen nicht nur die Ausfallsicherheit der Server sondern auch die Qualitätssicherung der eigenen Applikation garantiert.

Der "Amazon Elastic Cloud Computing" Dienst stellt eine interessante Ausprägung eines verteilten Systems dar, weil hier der Einsatz des Sensorenkonzepts beträchtliche Vorteile für die Kunden bringen kann. Die zentrale Überwachung der Testdaten mittels Web-Interface während der Laufzeit kann realisiert werden, jedoch ist von hohem Aufwand auszugehen. Das reine Angebot zur manuellen Integration der Sensoren, welche von Amazon zur Verfügung gestellt werden, in die eigene Cloud-Umgebung würde jedoch auch schon einen Qualitätsgewinn für das Service mit sich bringen.

Smartphones

Moderne Smartphones besitzen eine hohe Rechenleistung und verfügen über verschiedenste Möglichkeiten der Datenanbindung (von Wireless-LAN über Bluetooth bis UMTS/LTE-Funktechnologie). Sie sind somit potentielle Kandidaten für die Ausführung von verteilten Diensten.
Die Implementierung eines verteilten Systems für Smartphones stellt aufgrund der weiten Verbreitung der Betriebssysteme Android und IOS kein großes Problem dar, weil sich die Heterogenität der eingesetzten Technologien in Grenzen hält.

Beschreibung der momentanen Situation: Zur Zeit existieren kaum Lösungen zur Nutzung eines verteilten Systems auf Smartphones. Zu geringe Akkukapazitäten aktueller Mobiltelefone sowie hohe Preise für derzeitige High-End Geräte stellen noch Hindernisse dar, die aber mit der rasch fortschreitenden Entwicklung sicherlich bald behoben werden. Ausfallsicherheit kann nur durch die Replikation der Dienste auf verschiedenen Geräten gewährleistet werden.

Integration eines Sensors: Wird die Entwicklung von verteilten Systemen auch für Smartphones interessanter, kann mit Hilfe des WASI-Modells eine ausreichende Spezifikation, Implementierung, Überprüfung und Überwachung des Systems gewährleistet werden. Die Sensoren können über standardisierte Schnittstellen an die Komponenten des mobilen Services gebunden werden und unterstützen ein effizientes Monitoring während der Laufzeit. Die Leichtigkeit des Sensors genießt hier besondere Priorität, da die Rechenleistung begrenzt ist und die Ausführung der Applikation auf dem Smartphone nicht beeinträchtigt werden sollte. Für Android-Systeme ist die Integration besonders einfach, da die Entwicklung von Anwendungen auf der Programmiersprache Java basiert. Auch die Portierung eines Java-Sensors auf IOS (Programmiersprache: Objective-C) ist aufgrund der Ähnlichkeit der Sprachkonzepte leicht möglich.

Verbesserungen durch den Sensoreneinsatz: Durch effizientes Reporting kann die Ausfallsicherheit und Zuverlässigkeit des verteilten Systems sichergestellt werden. Bei der Verwendung verschiedener Dienste, welche über Schnittstellen zusammenarbeiten, gestaltet sich die Fehleridentifikation wesentlich einfacher, weil bereits während der Laufzeit der Status und die Aktivitäten jedes Service überwacht werden können.

Militärische Zwecke

Moderne Methoden der Kriegsführung bauen sehr stark auf den Einsatz von Computertechnologien auf. Der Eurofighter Kampfjet, wie ihn beispielsweise das österreichische Bundesheer einsetzt, besteht aus einer Vielzahl von technischen Systemen ohne welche die Navigation des Fliegers nicht mehr möglich wäre.
Durch die zunehmende Vernetzung von Einheiten ist der Gedanke an ein verteiltes System für militärische Einsätze durchaus legitim. Faktoren wie Skalierbarkeit, Flexibilität und Ausfallsicherheit der Entwicklung spielen dann eine entscheidende Rolle für den Erfolg der Operation. Grund genug die jetzige Situation zu betrachten und einen möglichen Einsatz von Sensoren in diesem Gebiet zu durchleuchten.

Beschreibung der momentanen Situation: Durch die strenge Geheimhaltung militärischer Entwicklungen sind Details über die Vernetzung von Kampfeinheiten nicht bekannt. Die US-amerikanische Armee prägt jedoch den Begriff der "netzwerkzentrierten Kriegsführung", welche die Überlegenheit bei Operationen gegen den Feind sichern soll. Auch die deutsche Bundeswehr setzt auf die Vernetzung von Jeeps, Panzern, Schiffen und Flugzeugen bei der Ausführung von Aktionen.

Integration eines Sensors: Unter der Annahme, dass jede Entität einer Armee als Dienst des verteilten Systems anzusehen ist, kann man sich Gedanken über die Integration des Sensorenkonzeptes, welches über Schnittstellen an die einzelnen Komponenten gebunden wird, machen. Die Zeitkomplexität muss bei der Einführung eines Sensors explizit berücksichtigt werden, jeder einzelne Dienst und die Kommunikation sind echtzeitfähig. Auf extreme Leichtigkeit des Sensors und die Priorität militärischer Aktionen gegenüber dem Monitoring von Daten muss geachtet werden. Die Implementierung hängt von der jeweiligen Umgebung ab.

Verbesserungen durch den Sensoreneinsatz: Die Effizienz von Einsätzen könnte durch das Sensorenkonzept analysiert und informative Berichte für die Befehlshaber erstellt werden. Die Geschwindigkeit, laufende Position und viele weitere relevante Parameter können für alle Einheiten genau analysiert und Fehler bei der Zusammenarbeit schnell festgestellt und behoben werden. Die Qualität der zunehmend automatisierten Kriegsführung könnte somit gesteigert werden.

Rettungseinsätze

Die Koordination von Einsätzen der Rettung ist eine komplexe Angelegenheit, sie basiert auf einer Benachrichtigung über Unfallereignis oder Notsituation und -ort durch Zeugen an die Behörden und einer anschließenden Dokumentation in Papierform. Diese Methode ist fehleranfällig und kann im Zweifelsfall über Leben oder Tod eines Menschen entscheiden.
In diesem Bereich wird in den nächsten Jahren der Einsatz moderner Technologien die Koordination und die Reaktionszeit maßgeblich verbessern. Die Integration des Sensorenkonzeptes in diese Systeme führt dann zu einer weiteren Qualitätssteigerung.

Beschreibung der momentanen Situation: Wurde die Rettungszentrale über ein Geschehnis informiert, muss diese die nächstgelegene Rettungsstelle darüber benachrichtigen. Hier beginnen die Schwierigkeiten, da die folgenden Fragen unter Zeitdruck beantwortet werden müssen. Welche Station liegt näher? Ist bereits ein Rettungswagen in der Nähe? Stimmt die angegebene Adresse? Haben die Einsatzkräfte auch die für die Rettungsaktion notwendige Ausrüstung?

Integration eines Sensors: Durch die Verwendung moderner Kommunikationstechnologien (komplette Vernetzung über Funk, GPS, Navigation) können die aktuellen Positionen der Einsatzkräfte ermittelt und kürzeste Wege gefunden werden. Die Zeitersparnis kann Leben retten. Durch den Einsatz von Sensoren können Rettungsaktionen laufend überwacht werden und deren Effizienz mit Hilfe von detaillierten Berichten analysiert werden. Das

Ziel ist die Implementierung eines zuverlässigen zeitkritischen verteilten Systems: Jeder Rettungswagen stellt hierbei ein Service dar, die Integration eines Sensors, welche über Schnittstellen an die jeweiligen Komponenten gebunden werden, garantiert die laufende Übermittlung der aktuellen Position per GPS. Auch die Meldung des Unfalls kann mit Hilfe eines GPS-Signals deutlich schneller und akkurater durchgeführt werden. Die Umsetzung kann in jeder beliebigen Programmiersprache erfolgen, auf eine gute Skalierbarkeit muss jedoch geachtet werden.

Verbesserungen durch den Sensoreneinsatz: Durch den Einsatz moderner Technologien inklusive einer Monitoring- und Reporting Lösung können die Reaktionszeiten und Wege deutlich reduziert, Fehler vermieden und mehr Menschenleben gerettet werden. Als logische Konsequenz dieser Maßnahmen kommt es außerdem zu beachtlichen Ressourcen- und Kosteneinsparungen. Die besonders wichtige, laufende Überprüfung der Funktionalität des gesamten Systems wird ebenfalls durch das Sensorenkonzept abgedeckt.

Durch die Integration moderner Kommunikationstechnologien können Rettungseinsätze optimiert werden, das Konzept der Sensoren kann hier bei der Umsetzung maßgeblich behilflich sein.

Automobilindustrie

Die Automobilindustrie setzt auf ständige Verbesserungen bestehender Konzepte. Das Schwinden der natürlichen Rohölvorkommen und die Maßnahmen der Regierungen gegen die globale Erwärmung führen zu stetig steigenden Preisen an den Tankstellen. Umso wichtiger wird das Thema "Effizienter Kraftstoffverbrauch moderner Motoren" und Konzepte zur weiteren Konsumreduktion fossiler Brennstoffe wie beispielsweise die "Start-Stop-Automatik".

Beschreibung der momentanen Situation: Die Konzepte werden in den jeweiligen Werken der Autohersteller entwickelt, umgesetzt und im Labor erprobt. Oft werden auch mehrere Testwägen an Mitarbeiter verteilt um den praktischen Einsatz zu erproben. Auch die Daten über die Nutzung der Endkunden werden in modernern Automobilen gesammelt und bei jedem Service an den Hersteller zur Auswertung geschickt. Die neueste Autogeneration unterstützt auch die Vernetzung mit Hilfe von WLAN, jedoch werden dabei derzeit noch keine Daten an die Hersteller übermittelt, die Technologie dient nur der Befriedigung von Kundenwünschen.

Integration eines Sensors: Bei zunehmender Vernetzung der Automobile können Daten über den Kraftstoffverbrauch mit Hilfe des Sensorenkonzepts gesammelt, an den Hersteller übermittelt und ausgewertet werden. Der Sensor wird in das "Embedded System" des jeweiligen Autos mit Hilfe von Schnittstellen integriert und schickt über Funk oder WLAN die Monitoring-Daten. Als Programmiersprache wird "Assembler" oder "C" vorgeschlagen, je nachdem, welches Konzept die Zielplattform verwendet.

Verbesserungen durch den Sensoreneinsatz: Durch die laufende Auswertung der Motorendaten, welche von den Kunden freiwillig übermittelt werden, und einer anschließenden Berichterstellung können Entwickler bestehende Konzepte analysieren, Fehler erkennen und ausbessern sowie Verbesserungen leichter umsetzen.

Die Auswertung von Motordaten war nur ein mögliches Beispiel für die Automobilindustrie, natürlich können mit dem Sensorenkonzept auch Aspekte wie Reifendruck, automatische Parksysteme, Aerodynamik und viele weitere Anwendungen effizient bei der alltäglichen Nutzung evaluiert werden.

Computerspielindustrie

Moderne Computerspiele besitzen eine jahrelange Entwicklungszeit mit zigtausenden Zeilen Code und Produktionskosten, die inzwischen jene mancher Hollywood-Filme übersteigen. Durch die hohe Komplexität ist die Fehlersuche oft mühsam.
Auch Online-Pokersysteme erfreuen sich in den letzten Jahren an steigender Popularität. Das Verhindern möglicher Betrugsversuche, wie beispielsweise die Automation des Spiels durch Verwendung eines Bots und anderer illegaler Methoden zur Gewinnmaximierung, stellt eine große Herausforderung an die Entwickler dar.
Beide oberhalb angeführten Probleme lassen sich mit Hilfe des Sensoreneinsatzes lösen.

Beschreibung der momentanen Situation: Finden Personen während des Spiels Fehler können sie diese an das Entwicklungsstudio melden und wenn Notwendigkeit besteht, wird der Fehler nach einer aufwendigen Suche mit einem Patch behoben. Bei Online-Poker Systemen wird das Verhalten an den Tischen laufend analysiert und mit Referenzdaten verglichen. Werden Ähnlichkeiten zu gängigen Werkzeugen zur Spielautomation entdeckt, wird der jeweilige Spieler verwarnt.

Integration eines Sensors: Dies bedeutet, dass Online-Poker Systeme bereits Sensoren nutzen, über die Implementierung ist aus Sicherheitsgründen nichts bekannt. Bei klassischen Spielen mit aufwendiger 3D-Grafik hingegen können Sensoren mit Hilfe von standardisierten Interfaces leicht in die bestehenden Komponenten integriert werden und detaillierte Debugging-Informationen bereits während der Laufzeit an die Entwickler schicken.

Verbesserungen durch den Sensoreneinsatz: Diese Daten werden gesammelt und ausgewertet, häufige Abstürze des Spiels an der gleichen Stelle im Programmcode deuten auf einen schwerwiegenden Fehler hin. Die Fehleridentifikation ist hierbei überflüssig, da die Sensoren schon die benötigten Informationen über die abnormale Termination des Programms gesendet haben. Fehler können somit schneller, effizienter und mit erheblich reduziertem Kommunikationsaufwand behoben werden.

Überwachung von Atomkraftwerken

Katastrophen wie der Super-GAU von Tschernobyl oder Fukushima werfen Fragen über die aktuellen Überwachungs- und Sicherheitsmaßnahmen von laufenden Atomreaktoren auf. Die europaweite Diskussion über die Abschaltung aller Kernkraftwerke ist berechtigt, jedoch ist eine Umrüstung auf alternative Energiequellen nicht so schnell umsetzbar, wie es vielfach gewünscht wird.
Dieses Beispiel soll zeigen, wie ein europaweites Frühwarnsystem, welches das Sensorenkonzept nutzt, atomare Unfälle vermeiden kann.

Beschreibung der momentanen Situation: In den Fällen von Tschernobyl oder Fukushima wurden die Unfälle und deren Ausmaße oft dementiert oder verharmlost um die weltweite, mediale Aufmerksamkeit zu reduzieren und die Menschen nicht zu beunruhigen. Die verherrend hohen Strahlungswerte wurden erst sehr spät publiziert und die Kraftwerke zuvor jahrelang nicht gewartet und geäußerte Sicherheitsbedenken ignoriert.

Integration eines Sensors: Durch die verpflichtende Partizipation aller Kernkraftwerke an einem Überwachungs- und Sicherheitsprogramm können Katastrophen durch vorzeitige Alarmierung und Reaktion vermieden werden. Die Idee besteht in der permanenten Überwachung aller physikalisch relevanten Daten jedes Reaktors. Werden Grenzwerte überschritten, werden automatisch Sicherheitsbeauftragte informiert, die sich der Missstände annehmen. Die gesammelten Werte sollen über den Sensor mehrmals täglich an einen zentralen Applikationsserver geschickt werden. Die Integration der Sensoren erfolgt geschickt über Schnittstellen, welche die laufende Überwachung des Systems gewährleisten. Die Programmiersprache hängt von der jeweiligen Zielplattform ab.

Verbesserungen durch den Sensoreneinsatz: Durch die frühzeitige Erkennung von Sicherheitsmängeln durch permanentes Monitoring aller relevanten Betriebswerte für jedes Kraftwerk können Katastrophen verhindert werden. Die Abschaltung aller Reaktoren dürfte noch einige Zeit dauern, da der Energiebedarf in unserer Gesellschaft ständig wächst und derzeitige alternative Stromquellen nicht die notwendige Kapazität liefern können. Die Sicherung der Qualität der bestehenden Atomkraftwerke ist somit essentiell.

Überwachung von Lagerbeständen

Große Lagerbestände von Warenhäusern sind ohne die Verwendung technischer Hilfsmittel nicht mehr verwaltbar. Die Masse an Artikeln erfordert den Einsatz maschineller Zählmethoden. Oft ist die Nachfrage nach Artikeln größer als das Angebot und Nachbestellungen werden beim Hersteller zu spät aufgegeben oder überhaupt vergessen. Die daraus resultierenden Lieferschwierigkeiten verärgern die Kunden und lassen sie zur Konkurrenz wechseln.
Die laufende Überprüfung der Lagerstände kombiniert mit automatisierter Nachbestellung von Produkten mit hoher Nachfrage kann mit Hilfe des Sensorenkonzepts umgesetzt werden. Die nachfolgenden Zeilen beschreiben die Vorgehensweise.

Beschreibung der momentanen Situation: Die Anzahl an Artikeln wird in vielen Lagern zwar technisch verwaltet und dokumentiert, eine automatisierte Nachbestellung von Produkten, die temporär ausverkauft sind, gibt es aber nur bei wenigen Supermärkten und Warenhäusern. Außerdem besteht bei der Automation das Problem, dass die Bestellmenge ohne vorheriger Regressionsanalyse unter Einbeziehung historischer Daten nicht gut geschätzt werden kann.

Integration eines Sensors: Die Sensoren überwachen die Lager und liefern alle benötigten aktuellen und vergangenen Verkaufszahlen sowie die Lagerinformationen. Ein gutes Programm ermittelt aus diesen Daten die für die Zukunft zu erwartende Absatzmenge. Je nach Größe und Verteilung der Lager können die Services und Komponenten anders

verteilt werden. Die Integration der Sensoren geschieht wiederum über die Verwendung von standardisierten Schnittstellen, welche die Anbindung an die Komponenten sicherstellen. Durch Konfigurationsdateien kann ein einmal programmierter Sensor an die gewünschten Szenarien angepasst werden, die Wiederverwendung ist somit gewährleistet.

Verbesserungen durch den Sensoreneinsatz: Die Lagerverwaltung wird durch die permanente Überwachung der Lagerbestände und die automatisierte Nachbestellung von ausverkauften Produkten gewährleistet. Die notwendigen Arbeitsschritte werden reduziert und Personal sowie Kosten eingespart.

Gesundheitsüberwachung

Viele Menschen sind auf die konstante Überwachung ihrer Gesundheitswerte angewiesen. Beispielsweise überprüfen viele Personen mehrmals täglich ihren Blutdruck und Zuckerkranke messen ihren Blutzuckerspiegel.
Hilfsmittel zur Ermittlung dieser Werte gibt es in großer Anzahl, jedoch nutzen wenige Entwicklungen moderne Kommunikationsmethoden und die Lösungen sind meist nicht automatisiert. Der häufige Besuch von Ärzten bleibt den Patienten somit nicht erspart.
Die Verwendung moderner Funktechnologie und der Einsatz des Sensorenkonzeptes können den ohnehin beschwerlichen Alltag schwerkranker Personen erleichtern. Moderne, bestehende Lösungen übertragen die Gesundheitswerte bereits mit Hilfe von UMTS/WLAN, jedoch werden nur die Daten an den jeweiligen Server übertragen. Eine Möglichkeit zur Feststellung des Status, beziehungsweise einer effizienten periodischen Überprüfung der korrekten Funktionalität des Gerätes kann mit Hilfe des WASI-Modell garantiert werden. Der "Well-Assisted SOA Implementation" Prozess hilft signifikant bei der Implementierung dieser nützlichen Merkmale.
Im folgenden Beispiel wird die Situation eines alten Menschen mit Herzrythmusstörungen analysiert und die Vorteile durch die Benutzung moderner Entwicklungen, die das Sensorenkonzept integrieren, erläutert.

Beschreibung der momentanen Situation: Der Patient hat von seinem Facharzt den Auftrag bekommen dreimal täglich im Abstand von 4h seinen Blutdruck zu messen und die Werte in eine vorgefertigte Tabelle einzutragen. Der alte Mann lebt alleine in seiner Wohnung, hat des öfteren Schwindelanfälle und Angst in seiner Wohnung einen Herzinfarkt zu erleiden.

Integration eines Sensors: Ein spezielles Armband, welches per Funk oder Wireless-Lan an ein zentrales Monitoring System angebunden ist, kontrolliert mit Hilfe des integrierten Sensors laufend die Herzfrequenz des Patienten und misst in zuvor definierten Zeitabständen den Blutdruck des Mannes. Diese Daten werden dann zum Server geschickt und dort protokolliert. Kommt es zu Herzrhythmus-Störungen wird sofort der Notarzt durch das Senden der aktuellen GPS-Daten über den Notfall informiert. Die Integration eines speziell angepassten Sensors kann für aktuelle, kommunikationsfähige Messgeräte aller Art einfach durch die Verwendung einer Schnittstelle umgesetzt werden.

Verbesserungen durch den Sensoreneinsatz: Der Einsatz des Sensorenkonzeptes für Messgeräte im Gesundheitsbereich erspart dem Patienten viele Arztbesuche und die Übermittlung der aktuellen Werte erfolgt automatisch. Der Arzt kann sie sich dann jederzeit und

überall anschauen. Des Weiteren kann die korrekte Funktionalität des Gerätes jederzeit überprüft werden und die laufenden Monitoring-Daten dem Hersteller bei der Erkennung von Fehlfunktionen behilflich sein. Außerdem können durch die automatische Benachrichtigung des Notarztes viele Menschenleben gerettet werden, weil schnell und effizient gehandelt werden kann.

Videoüberwachung

Die Überwachung der Geschehnisse mit Hilfe von Videokameras verteilt sich auf verschiedenste Anwendungsgebiete: U-Bahn-Stationen, Firmengelände und Museen sind nur einige Beispiele hierfür. Das Videomaterial wird aufgezeichnet oder gespeichert und kann bei Bedarf jederzeit analysiert werden. Teilweise werden die observierten Orte von Sicherheitsbeauftragten auf verschiedenen Bildschirmen "live" beobachtet.
Des Weiteren existieren Algorithmen zur automatischen Gesichtserkennung, beispielsweise können damit polizeilich gesuchte Menschen identifiziert werden. Hierbei kann die Integration des Sensorenkonzeptes die Überwachung automatisieren und die laufenden Personalkosten senken.

Beschreibung der momentanen Situation: Wie bereits erwähnt, wird das Videomaterial entweder aufgezeichnet, gespeichert oder von Personen überwacht. Gesichtserkennungsalgorithmen helfen bei der Identifikation von Personen. Diese Vorgänge werden zukünftig vermehrt automatisiert ablaufen, die Verwendung des Sensorenkonzeptes ist hierfür eine mögliche Lösung.

Integration eines Sensors: Die Sensoren können in bestehende Videoüberwachungssysteme integriert werden. Ein kleiner, vernetzter Computer mit Kamera und einem laufenden Softwaresystem kann als Service eines verteilten Systems betrachtet werden, welcher die Integration eines Sensors durch Verwendung von standardisierten Schnittstellen ermöglicht. Die Videodaten werden durch die Sensoren laufend analysiert und bei der Erkennung eines Gesuchten mit Angabe von Zeit und Ort an das zentrale Überwachungssystem weitergeleitet.

Verbesserungen durch den Sensoreneinsatz: Polizeilich gesuchte Personen können mit dieser Vorgangsweise schnell identifiziert und noch schneller gefunden werden. Dieses Konzept lässt sich adaptiert auch für die Verkehrsüberwachung einsetzen, nur der Algorithmus für die Erkennung wird getauscht.

Fazit für die Einsatzmöglichkeiten

Wie man an wenigen Beispielen sieht, sind die Möglichkeiten für den Einsatz der Sensoren und einer effizienten Service-orientierten Architektur mit inkludiertem Monitoring zur Laufzeit sehr vielfältig: Neue verteilte Systeme, die mit Hilfe des WASI-Modells erstellt werden, können viele bestehende Gegebenheiten deutlich vereinfachen und verbessern. Bestehende Umgebungen können allerdings auch häufig einfach um das Sensorenkonzept erweitert werden, die Qualität der weiteren Entwicklung wird erheblich gesteigert.

Vielfach können bestehende Geschäftsprozesse verbessert oder Industrieumgebungen durch den Einsatz von Sensoren, welche laufend Daten des betrachteten Systems liefern, bereichert werden. Die Fehlersuche wird vereinfacht und die Verbesserung bestehender Konzepte gefördert. Im Gesundheitsbereich kann die Integration von Sensoren in verschiedenen Situationen sogar Menschenleben retten.

4.8 Proof of Concept: WASI-Modell

Dieser Abschnitt präsentiert die Ergebnisse der Analyse der Einsatzfähigkeit des "Well-Assisted SOA Implementation" (WASI) Modells. Für diesen Zweck wurde das Beispiel "Gesundheitsüberwachung"' aus dem Kapitel "Beispiele für den Einsatz der Sensoren" für eine theoretische Implementierung mit Hilfe des WASI-Prozess herangezogen und Schritt für Schritt ausgearbeitet. Die folgenden Subkapitel orientieren sich dabei am WASI-Vorgehensmodell und demonstrieren dessen Verwendung in der Praxis.

Business-Requirements

Der Auftraggeber ist ein großer Pharmaziekonzern, der seine Produktpalette um das Gerät "Advanced Health Monitoring System" (AHMS) für eine automatisierte, zuverlässige und permanente Überwachung der Herzfrequenz eines Patienten erweitern will. Zusammen mit den Zuständigen dieses Unternehmens wird die gewünschte Funktionalität analysiert und dokumentiert:

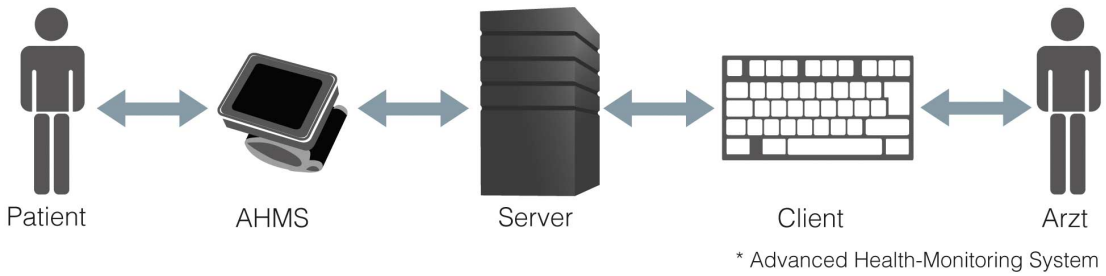

Abbildung 4.5: Skizze für "Advanced Health Monitoring System"

1. Überblick verschaffen: Die Skizze 4.5 wurde zusammen mit dem Kunden erstellt und zeigt die gewünschte Funktionalität. Ein Patient soll den ganzen Tag ein Messgerät tragen, welches automatisiert permanent die Herzfrequenz misst und mit Hilfe von Funktechnologie die gewonnenen Daten an einen zentralen Server sendet. Der Arzt, der den Patient betreut, soll mit Hilfe eines Clients jederzeit die Gesundheitswerte von dieser Datenbank abrufen können.

2. Funktionale & technische Anforderungen: Die Anforderungen wurden identifiziert, analysiert und mit Hilfe von UML-Anwendungsfalldiagrammen illustriert. Ein Requirement-Diagramm aus dieser Liste wird als Beispiel in Abbildung 4.6 veranschaulicht.

- Permanente Messung der Herzfrequenz: Priorität 10/10
- Temporäre Speicherung der Daten auf dem Gerät: Priorität 10/10
- Senden der Daten über WLAN/UMTS: Priorität 10/10
- Zentrale Speicherung der Daten auf einem Server: Priorität 10/10
- Abrufen der Daten mit einem Thin-Client: Priorität 10/10:
- Sicherheit - Authentifizierung & Autorisierung: Priorität 10/10
- Zuverlässigkeit des Gerätes: Priorität 10/10

3. Zeitplan: Die geschätzte Projektdauer beträgt sechs Monate für ein kleines Team bestehend aus einem Projektleiter, zwei Programmierern und einem Testleiter, wobei folgende Meilensteine identifiziert wurden:

   - Implementierter Server: zur Speicherung der Daten, für die Authentifizierung und Autorisierung, für eine effiziente Überwachung der Zuverlässigkeit des Gerätes und für den Zugriff auf die ermittelten Messdaten. Deadline - 2 Monate nach Projektstart
   - Fertigstellung des Dienstes: Die periodische Messung der Blutdruckwerte, eine temporäre Speicherung auf dem Gerät und der Versand der Daten muss fehlerfrei funktionieren. Ein Status-Monitoring des Gerätes muss gewährleistet sein. Deadline - 4 Monate nach Projektstart
   - Fertigstellung des Thin-Clients: Der Zugriff auf das bereitgestellte User-Interface muss über einen Browser möglich sein. Deadline - 5 Monate nach Projektstart

4. Testplan: Die folgenden Kategorien werden im Testplan berücksichtigt sowie im Laufe des Projektes ständig aktualisiert und erweitert: Komponententests, Servicetest, Integrationstest, Systemtest und Abnahmetest. Für die Fertigstellung der Meilensteine müssen die jeweiligen, relevanten Tests fehlerfrei terminieren. Für die Erstellung und Ausführung der Testfälle wird das "jUnit"-Testframework verwendet.

5. SOA Governance: Der Pharmakonzern wünscht die Einhaltung einer 99-prozentigen Ausfallsicherheit für den Server sowie die besondere Berücksichtigung einer strengen Sicherheitsrichtlinie für den Zugriff auf die sensiblen Datensätze der Patienten. Diese Punkte wurden dokumentiert und im weiteren Entwicklungsverlauf beachtet.

6. Risikomanagement: Die Projektrisiken wurden identifiziert, eine Auswahl davon wird in den folgenden Zeilen präsentiert:

   - Verletzung der SOA-Governance: Die Entwicklung erfüllt die Anforderungen, jedoch wurde die strenge Sicherheitslinie nicht korrekt implementiert. Geschätzte Schadenshöhe: Totalverlust durch Klagen. Eintrittswahrscheinlichkeit: 3 Prozent. Priorität: 10/10
   - Ausfall eines Teammitglieds: Ein Mitglied verlässt aus verschiedensten Gründen das Team, es muss Ersatz gesucht werden. Das Projekt kann nicht in vorgegebener Zeit abgeschlossen werden. Geschätzte Schadenshöhe: 10 bis 25 Prozent des Auftragvolumens. Eintrittswahrscheinlichkeit: 25 Prozent. Priorität: 9/10

- Gesamtaufwand wurde unterschätzt: Die Zeit für die Fertigstellung wurde erheblich unterschätzt, weitere Ressourcen werden benötigt. Geschätzte Schadenshöhe: 10-20 Prozent des Auftragsvolumens. Eintrittswahrscheinlichkeit: 20 Prozent. Priorität: 8/10

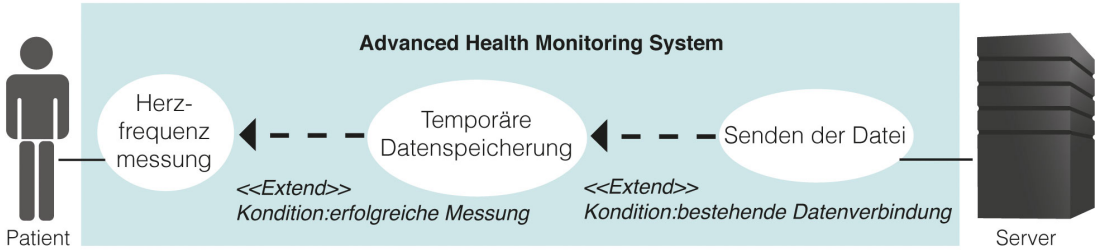

Abbildung 4.6: Anwendungsfalldiagramm für "Advanced Health Monitoring System"

Vorbereitung Abnahmetest

Die im Schritt "Business Requirements" identifizierten Anforderungen werden in einer Checkliste angeführt und in den Testplan integriert. Gemeinsam mit dem Kunden wird nach der Implementierung des Gesamtsystems die Zufriedenheit des Auftraggebers für jeden einzelnen Punkt bewertet.
Außerdem wird ein Punkt für "Security Penetration Tests" eingeführt und das Vorgehen gemäß der "Security Guideline" des WASI-Modells geplant. Für das "Advanced Health Monitoring System" muss die Autorisierung und Authentifizierung von Server und Client besondere Beachtung erhalten und entsprechende Tests vorbereitet werden.

Funktionaler Systementwurf

Ein kompletter Überblick über das AHM-System wird geschaffen. Hierfür wurde mit Hilfe der "Business Process Modelling Notation" (BPMN) der Ablauf modelliert (vereinfacht dargestellt in Abbildung 4.7), anschließend die Services identifiziert und Schlüsseldienste festgelegt.
Folgende Services wurden identifiziert (allesamt Schlüsseldienste):

- Service zur Messung, Speicherung und Versand der Gesundheitswerte: Das Gerät soll permanent die Herzfrequenz des Patienten messen, temporär auf dem Gerät speichern und an den Server weiterschicken.

- Service zum Empfang und Speicherung der Daten: Auf dem Server muss ein Dienst laufen, der die Daten empfängt und mit einer eindeutigen Identifikationsnummer in einer Datenbank speichert. Unterschreitet die Herzfrequenz des Patienten einen bestimmten Wert in zwei hintereinanderfolgenden Messungen wird der Notarzt verständigt.

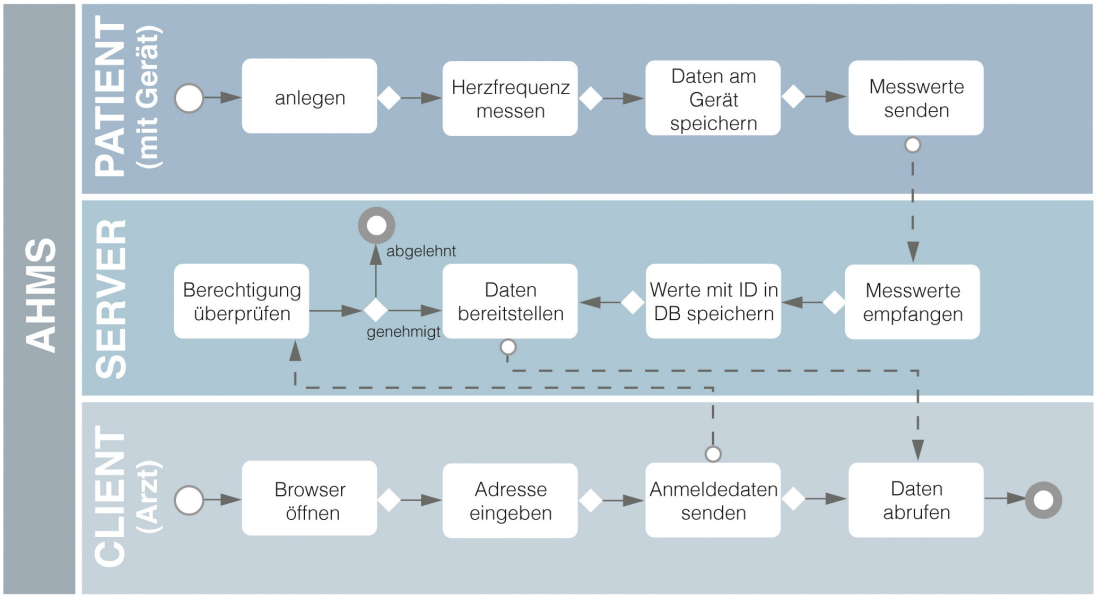

Abbildung 4.7: Vereinfachtes BPMN-Diagramm für "Advanced Health Monitoring System"

- Service für Authentifizierung und Autorisierung: Eine Website mit einer Anmeldemaske muss vom Service bereitgestellt werden. Die Anmeldedaten, welche durch den Client mit Hilfe eines Browsers gesendet werden, müssen von diesem Service überprüft werden. Die Identität und die Korrektheit der Benutzerdaten muss festgestellt werden.

- Service zur Bereitstellung der Daten für Clients: Die Daten müssen für autorisierte Clients in Form einer Homepage abrufbar sein. Hierbei muss beachtet werden, dass der jeweilige Arzt nur die Daten seiner Patienten und nicht alle Daten einsehen kann.

Vorbereitung Systemtest

Im Testplan werden die Services dokumentiert und Vorbereitungen zur finalen Überprüfung der implementierten Lösung vorgenommen. Hierfür werden für das "Advanced Health Monitoring System" zwei Probanden ausgewählt, die das System jeweils einen Tag testen. Für die Überprüfung der Notarztverständigung wird eine Kooperation mit einer Rettungszentrale eingegangen, die über die korrekte Funktionalität im Ernstfall berichtet. Alle diese Aktivitäten werden detailliert im Testplan festgehalten.
Zusätzlich wird mit Hilfe eines geplanten Load-Test (d.h. das System unter hoher Belastung mit vielen gleichzeitigen Zugriffen) die Ausfallsicherheit überprüft. Die Authentifizierungs- und Autorisierungsmechanismen werden ebenfalls auf korrekte Funktionalität getestet um die Einhaltung der geforderten Richtlinien durch das Unternehmen zu gewährleisten (SOA Governance Testing).

Technischer Systementwurf

Das Zusammenspiel der Services, welche im funktionalen Systementwurf identifiziert wurden, wird analysiert. Die Technologien werden festgesetzt und die Dienste in Komponenten aufgespaltet.
Die Analyse fand keine unerwünschten Abhängigkeiten zwischen den einzelnen Services. Alle Services werden im Laufe der Implementierung durch Kommentare als Schlüsseldienste markiert. Das Entwicklungsteam besitzt fundierte Kenntnisse über die Programmiersprache Java, welche somit auch als Sprache für die Implementierung ausgewählt wird. Als Kommunikationstechnologie für die Kommunikation zwischen den einzelnen Services wird das "Simple Object Access Protocol" (kurz: SOAP) eingesetzt. Zur Speicherung und Abfrage der Daten wird "MySQL" verwendet. Für die Konfigurationsdateien wird XML als Standard vorgeschrieben.
Anschließend werden die einzelnen Services in Komponenten unterteilt. In dieser Arbeit wird

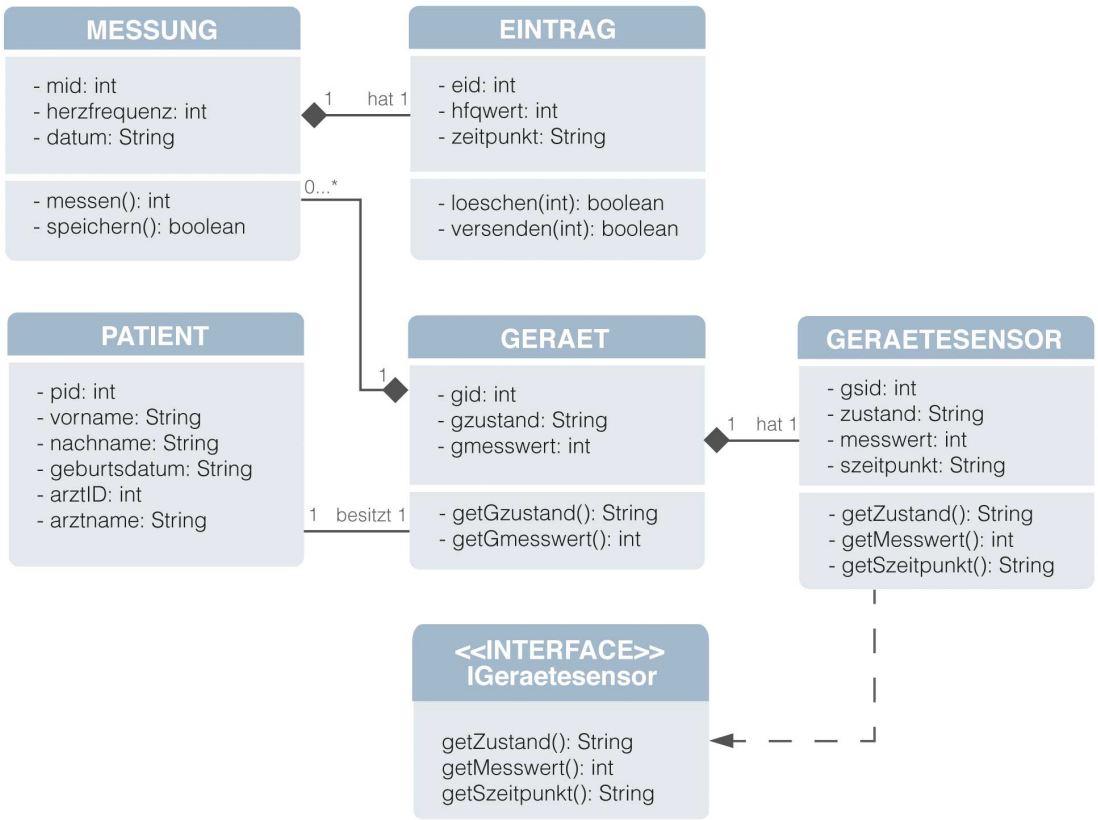

Abbildung 4.8: Klassendiagramm eines AHMS-Service

der Dienst "Messung, Speicherung und Versand der Gesundheitswerte" als Exempel mit Hilfe eines Klassendiagramms in einzelne Komponenten gespaltet werden. Die Abbildung 4.8 zeigt dieses Diagramm: Das Interface "Geraetesensor" wird verwendet um den Sensor zu implemen-

tieren, welcher den aktuellen Zustand und Messwert des Gerätes überwacht. Dieser holt sich die Daten vom Gerät mit Hilfe der Methoden "getGzustand()" und "getGmessung()". Ein Gerät kann beliebig viele Messungen vornehmen, wobei jede Messung temporär auf dem Speicher des Messgeräts als Eintrag mit allen relevanten Daten gesichert wird. Des Weiteren gibt es einen Patient, der genau ein Gerät besitzt.

Die Schnittstellen zwischen den Services werden ebenfalls im technischen Systementwurf

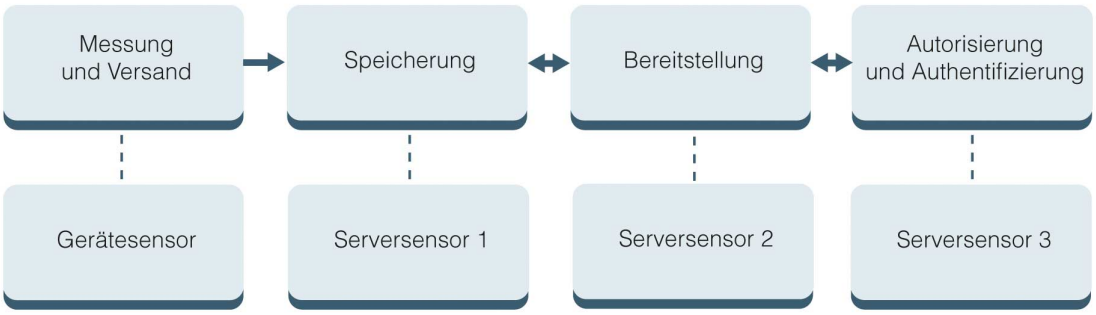

Abbildung 4.9: Schnittstellen zwischen den AHMS-Services

spezifiziert. Die Skizze 4.9 illustriert das gewünschte Zusammenspiel: Das Service "Messung, Speicherung und Versand der Gesundheitswerte" sendet die Daten zum Dienst "zum Empfang und Speicherung der Daten" zum Server. Das Service zur "Bereitstellung der Daten für die Clients" arbeitet mit dem "Authentifizierungs- und Autorisierungsdienst" und dem Dienst zur "Speicherung der Daten auf dem Server" zusammen. Nur bei genehmigter Berechtigung sollen spezifische Patientendaten für den jeweiligen Arzt zur Verfügung gestellt werden.
Auch die Integrationspunkte der Sensoren werden in der Skizze 4.9 für jedes Service spezifiziert: Jedes Service soll mit Hilfe eines Sensors überwacht werden, Details hierfür werden erst bei der Sensoren- und Servicespezifikation gegeben.

Vorbereitung Integrationstests

Für die Vorbereitung der Integrationstests wird das im technischen Systementwurf spezifizierte Zusammenspiel zwischen den einzelnen Services mit Hilfe von Tests abgedeckt. Das AHMS-Testsystem soll mit "jUnit"-Tests unter Zuhilfenahme der Sensoren die Kooperation zwischen den Diensten testen. Das "Advanced Health Monitoring System" besteht nur aus Key Services, somit müssen diese Tests fehlerfrei terminieren. Die geplanten Überprüfungen werden im Testplan festgehalten.
Alle Services, die mit dem "Authentifizierungs- und Autorisierungsdienst" zusammenarbeiten, müssen zusätzlich auf Sicherheit überprüft werden. Entsprechende jUnit-Tests werden im Testplan festgehalten. Die Ausfallsicherheit bei der Kommunikation zwischen den Services wird mit "Load-Tests" realisiert. Alle genannten Aktivitäten werden im Testplan dokumentiert.

Sensorenspezifikation

Im ersten Schritt werden die relevanten Daten für jeden Dienst identifiziert und die benötigten Sensoren definiert werden:

- Messung & Versand: Hier ist der Messwert und der Status des Gerätes von Interesse, ein "Gerätesensor" wird benötigt.
- Empfang und Speicherung der Daten auf dem Server: Hier sollen die empfangenen Daten zu jedem Zeitpunkt sowie der Status des Dienstes jederzeit abrufbar sein. Da das Service auf dem Server läuft, wird es der Einfachheit halber als "Serversensor1" bezeichnet.
- Bereitstellung der Daten: Hier soll die Verfügbarkeit des Services durch die Sensoren laufend überprüft werden. Hierfür wird der "Serversensor 2" verwendet.
- Autorisierung & Authentifizierung: Die empfangenen Anmeldedaten und der Status des Dienstes auf dem Server soll abfragbar sein. Dies wird mit dem "Serversensor 3" bewerkstelligt.

Für alle Services wird die Programmiersprache Java verwendet. Die Schnittstellen werden so entworfen, dass die relevanten Daten der Sensoren mit Hilfe von Methoden einfach abgerufen werden können. Die Überprüfung der Ausfallsicherheit und der Sicherheitsaspekte des Systems wird bei der Erstellung der Sensoren ebenfalls berücksichtigt.
Des Weiteren soll für jeden Sensor eine eigene XML-Konfigurationsdatei erstellt werden, die eine Anpassung der Abfragezeit für die relevanten Daten ermöglicht.

Service-Spezifikation

Die Spezifikation der einzelnen Services wurde bereits zu großen Teilen im technischen Systementwurf umgesetzt. Das Diagramm 4.8 zeigt bereits ein Beispiel für die Unterteilung eines Dienstes in entsprechende Komponenten.
Die Programmiersprache Java und die Unabhängigkeit der einzelnen Services voneinander wurde ebenfalls bereits spezifiziert. Für die Integration der Sensoren werden jeweils relevante Komponenten der Services ausgewählt und die Sensoren mit Schnittstellen eingebunden.

Erstellung der Service-Level-Tests

Vor der eigentlichen Implementierung werden für jedes Service Tests erstellt, welche die spezifizierte Funktionalität abbilden sollen. Hierfür werden die Sensorenwerte für die Überprüfungen verwendet.
Um auf das Beispiel für den Dienst "Messung & Versand der Gesundheitswerte" zurückzukommen, werden folgende Testfälle für die Überprüfung des Services spezifiziert:

- Simulation eines Puls mit 75 Schlägen pro Minute: Der Herzschlag wird simuliert und liefert einen konstanten Wert für das Gerät. Der gewonnene Messwert mit dem spezifizierten Testwert von 75 verglichen. Hierbei dürfen keine Abweichungen gefunden werden.

- Die Daten werden im Testsystem über SOAP an ein Test-Remote-System geschickt, welches die Zahl 75 korrekt empfangen muss.

Für alle Services wird ein ähnliches Vorgehen vorgeschlagen.

Komponenten-Spezifikation

Den einzelnen Komponenten, ihren Methoden und Variablen werden Namen gegeben, die den Namenskonventionen der Programmiersprache Java entsprechen: Klassennamen beginnen mit Großbuchstaben, Methoden und Variablen mit Kleinbuchstaben, Konstanten hingegen bestehen nur aus Großbuchstaben.
Die einzelnen Komponenten werden spezifiziert und ihre Funktionsweise beschrieben, ihre Schnittstellen angegeben und die Integration der Sensoren definiert.
Auch hier wird auf das Beispiel "Dienst für Messung und Versand" verwiesen und die einzelnen Komponenten hierfür angegeben:

- Klasse Patient: Der Patient soll mit eindeutiger ID, Vorname, Nachname, Geburtsdatum, ID des zuständigen Arztes und dem Namen dieses Arztes im System erscheinen. Jeder Patient hat genau ein Gerät zur Messung der Herzfrequenz.

- Klasse Geraet: Das Gerät soll eine eindeutige ID, einen Zustand und einen Messwert haben. Die beiden letztgenannten Werte sollen jederzeit abfragbar sein. Jedes Gerät soll genau einem Patient zugeordnet sein.

- Klasse Messung: Eine Messung hat eine eindeutige ID, einen Wert für die Herzfrequenz und das Datum der Messung. Diese Werte müssen mit Methoden abgefragt werden können. Beliebig viele Messungen werden einem Gerät zugeordnet.

- Klasse Eintrag: Ein Eintrag hat eine eindeutige ID, den Wert der Herzfrequenzmessung und den Zeitpunkt der Messung. Des Weiteren sollen Methoden zum Löschen des Eintrags auf dem Gerät mit Hilfe der Eintrags-ID sowie der Versand der Daten über SOAP an weitere Services möglich sein. Eine Messung hat genau einen gespeicherten Eintrag auf dem Gerät.

- Sensor Geraetesensor: Der Gerätesensor wird über eine Schnittstelle mit genau einem Gerät verbunden. Er ist spezifiziert mit einer eindeutigen ID und der Zustand, der aktuelle Messwert und der Zeitpunkt der Messung können jederzeit abgefragt werden. Die Implementierung soll über ein zuvor spezifiziertes Interface erfolgen, welches den Sensor spezifiziert.

Erstellung der Komponententests

Die einzelnen Komponenten werden unabhängig voneinander mit jUnit-Tests überprüft. Gibt es Abhängigkeiten zwischen Komponenten werden diese mit Hilfe von Dummy-Objekten simuliert. Im Beispiel "Messung und Versand der Werte" kann beispielsweise die Klasse Messung getestet werden. Durch den Aufruf der Methode "messen()" werden die Gerätedaten abgerufen. Dadurch, dass das Gerät nicht verfügbar ist, wird ein Dummy-Objekt implementiert, was einen fixen

Messwert zurückliefert. Der Testwert muss dann mit dem geholten Dummy-Wert übereinstimmen. Als Beispiel wird hier eine Liste einiger Testfälle für das Service "Messung und Versand der Werte" angegeben:

- Messung01 (M01T): Die Überprüfung der Methode "messen()" kontaktiert ein Dummy-Objekt, welches das Gerät spezifiziert und einen fixen Messwert zurückgibt. Dieser muss mit dem erwarteten Testwert übereinstimmen.
- Messung02 (M02T): Ein Testfall, der die Methode "speichern()" der Klasse Messung zur temporären Speicherung auf dem Gerät testet. Muss den Wert "true" zurückgeben.
- Eintrag01 (E01T): Die Methode "loeschen(1)" soll den Dummy-Eintrag Nummer 1 löschen und es wird der Wert "true" erwartet.
- Eintrag02 (E02T): Der Aufruf der Funktion "versenden(1)" soll den ersten Dummy-Eintrag an den Server schicken, es wird der Wert "true" als Ergebnis erwartet.
- Gerät01 (G01T: Der Zustand des Gerätes soll getestet werden. Je nach Zustand ("eingeschaltet, "ausgeschaltet", "reagiert nicht") soll "getGzustand()" diesen Zustand als String zurückgeben (Tests durchführen mit Gerät im jeweiligen Status für korrekte Testausführung).
- Gerät02 (G02T): Die Methode "getGmesswert()" soll den ermittelten Herzfrequenzwert des Gerätes zurückgeben. Dieser kann auf einen konstanten Wert von 75 simuliert werden, dieser Wert wird dann vom Test erwartet.

Die Tabelle 4.8 weist als Beispiel jedem identifizierten Testfall einen, keinen (oder in manchen Fällen) mehrere Sensoren nach Bedarf zu. Dabei bedeutet "-", dass kein Sensor benötigt wird, und "X", dass der jeweilige Sensor für den Testfall verwendet wird.

-	**M01T**	**M02T**	**E01T**	**E02T**	**G01T**	**G02T**
GeraeteSensor	-	-	-	-	X	X
ServerSensor1	-	-	-	-	-	-
ServerSensor2	-	-	-	-	-	-
ServerSensor3	-	-	-	-	-	-

Tabelle 4.1: Beispiel - Benötigte Sensoren für die Testfälle

Verteilte Implementierung inklusive der Testphasen

Der WASI Implementierungs- und Testprozess gliedert sich für das "AHM"-System in folgende Punkte:

1. Implementierung der Komponenten: Unter den Entwicklern werden die zuvor erstellten Komponententests aufgeteilt. Jeder Programmierer implementiert dann seine Komponente/n, bis alle Tests fehlerfrei terminieren. Die testgetriebene Entwicklung wird hierbei mittels des "erneuten Komponententests" (bis zur Fehlerfreiheit aller Komponenten) umgesetzt. Der Status der Implementierung jeder Komponente kann vom Projektleiter über die zentralisierte Testumgebung eingesehen werden.

2. Implementierung des Services: Sind alle Komponenten fehlerfrei, werden Verantwortliche zur Kombination der Klassen zu einzelnen Services bestimmt. Ist dieser Vorgang beendet, wird für jedes Service der zuvor implementierte Service-Level-Test aufgerufen ("Erneuter Service-Level-Test"). Auch dieser Vorgang kann vom Projektleiter zentralisiert überwacht werden. Terminieren alle Tests fehlerfrei, kann zum nächsten Schritt übergegangen werden.

3. Orchestration der Services: Die einzelnen Services ("Messung & Versand", "Empfang und Speicherung", "Bereitstellung der Daten", "Authentifizierung und Autorisierung") werden miteinander kombiniert und die Integrationstests erstellt ("Integrationstests"). Hierbei werden die Vorbereitungen für die Integrationstests inkludiert. Diese Tests sollen die Überprüfungen automatisieren und stützen sich auf den Sensoreneinsatz zur leichteren Fehlerlokalisierung. Das zentralisierte Testsystem soll die erstellten Tests automatisiert ausführen und den Projektleiter über gefundene Fehler benachrichtigen. Wurden alle Services orchestriert, folgt die Phase des Systemtests.

4. Letzte Ausbesserungen an der Service-orientierten Architektur: Der "Systemtest" inkludiert die Vorbereitungen aus der Dokumentationsphase und überprüft die korrekte Implementierung der einzelnen Anforderungen. Dieser soll wieder zentralisiert und automatisiert ausgeführt werden können. Bei der Entdeckung von Fehlern werden die Verantwortlichen für die fehlerhaften Komponenten vom Projektleiter verständigt. Die letzte Phase im WASI-Prozess ist der Abnahmetest.

5. Letzte Änderungswünsche durch den Kunden: Für den "Abnahmetest" muss kein eigener Test implementiert werden, vielmehr wird mit dem Kunden nochmals die korrekte Erstellung der Software bezüglich der gemeinsam gefundenen Anforderungen überprüft und die Zufriedenheit des Auftraggebers beurteilt. Werden weitere Features gewünscht, startet der Prozess wieder bei der "Definition der Business Requirements", die gesamte Dokumentation muss dann überarbeitet und das System entsprechend aktualisiert werden. Dem Kunden wird auch eine regelmäßige Überprüfung der Sicherheit (die Sicherheitstests wurden im Systemtest integriert) vorgeschlagen.

Fazit: Proof of Concept

Die Verwendung des "Well-Assisted SOA Implementation" Modells wurde anhand des einfachen Beispiels analysiert. Das Ergebnis ist eindeutig: Das WASI-Modell lässt sich hervorragend zur kompletten Spezifikation, Implementierung und zur Überprüfung einer Service-orientierten Architektur verwenden. Einen klaren Kritikpunkt stellt der umfangreiche Dokumentationsaufwand dar: Würde man das "Advanced-Health Monitoring" System nicht nur exemplarisch sondern

komplett mit allen benötigten Diagrammen spezifizieren, hätte man schon für dieses kleine Beispiel eine reichlich gefüllte Dokumentationsmappe. Bei größeren Projekten ist die vollständige Spezifikation jedoch anzuraten um einen erfolgreichen Projektabschluss garantieren zu können. Service-orientierte Architekturen wachsen stetig und steigen in der Komplexität, eine komplette Dokumentation ist notwendig.

KAPITEL 5

Benchmark der Testprozesse aller Modelle

Der Testprozess des im vorherigen Kapitel definierten WASI-Modells wird nun mit allen anderen Testmethoden gängiger Modelle (siehe Kapitel: Grundlagen - Softwareentwicklungsprozesse) verglichen um die Vorteile aufzuzeigen und mögliche bestehende Schwächen am definierten Modell zu finden. Dabei soll besonderes Augenmerk auf die Verwendbarkeit für das Testen von Service-orientierten Architekturen gelegt werden.
Dieses Kapitel definiert zunächst die Parameter für die Analyse der einzelnen Modelle, zusätzlich soll eine Metrik für die Bewertung der einzelnen Kategorien festgelegt werden. Anschließend werden die Bewertungsgrundlagen definiert und jedes Prozessmodell bezüglich der zuvor definierten Parameter einzeln analysiert, quantitativ beurteilt und die vergebene Bewertungen begründet. Abschließend wird mit Hilfe des arithmetischen Mittels aller Ergebnisse für jeden einzelnen Prozess die Gesamtbeurteilung gebildet und graphisch in Form eines Diagramms dargestellt.

5.1 Parameter, Metrik & Bewertungsgrundlagen

Dieses Kapitel definiert die Parameter für einen effizienten Vergleich aller in dieser Arbeit erwähnten Entwicklungsprozesse, sowie die Metrik zur Bewertung derselben. Es werden außerdem die Grundlagen zur Bewertung erläutert.

Definition der Parameter für den Benchmark

Dieser Abschnitt definiert die erforderlichen Parameter zur Evaluierung der Testprozesse der Entwicklungsmodelle bezüglich ihrer Eignung zum Überprüfen einer Service-orientierter Architektur. Folgende zehn Dimensionen wurden nach erneuter Recherche der Artikel des Related Work Kapitels ermittelt:

- Unit-Testing: Dieser Parameter definiert, wie gut die Komponenten der Service-orientierten Architektur durch das jeweilige Prozessmodell überprüft werden.

- Service-Testing: Gibt das Maß an, wie effizient Services durch den jeweiligen Entwicklungsprozess getestet werden können.

- Integration-Testing: Diese Dimension legt fest, wie gut das Konzept der Integrationstests im jeweiligen Vorgehensmodell umgesetzt wird.

- System-Testing & Abnahme-Tests: Dieser Parameter definiert die Qualität der Beschreibung für die Durchführung von Systemtest und Abnahmetest für jedes einzelne Prozessmodell.

- Sicherheitstests & Security Penetration Testing: Das Maß der Beschreibung für die Umsetzung von Sicherheitstests während der Projektlaufzeit und der Security Penetration Tests bei Projektende für alle Prozessmodelle soll bestimmt werden.

- SOA Governance Testing: Dieser Parameter definiert die Qualität der Einhaltung der Überprüfung der unternehmensinternen Richtlinien und Standards, welche für die Service-orientierte Architektur spezifiziert wurden, für den jeweiligen Prozess.

- Regression Testing: Definiert den Grad, der für den jeweiligen Prozess Systemtests nach der Ausbesserung gefundener Fehler vorschreibt.

- Testgetriebene Entwicklung: Der Parameter beantwortet die Fragestellung ob und in welchem Ausmaß im betrachteten Prozess die testgetriebene Entwicklungsmethode verwendet wird.

- Häufigkeit der Test-Iterationen: Die Dimension der Häufigkeit verschiedenster Tests während der Entwicklungsphase wird beschrieben.

- Monitoring während der Testausführung: Die laufenden Testfälle sollen während der Ausführung der einzelnen Tests überwacht werden können um effizientes Live-Monitoring und Reporting der Testumgebung zu garantieren. Die Erfüllung dieses Punktes soll vom Benchmark erfasst werden.

Definition der Metrik

Dieser Abschnitt definiert die erforderliche Metrik zur quantitativen Bestimmung der Parameter des Benchmark-Prozesses.
Die Beurteilung erfolgt mit Hilfe eines einfachen Punktesystems, welches ganzzahlige Werte im Bereich von 0 bis 4 definiert, wobei die Zahl 0 das Äquivalent zu einer "unausreichenden", die Zahl 1 einer "genügender", die Zahl 2 einer "befriedigenden", die Zahl 3 einer "guten" und die Zahl 4 einer "sehr guten" Eignung zum Testen einer Service-orientierten Architektur entspricht. Der Punkteschnitt jedes Prozesses ist als arithmetisches Mittel aller Beurteilungen definiert.

Bewertungsgrundlagen

Ist in einer Prozessbeschreibung der Parameter nicht spezifiziert, werden 0 Punkte ("unausreichend") vergeben, weil das Modell eine ungenügende Spezifikation zum Testen einer Service-orientierten Architektur bietet.
Ein Punkt ("genügend") bedeutet, dass die Testart prinzipiell möglich ist, aber eine detaillierte Beschreibung in der Dokumentation des Testprozesses fehlt.
Zwei Punkte ("befriedigend") spezifizieren eine ausreichende Behandlung der Testart im jeweiligen Prozess, jedoch könnten zusätzliche Details gegeben werden.
Werden drei Punkte vergeben, ist von einer fast vollständigen Beschreibung des entsprechenden Parameters in der Prozessdokumentation auszugehen.
Die Beurteilung mit vier Punkten ("sehr gut") bedeutet, dass die Testdokumentation des Parameters vollständig für den jeweiligen Prozess ausgeführt ist.
Die Informationen jedes Prozesses für die Analyse wurden aus dem Kapitel "Bestehende Software-Entwicklungsprozesse" (und dessen weiterführende Quellenangaben) entnommen.

5.2 Benchmark-Ergebnisse

In diesem Abschnitt sollen die Ergebnisse der Evaluierung der Testvorgänge für die einzelnen Prozessmodelle präsentiert werden. Neben der quantitativen Bewertung aller Parameter mit der zuvor definierten Metrik soll auch eine Begründung für jeden ermittelten Wert abgegeben werden. Abschließend wird eine Gesamtbeurteilung aller evaluierten Prozessmodelle gegeben und graphisch mit Hilfe eines Diagramms dargestellt.

Bewertung: Wasserfallmodell

Das erweiterte Wasserfallmodell wurde zur detaillierten Bewertung ausgewählt, weil der einfache Wasserfallprozess in der Praxis für die Entwicklung aktueller Software kaum mehr verwendet wird.

Unit-Testing: 3 Punkte - Unit-Testing wird im erweiterten Wasserfallmodell begleitend mit der Coding-Phase spezifiziert. Durch die begleitende Durchführung dieser Überprüfung aller Komponenten während der Implementierung wird die Qualität der Service-orientierten Architektur auf dieser Ebene gewährleistet.

Service-Testing: 1 Punkt - Service-Testing wird in diesem Modell nicht explizit erwähnt, jedoch ist mit einem geschickten Arrangement von Integrationstests bestimmter Komponenten die Überprüfung der Funktionalität eines Service möglich.

Integration-Testing: 2 Punkte - Ein grundsätzlicher Integrationstest bei Fehlerfreiheit aller Komponententests wird definiert, allerdings sind die Unterschiede zwischen diesen und dem Systemtest schwer differenzierbar.

System-Testing & Abnahme-Tests: 2 Punkte - Die Durchführung eines Systemtests wird rudimentär beschrieben, Abnahmetests sind im erweiterten Wasserfallmodell nicht definiert.

Die Kunden müssen somit eventuell mit einer mangelhaften Qualität der Service-orientierten Architektur rechnen.

Sicherheitstests & Security Penetration Testing: 0 Punkte - Weder Sicherheitstests noch abschließende Security Penetration Tests werden im Prozess erläutert. Die Ausführungszeiten dieser Tests in den einzelnen Phasen sind somit auch unbekannt, was eine korrekte Überprüfung im Modell unmöglich macht.

SOA Governance Testing: 0 Punkte - Die Situation verhält sich hier wie beim vorherigen Punkt "Sicherheitstests & Security Penetration Testing".

Regression-Testing: 1 Punkt - Erneutes Testen nach der Ausbesserung von Fehlern wird im Wasserfallmodell spezifiziert, jedoch werden nur die geänderten Teile der Software und nicht die gesamte Service-orientierte Architektur getestet.

Testgetriebene Entwicklungsmethode: 0 Punkte - Die testgetriebene Entwicklungsmethode wird bei diesem Prozess nicht eingesetzt, weil sämtliche Tests erst nach der Implementierung erstellt werden.

Häufigkeit der Test-Iterationen: 1 Punkt - Das erweiterte Wasserfallmodell erlaubt erneute Testiterationen, diese werden in der Prozessbeschreibung allerdings nicht konkret angegeben.

Monitoring während der Testausführung: 0 Punkte - Eine Überwachung der Zustände und der Schnittstellen einzelner Services der Service-orientierten Architektur ist im "Waterfall Model" weder spezifiziert, noch ohne großer Erweiterung der gesamten Umgebung umsetzbar.

Bewertung: V-Modell

Das V-Modell spezifiziert für alle Tests eine "Bottom Up" Vorgehensweise, welche sich sehr gut für die Überprüfungen einer Service-orientierten Architektur eignen. Die Bewertung betrachtet alle relevanten Testpunkte im Detail.

Unit-Testing: 4 Punkte - Die Spezifikation, Dokumentation und Durchführung der Tests der einzelnen Komponenten wird vom Modell präzise beschrieben.

Service-Testing: 3 Punkte - Ein Servicetest kann als "Sub-System-Validation" im V-Modell angesehen werden. Die Vorgehensweise ist dann ebenfalls genau spezifiziert.

Integration-Testing: 3 Punkte - Häufiges Integration-Testing wird im Prozess unter dem Punkt "System Integration and Test" behandelt, eine kleine Schwäche ist hier ebenfalls die fehlende Spezifikation des Begriffs "Service".

System-Testing & Abnahme-Tests: 4 Punkte - Beide Testkonzepte werden im Prozessmodell als finale Überprüfungsschritte detailliert dokumentiert. Bei kompletter Fehlerfreiheit des Systems kann das Projekt beendet werden, sonst wird der Prozess erneut beim ersten Dokumentationsschritt gestartet.

Sicherheitstests & Security Penetration Testing: 1 Punkt - Das V-Modell definiert keine Überprüfungen der Sicherheit in den Testphasen, allerdings kann der Prozess ohne Probleme um diese Tests erweitert werden.

SOA Governance Testing: 1 Punkt - Der Prozess verhält sich hier ebenso wie bei Punkt "Sicherheitstests & Security Penetration Testing" beschrieben.

Regression-Testing: 2 Punkte - Werden Fehler gefunden, werden Teile der Service-orientierten Architektur (Komponenten/Services) neu getestet, das gesamte System jedoch nicht.

Testgetriebene Entwicklungsmethode: 0 Punkte - Die testgetriebene Entwicklungsmethode wird vom V-Modell nicht verwendet. Die Dokumentation/Programmierung erfolgt nach dem Top-Down- und die anschließende Überprüfung der Funktionalität nach dem Bottom-Up-Prinzip.

Häufigkeit der Test-Iterationen: 1 Punkt - Die Tests folgen eigentlich keinem iterativen Kreislauf, jedoch kann ein Sprung an den Projektbeginn und das erneute Durchlaufen des gesamten Prozesses als Iteration betrachtet werden.

Monitoring während der Testausführung: 0 Punkte - Im V-Modell ist ein effizientes Monitoring der Services und ihrer Schnittstellen ebenfalls nicht spezifiziert und daher kaum realisierbar.

Bewertung: W-Modell

Das W-Modell stellt eine Erweiterung des V-Prozesses dar, ein iteratives Testvorgehen wird gefördert.

Unit-Testing: 4 Punkte - Die Komponententests sind, wie auch im V-Modell, genau dokumentiert und das Vorgehen für den Testprozess definiert.

Service-Testing: 3 Punkte - Die Services können wiederum als Subsysteme definiert werden, ein effizienter Service-Test ist somit gegeben.

Integration-Testing: 3 Punkte - Die Integrationstests sind hier mit gleicher Punktezahl wie beim V-Modell bewertet, es fehlt nur die Spezifikation des Begriffs "Service".

System-Testing & Abnahme-Tests: 4 Punkte - Sowohl die System- als auch Abnahmetests sind im W-Modell vollständig erläutert und das jeweilige Vorgehen definiert.

Sicherheitstests & Security Penetration Testing: 1 Punkt - Eine grundlegende Definition fehlt innerhalb des Prozesses, allerdings lässt sich der W-Prozess um diese Tests erweitern.

SOA Governance Testing: 1 Punkt - Die grundlegende Definition ist auch hier nicht gegeben, das Überprüfen der unternehmensinternen Richtlinien kann jedoch zum W-Modell hinzugefügt werden.

Regression-Testing: 3 Punkte - Bei Änderungen werden nicht nur die unmittelbar betroffenen Teile der Architektur erneut geprüft, sondern auch andere abhängige Komponenten.

Testgetriebene Entwicklungsmethode: 1 Punkt Durch die Vorbereitung der einzelnen Tests im Vorfeld kann das W-Modell auch um eine testgetriebene Entwicklungsmethode erweitert werden. Dies wird allerdings nicht konkret spezifiziert.

Häufigkeit der Test-Iterationen: 2 Punkte Das W-Modell an sich ist kein iteratives Modell, es lassen sich dennoch wiederholte Testiterationen durchführen (wird beispielsweise in einem Service gefunden, werden natürlich auch die Komponenten erneut getestet).

Monitoring während der Testausführung: 0 Punkte - Der Prozess definiert keine Methoden zur Überwachung der aktuellen Stati der Services und Testaktivitäten während der Laufzeit.

Bewertung: Spiralmodell

Das Spiralmodell spezifiziert einen vollständig iterativen und zu gleich inkrementellen Prozess. Die Durchführung der einzelnen Tests wird in diesem Modell leider nicht näher spezifiziert, eine grundsätzliche Bewertung ist jedoch möglich.

Unit-Testing: 2 Punkte - Die Tests bei Fertigstellung der Implementierung einzelner Komponenten (d.h. am Ende der jeweiligen Iteration) werden durchgeführt. Nähere Informationen über Unit-Tests werden nicht gegeben.

Service-Testing: 1 Punkt - Der Begriff "Service" ist nicht spezifiziert, durch geschickte Anordnung der einzelnen Iterationen lassen sich aber auch Service-Tests durchführen.

Integration-Testing: 2 Punkte - Integrationstests werden im Spiralmodell für spätere Iterationen definiert, jedoch fehlt eine detaillierte Beschreibung dieser Testart.

System-Testing & Abnahme-Tests: 2 Punkte - Die System- und Abnahmetests werden im Spiralmodell auch in späteren Iterationsphasen durchgeführt, auch hier fehlt eine Dokumentation zur Vorgehensweise.

Sicherheitstests & Security Penetration Testing: 1 Punkt - Sicherheitstests sind bei diesem Prozessmodell nicht spezifiziert, jedoch lassen sie sich problemlos für bestehende Iterationen einführen.

SOA Governance Testing: 1 Punkt - Auch die Überprüfung unternehmensinterner Richtlinien ist nicht definiert, lässt sich allerdings auch einfach im Modell für geplante Iterationen einfügen.

Regression-Testing: 0 Punkte - Eine erneute Überprüfung des gesamten Systems nach Fehlerausbesserungen ist im Modell nicht vorgesehen und durch den iterativen und inkrementellen Ansatz auch schwer zu realisieren.

Testgetriebene Entwicklungsmethode: 0 Punkte - In den einzelnen Iterationen wird stets nach der Implementierung getestet, ein Einsatz der testgetriebenen Entwicklungsmethode ist nicht möglich.

Häufigkeit der Test-Iterationen: 3 Punkte - Unit-Tests werden in jeder Iteration durchgeführt, die Anzahl an Wiederholungen für Integrations- und Systemtests sind jedoch wesentlich geringer anzusiedeln.

Monitoring während der Testausführung: 0 Punkte - Das Spiralmodell definiert auch keine Überwachung der Tests während der Ausführung.

Bewertung: IBM Rational Unified Process

Der "IBM Rational Unified Process" ist iterativ und inkrementell, die einzelnen Testarten - und phasen werden sehr genau spezifiziert.

Unit-Testing: 4 Punkte - Komponententests werden im Rational Unified Process vollständig spezifiziert und im Laufe jeder Iteration ausgeführt.

Service-Testing: 2 Punkte - Der Begriff "Service" wird im RUP nicht näher spezifiziert, allerdings lassen sich einzelne Iterationen für Service-Tests definieren.

Integration-Testing: 4 Punkte - Der gesamte RUP Prozess forciert eine Integration mehrerer Komponenten und deren anschließende Validierung.

System-Testing & Abnahme-Tests: 4 Punkte - System- und Abnahmetests sind im Rational Unified Process vollständig dokumentiert.

Sicherheitstests & Security Penetration Testing: 1 Punkt - Die Überprüfung der Sicherheit der Service-orientierten Architektur wird im RUP nicht explizit beschrieben, jedoch kann diese ohne größeren Aufwand bereits in der Testplanungsphase berücksichtigt und später umgesetzt werden.

SOA Governance Testing: 1 Punkt - Die Überprüfung der SOA Governance einer Service-orientierten Architektur wird nicht näher spezifiziert, aber durch die Aufnahme von SOA Governance Informationen können diese in späteren Tests auf eine korrekte Umsetzung untersucht werden.

Regression-Testing: 2 Punkte - Wie das V-Modell sieht der RUP keine erneute Überprüfung der gesamten Service-orientierten Architektur vor, aber die geänderten Teile der Software werden erneut verifiziert.

Testgetriebene Entwicklungsmethode: 0 Punkte - Der RUP Prozess sieht für die Implementierung einen iterativen und inkrementellen Ansatz vor, die Tests werden erst nach der Programmierung erstellt.

Häufigkeit der Test-Iterationen: 3 Punkte - Komponenten- und Integrationstests werden häufig ausgeführt, jedoch sind Servicetests nicht spezifiziert.

Monitoring während der Testausführung: 0 Punkte - Vom Prozess wird kein Konzept zur Überwachung von Services zur Testlaufzeit spezifiziert.

Bewertung: Open-Unified Process

Der "Open-Unified Process" (OUP) lehnt sich am "IBM Rational Unified Process" an, außerdem kann er als offenes Framework um weitere Konzepte ergänzt werden.

Unit-Testing: 4 Punkte - Wie beim RUP ist der Test einzelner Komponenten genau dokumentiert.

Service-Testing: 2 Punkte - Der Begriff "Service" ist im "Open-Unified Process" nicht spezifiziert, auch hier können Servicetests jedoch als nachträgliche Erweiterung eingeführt werden.

Integration-Testing: 4 Punkte - Der "OUP" hat eine ausgezeichnete Dokumentation des Vorgehens bei Integrationstests.

System-Testing & Abnahme-Tests: 4 Punkte - Die System- und Abnahmetests werden im "Open-Unified-Process" genauestens beschrieben.

Sicherheitstests & Security Penetration Testing: 2 Punkt - Das Thema Sicherheit wird vom Prozess nicht behandelt, allerdings lassen sich Sicherheitskomponenten nachträglich noch zum Framework hinzufügen.

SOA Governance Testing: 2 Punkt - Die SOA Governance Testing Komponente fehlt auch im Open Unified Prozess, lässt sich jedoch auch leicht nachrüsten.

Regression-Testing: 2 Punkte - Wie auch im "IBM Rational Unified Process" werden ausgebesserte Komponenten neu getestet, jedoch fehlt eine Überprüfung des Systems.

Testgetriebene Entwicklungsmethode: 1 Punkt - Die Methode der testgetriebenen Entwicklung lässt sich durch nachträgliche Änderungen an der Verfahrensweise integrieren, wird aber für den iterativen und inkrementellen Prozess nicht empfohlen.

Häufigkeit der Test-Iterationen: 3 Punkte - Auch der Unified Process definiert keine regelmäßigen Service-Tests, sonst ist die Testspezifizierung vollständig.

Monitoring während der Testausführung: 1 Punkt - Das Konzept einer Überwachung des System- und Teststatus während der Laufzeit wird nicht vom Open-Unified Process spezifiziert, allerdings könnte mit erheblichem Aufwand das Framework um diese Technologie erweitert werden.

Bewertung: Extreme Programming

"Extreme Programming" (XP) verwendet eine testgetriebene Entwicklung für die Implementierung, forciert häufige Integrationstests und fördert den Kontakt zu den Kunden. Die Dokumentation wird so gering wie möglich ausgeführt.

Unit-Testing: 4 Punkte - Komponententests werden vor der Implementierung erstellt. Dies garantiert optimale Qualität.

Service-Testing: 2 Punkte - Service-Tests werden nicht konkret beschrieben, lassen sich aber wegen des geringen Dokumentationsaufwands leicht in den bestehenden Prozess inkludieren.

Integration-Testing: 4 Punkte - Durch die Methode des "Continous Integration Testing" wird die Qualität der Software gesichert.

System-Testing & Abnahme-Tests: 3 Punkte - Die Systemtests sind beim XP Modell nur rudimentär spezifiziert, durch den Akzeptanztest mit den Kunden aber auch nur bedingt notwendig. Eine klare Trennung dieser beiden Testarten ist allerdings nicht gegeben.

Sicherheitstests & Security Penetration Testing: 1 Punkt - Diese Tests sind im Extreme Programing Modell nicht definiert. Während der Implementierung und beim Akzeptanztest kann allerdings darauf geachtet werden.

SOA Governance Testing: 1 Punkt - Eine Überprüfung der Richtlinien des Unternehemsn ist beim XP Prozess nicht vorgesehen, kann aber für das Projekt explizit vorgegeben werden.

Regression-Testing: 1 Punkt - Die ausgebesserte Komponente wird in diesem Prozessmodell erneut getestet, eine Überprüfung der gesamten bisherigen Implementierung ist nicht vorgesehen.

Testgetriebene Entwicklungsmethode: 4 Punkte - Extreme Programming basiert auf der testgetriebenen Entwicklungsmethode.

Häufigkeit der Test-Iterationen: 4 Punkte - Häufige Iterationszyklen während des gesamten Projektes sollen die Qualität sichern.

Monitoring während der Testausführung: 0 Punkte - Ein Konzept zur Überwachung der Services und Tests während der Laufzeit ist nicht in der Prozessbeschreibung vorhanden.

Bewertung: Crystal Family

Die Crystal Family definiert kurze Iterationszyklen und Testautomatisierung, je nach Projektgröße wird eine andere Vorgehensweise vorgeschlagen.

Unit-Testing: 4 Punkte - Die Testautomatisierung aller Komponenten-Tests garantiert die Qualität auf diesem Implementierungslevel.

Service-Testing: 1 Punkt - Das Service-Konzept wird nicht spezifiziert, Service-Testing ist dennoch möglich.

Integration-Testing: 2 Punkte - Integrations-Tests werden nur für große Projekte mit vielen beteiligten Entwicklern vorgeschrieben. Für diesen Fall wird auch eine Automation der Tests vorgeschlagen.

System-Testing & Abnahme-Tests: 3 Punkte - Diese Testarten werden in der Crystal Family ausreichend beschrieben.

Sicherheitstests & Security Penetration Testing: 1 Punkt - Sicherheitstests sind in der Crystal Family nicht dokumentiert, können aber in den Prozess integriert werden.

SOA Governance Testing: 1 Punkt - Tests zur Einhaltung der Unternehmensstandards sind ebenfalls nicht in der Beschreibung der Crystal Family vorhanden, können aber zum bestehenden Projekt hinzugefügt werden.

Regression-Testing: 1 Punkte - Ein erneuter Test der gesamten Architektur bei Fehlerausbesserungen ist für die Prozessfamilie nicht definiert.

Testgetriebene Entwicklungsmethode: 1 Punkt - Die Crystal Family gibt keine konkrete Definition über zu verwendende Entwicklungsmethode, die testgetriebene Entwicklung kann mit ausreichender Planung eingesetzt werden. Dies widerspricht jedoch der Vorgehensweise eines agilen Prozesses.

Häufigkeit der Test-Iterationen: 4 Punkte - Der Prozess spezifiziert häufige Releases und Iterationen innerhalb des Projektes.

Monitoring während der Testausführung: 0 Punkte - Ein Konzept für effizientes Monitoring der Testumgebung einer Service-orientierten Architektur wird nicht spezifiziert.

Bewertung: Scrum

Scrum gibt nur Vorschläge für das Vorgehen bei der Implementierung und der Überprüfung der Software an (beispielsweise die testgetriebene Entwicklung), andere Entwicklungsprozesse wie beispielsweise Extreme Programming können auch für die Umsetzung verwendet werden. Damit gestaltet sich eine Bewertung dieses Prozesses als schwierig, Annahmen werden für einige Punkte getroffen.

Unit-Testing: 2 Punkte - Es wird von einer testgetriebenen Entwicklungsmethode ausgegangen. Damit wird die Qualität der Komponenten sichergestellt, allerdings besteht keine Beschreibung über die abschließende Testdokumentation.

Service-Testing: 1 Punkt - Das Konzept des "Services" wird nicht von Scrum präsentiert, mit etwas Kreativität des Teams lassen sich aber einzelne Sprints für Service-Tests verwenden.

Integration-Testing: 1 Punkt - Integrationstests werden ebenfalls nicht spezifiziert und müssen bei den Sprints explizit beachtet werden.

System-Testing & Abnahme-Tests: 1 Punkt - Abschließende System- und Abnahmetests sind nicht in der Prozessbeschreibung enthalten, können jedoch in die letzten Sprints integriert werden.

Sicherheitstests & Security Penetration Testing: 0 Punkte - Sicherheitstests werden für den Prozess nicht spezifiziert und sind auf Grund der extrem agilen Vorgehensweise auch schwer im jeweiligen Projekt zu realisieren.

SOA Governance Testing: 1 Punkt - Die Einhaltung der festgesetzten Unternehmensstandards werden vom "Scrum" Prozess nicht explizit festgelegt. Eine optionale Spezifikation ist möglich.

Regression-Testing: 2 Punkte - Wird die testgetriebene Entwicklungsmethode verwendet, wird bei Ausbesserungen stets die Komponente neu getestet. Eine Fehlerfreiheit der gesamten Service-orientierten Architektur kann aber nicht gewährleistet werden.

Testgetriebene Entwicklungsmethode: 4 Punkte - Scrum eignet sich zur Integration der testgetriebenen Entwicklungsmethode.

Häufigkeit der Test-Iterationen: 4 Punkte - Scrum forciert häufige Test-Iterationen in sogenannten Sprints.

Monitoring während der Testausführung: 0 Punkte - Scrum definiert kein Konzept für Monitoring einer Service-orientierten Architektur.

Bewertung: Kanban

Kanban besitzt noch eine geringere Spezifikation als der Scrum Prozess. Eine Bewertung kann nur mittels einiger Annahmen vollzogen werden.

Unit-Testing: 2 Punkte - Mit Hilfe des Einsatzes der testgetriebenen Entwicklung kann die Qualität der einzelnen Module der Service-orientierten Architektur garantiert werden.

Service-Testing: 1 Punkt - Service-Tests können auch in der Planung berücksichtigt werden, dies wird allerdings nicht vom Kanban-Vorgehensmodell spezifiziert.

Integration-Testing: 1 Punkt - Wird Kanban mit Extreme Programming kombiniert, können Integrationstests durchgeführt werden. Eine Dokumentation über Integration-Testing ist nicht vorhanden.

System-Testing & Abnahme-Tests: 1 Punkt - Ähnlich verhält sich die Situation bei System- und Abnahmetests. Diese können bei der Planung berücksichtigt werden, der reine Managementprozess spezifiziert dieses Vorgehen nicht.

Sicherheitstests & Security Penetration Testing: 0 Punkte - Die Überprüfung der Sicherheit kann in diesem agilen Prozess mit geringer Planung schwer umgesetzt werden.

SOA Governance Testing: 0 Punkte - Ein fehlender Testplan und mangelhafte Dokumentation der Service-orientierten Architektur verhindern eine effiziente Überprüfung der unternehmensinternen Richtlinien und Standards.

Regression-Testing: 2 Punkte - Bei Verwendung der testgetriebenen Entwicklung, wird bei Ausbesserungen stets die Komponente neu getestet. Ein fehlerloses System kann nicht garantiert werden.

Testgetriebene Entwicklungsmethode: 4 Punkte - Kanban eignet sich wie auch Scrum zur Integration der testgetriebenen Entwicklungsmethode.

Häufigkeit der Test-Iterationen: 2 Punkte - Kanban definiert keine fixen Zeiten für die Iterationen im Projekt. Von häufigen Testiterationen kann daher nicht ausgegangen werden.

Monitoring während der Testausführung: 0 Punkte - Kanban fehlt ebenfalls ein Konzept für Monitoring einer Service-orientierten Architektur.

Bewertung: Modellgetriebene Entwicklungsmethode

Ein modellgetriebener Ansatz zur Überprüfung der korrekten Funktionalität einer Service-orientierten Architektur stellt eine interessante Alternative zu den bisher analysierten Testkonzepten dar. Das "UML Testing Profile" (U2TP) hilft bei der Umsetzung der Tests durch eine genaue Spezifikation des gesamten Systems.

Unit-Testing: 3 Punkte - Einzelne Komponenten werden bei der modellgetriebene Entwicklung mit Hilfe des spezifizierten Testmodells verifiziert.

Service-Testing: 3 Punkte - Auch einzelne Services werden mit Hilfe des Testmodells auf korrekte Funktionalität überprüft.

Integration-Testing: 3 Punkte - Die zuvor definierten Schnittstellen werden auch automatisch mittels des U2TP-Modells kontrolliert.

System-Testing & Abnahme-Tests: 2 Punkte - Das Testmodell garantiert die Überprüfung des Systems, Abnahmetests werden allerdings nicht spezifiziert.

Sicherheitstests & Security Penetration Testing: 0 Punkte - Um Sicherheitstests zu gewährleisten müsste man den UML-Teststandard (U2TP) erweitern. Abweichungen von Standards führen jedoch früher oder später zu Problemen, weshalb davon abgeraten wird.

SOA Governance Testing: 0 Punkte - Ohne Anpassung/Erweiterung des U2TP-Standards ist auch keine Überprüfung der SOA Governance möglich.

Regression-Testing: 4 Punkte - Bei jeder Änderung wird beim nächsten Testdurchlauf erneut das ganze System auf Fehler überprüft.

Testgetriebene Entwicklungsmethode: 0 Punkte - Beim Einsatz der modellgetriebenen Entwicklungsmethode werden alle Tests erst nach der Erstellung des Quellcodes durchgeführt.

Häufigkeit der Test-Iterationen: 1 Punkt - Es wird nicht spezifiziert, wie oft Testdurchläufe während der Implementierung geschehen sollen.

Monitoring während der Testausführung: 0 Punkte - Ein Konzept zur Überwachung der Services und der Tests existiert nicht.

Bewertung: Feature Driven Development

Feature Driven Development (FDD) erstellt eine Merkmalsliste, die dann Punkt für Punkt implementiert und anschließend auch getestet wird. Die Spezifikation der FDD beschreibt allerdings Unit-, Integrations- und Systemtests.

Unit-Testing: 3 Punkte - Jedes einzelne Feature, welches in einer Service-orientierten Architektur eine Komponente darstellen kann, wird nach der Implementierung getestet.

Service-Testing: 1 Punkt - Der Begriff "Service" ist auch hier nicht definiert, aber durch geschickte Anordnung der Integrationstests können einzelne Dienste dennoch validiert werden.

Integration-Testing: 3 Punkte - Integrationstests werden bei FDD nach der Fertigstellung mehrerer Features durchgeführt.

System-Testing & Abnahme-Tests: 2 Punkte - Systemtests werden bei der Feature-getriebenen Entwicklung vollständig spezifiziert, die Definition von Abnahmetests ist ungenügend.

Sicherheitstests & Security Penetration Testing: 0 Punkte - Die Überprüfung der Sicherheit ist kaum realisierbar, weil jede Iteration genau ein Merkmal testet, Services werden hierbei nicht überwacht.

SOA Governance Testing: 0 Punkte - Auch die Einhaltung der unternehmensinternen Richtlinien können nicht validiert werden, weil die Tests auf einzelne Features ausgelegt sind.

Regression-Testing: 1 Punkt - Werden Fehler in der Implementierung festgestellt, wird die fehlerhafte Komponente erneut getestet, eine Überprüfung des Gesamtsystems fehlt.

Testgetriebene Entwicklungsmethode: 0 Punkte - Ein Feature wird geplant, implementiert und anschließend getestet. Die testgetriebene Entwicklungsmethode wird somit nicht verwendet.

Häufigkeit der Test-Iterationen: 1 Punkt - Tests werden nur wiederholt, wenn Fehler im jeweiligen Feature gefunden werden.

Monitoring während der Testausführung: 0 Punkte - Ein Konzept zum Monitoring einzelner Features oder Services wird nicht vorgestellt.

Bewertung: Test Driven Development

Die testgetriebene Entwicklung kann die Qualität und das Verständnis einer Implementierung erheblich steigern, jedoch können häufige Änderungen des Programmcodes bei geringer Dokumentation die Komplexität der Service-orientierten Architektur anheben.

Unit-Testing: 4 Punkte - Die Komponenten der SOA werden durch die testgetriebene Entwicklungsmethode optimal getestet.

Service-Testing: 3 Punkte - Eine Überprüfung eines Dienstes kann ebenfalls testgetrieben erfolgen, jedoch gestaltet sich die Testvorbereitung ohne ausreichender Dokumentation schwierig.

Integration-Testing: 1 Punkt - Ein Test des Zusammenspiels mehrerer Komponenten ist ohne großer Vorstellungskraft nicht realisierbar.

System-Testing & Abnahme-Tests: 0 Punkte - Die Größe einer Service-orientierten Architektur zeigt die Grenzen der testgetriebenen Entwicklung, weil niemals alle Tests im Vorfeld spezifiziert werden können.

Sicherheitstests & Security Penetration Testing: 1 Punkt - Ein Teil aller Sicherheitstests kann vor der Implementierung erstellt werden, eine konkrete Spezifikation ist durch die Methode nicht gegeben.

SOA Governance Testing: 0 Punkte - Die Überprüfung der Einhaltung unternehmensinterner Richtlinien kann nicht testgetrieben erfolgen.

Regression-Testing: 1 Punkt - Die Komponente und Services werden nach Ausbesserungen von Fehlern erneut getestet, das gesamte System jedoch nicht.

Testgetriebene Entwicklungsmethode: 4 Punkte - Der Name impliziert das testgetriebene Vorgehen.

Häufigkeit der Test-Iterationen: 3 Punkte - Tests werden iteriert, bis die Implementierung fehlerfrei ist. Integrations- und Systemtests können, wie bereits oberhalb erwähnt, nicht mit der testgetriebenen Entwicklungsmethode überprüft werden.

Monitoring während der Testausführung: 0 Punkte - Ein Konzept zur Überwachung der Testausführung für eine Service-orientierte Architektur wird in der testgetriebenen Entwicklungsmethode nicht spezifiziert.

5.3 Vergleich der Bewertungen

Dieser Abschnitt vergleicht die durchschnittlichen Ergebnisse aller Testprozesse, welche im letzten Kapitel ermittelt wurden, direkt miteinander. Ein Ranking der evaluierten Prozesse soll klären, welche Modelle für das Testen von Service-orientierten Architekturen am Besten geeignet sind.
Die Tabellen im Apendix A.1 werden zur Vollständigkeit in diese Arbeit aufgenommen, sie geben eine Übersicht aller im letzten Teil bereits präsentierten Ergebnisse. Zusätzlich wurden aller Werte für jedes Modell summiert und der jeweilige Durchschnittswert mit der Formel Summe aller Bewertungen/Anzahl aller Bewertungen berechnet.
Die Abbildung 5.1 zeigt die ermittelten durchschnittlichen Werte für jedes Modell im direkten Vergleich. Die Prozesse wurden in folgende Kategorien eingeordnet:

- Top-Down/Bottom-Up Vorgehensweise: Hierzu zählen das Wasserfallmodell (rein nach dem Top-Down Prinzip), das V- und das W-Modell (Top-Down/Bottom-Up).

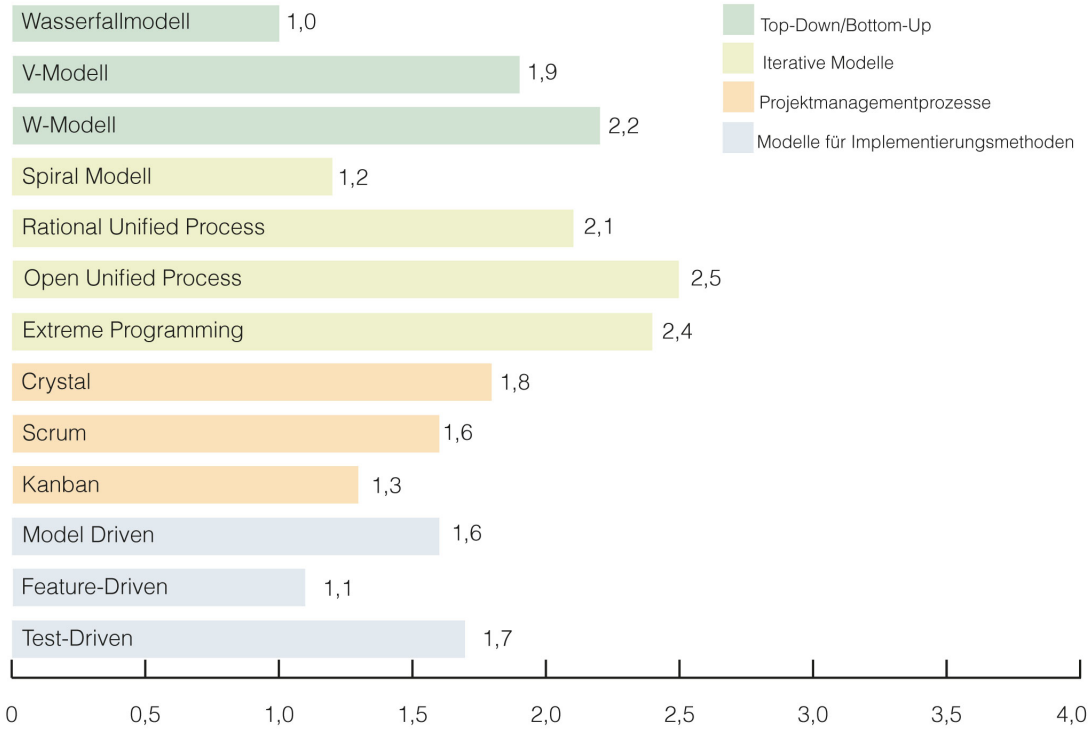

Abbildung 5.1: Vergleich der Ergebnisse

- Iterative Modelle: In der Menge der iterativen Modelle sind das Spiralmodell, der IBM Rational Unified Process, der Open Unified Process und die Extreme Programming Methode zu finden.
- Projektmanagement-Prozesse: Die Crystal Family, der Scrum-Prozess und die Kanban Vorgehensweise gehören zu der Gruppe der Modelle für effizientes Projektmanagement.
- Modelle für Implementierungsmethoden: Diese Modelle umfassen rein technische Ansätze für die Implementierung und das Testvorgehen von Software. Hierzu zählen die modellgetriebene Entwicklungsmethode, das "Feature-Driven Development" und die testgetriebene Entwicklung.

Deutlich ersichtlich ist hierbei, dass reine Projektmanagement-Prozesse wie Scrum oder Kanban sich im Schnitt am Wenigsten für das Testen von Service-orientierten Architekturen eignen. Die Beschreibung der Vorgehensweise ist bei diesen Modellen minimal gehalten, dies ist für Tests von SOA-Umgebungen nicht optimal.
Modelle mit Top-Down/Bottom-Up Vorgehen mit möglichen Rückschritten zu vorherigen Phasen sowie iterative Prozessmodelle schneiden im Vergleich sehr gut ab, eine rein agile Vorgangsweise ist aufgrund der Größe und Komplexität einer Service-orientierten Architektur nicht zu empfehlen.
Das WASI-Modell versucht die Vorteile aller analysierten Entwicklungsmethoden zu kombinieren:

Kleine Schwächen sind nur bei der Verwendung der testgetriebenen Entwicklungsmethode und bei den Iterationen zu erkennen. Die "Test-Driven-Development" Methode wird im WASI-Modell aus Qualitätsgründen nur bis zur Service-Ebene benutzt und die Anzahl der Iterationen ist geringer als bei rein iterativen Prozessmodellen. Abgesehen von diesen Kompromissen eignet sich das WASI-Modell hervorragend zur Spezifikation, Implementierung und Überprüfung einer Service-orientierten Architektur und deren Schnittstellen, eine Lösung für effizientes Monitoring einer SOA-Umgebung wurde zusätzlich geschaffen.

KAPITEL 6

Zusammenfassung

Die Ergebnisse, welche in den letzten zwei Kapiteln präsentiert wurden, werden hier übersichtlich und prägnant zusammengefasst. Diese Diplomarbeit soll abschließend mit einem Ausblick auf eine mögliche Zukunft der Qualitätssicherung heterogener Systemlandschaften enden.

6.1 Ergebnis & Bewertung

Das "Well-Assisted SOA Implementation" (WASI) Modell definiert einen Entwicklungsprozess, der in allen Projektphasen, von der Ermittlung der Business Requirements, der Dokumentation, der eigentlichen Implementierung bis hin zur Überprüfung des gesamten Systems, auf die Anforderungen einer Service-orientierten Architektur zugeschnitten ist und den speziellen Aufbau der Software-Umgebung berücksichtigt. Durch eine ausführliche Recherche wurden folgende qualitätssichernden Maßnahmen und Konzepte gefunden und implementiert:

- Regressionstests: Diese Tests überprüfen das komplette System nach der Ausbesserung gefundener Fehler erneut, die Fehleranzahl wird somit minimiert.

- Sensoren: Das Konzept des Sensors erlaubt effizientes Monitoring der Testfälle und der Services bereits während der Laufzeit. Sensoren werden über standardisierte Schnittstellen mit den einzelnen Komponenten verbunden.

- Risikomanagement: Methoden für die Identifikation, die Beurteilung, die Steuerung und die Überwachung von Risiken werden im Prozessmodell integriert um die Möglichkeit des Scheiterns des Projektes zu reduzieren.

- SOA Governance Testing: Die unternehmensinternen Richtlinien und Standards werden genau spezifiziert und während der Ausführung des WASI-Prozesses auf der jeweiligen Ebene häufig überprüft.

- Testgetriebene Entwicklung für Komponenten, Services und Sensoren: Der Einsatz der testgetriebenen Entwicklungsmethode soll die Qualität der Service-orientierten Architektur steigern.
- Spezifikation der Services und Komponenten: In allen bisherigen Prozessmodellen sind die Begriffe "Service" und "Komponenten" für SOA nicht spezifiziert. Das WASI-Modell beseitigt diesen Missstand.

Das W-Modell wurde wegen der Spezifikation des Systems nach dem Top-Down Prinzip und der Überprüfung nach der Bottom-Up Methode als Grundlage für das WASI-Modell ausgewählt und entsprechend an die zuvor ermittelten Anforderungen an Service-orientierte Architekturen angepasst.
Das Ergebnis ist ein detailliertes Prozessmodell, welches, wenn richtig eingesetzt, die Qualität einer SOA durch effiziente Überprüfungs- und Monitoringmethoden, erheblich steigern kann.

Ein anschließender Vergleich, welcher die Frage klärt, welches Prozessmodell sich am Besten zur Überprüfung einer Service-orientierten Architektur eignet, bringt das gewünschte Ergebnis: Das WASI-Modell erzielt die Höchstpunktezahl.
Kleine Schwächen sind bei der moderaten Anzahl an Testiteration und beim eingeschränkten Einsatz der testgetriebenen Entwicklungsmethode zu erkennen. Des Weiteren werden die Sensoren bei Einsatz des WASI-Modells in die Komponenten des Systems integriert. "Blackbox Services" hingegen werden nur über die bereitgestellten Schnittstellen getestet, dieses Verfahren wird bereits von zahlreichen Programmierern verwendet. Ein laufendes Monitoring dieser speziellen Service-Kategorie ist somit auch mit Hilfe des WASI-Modells nicht möglich.
Das gesetzte Ziel wurde jedoch erreicht: Das WASI-Modell ermöglicht effizientes Testen und Monitoring einer Service-orientierten Architektur. Aus diesen Daten lassen sich schnell Berichte generieren, die dem IT-Management bei wichtigen Entscheidungen helfen. Das WASI-Modell sichert nicht nur die Qualität der Schnittstellen einer heterogenen Systemlandschaft, sondern der gesamten Software-Umgebung.

6.2 Ausblick

Service-orientierte Architekturen gewinnen in der Softwareentwicklung stetig an Bedeutung, die jetzige Situation ist allerdings überraschend: Enorme Defizite sind bei der Überprüfung dieser speziellen Ausprägung eines verteilten Systems erkennbar, Konzepte für Integrationstests von unabhängigen Services und deren Schnittstellen sind Mangelware.
Diese Diplomarbeit ist ein erster Schritt in die Richtung zur Beseitigung dieses Missstandes, jedoch kann man in den nächsten Jahren mit der Publikation einer Vielzahl weiterer fortgeschrittener Testkonzepte für Service-orientierte Architekturen rechnen, da die Komplexität der Fehlersuche mit der Größe eines verteilten Systems signifikant steigt und effiziente Methoden zur Qualitätssicherung und damit verbundener Kostenreduktion immer wichtiger werden.

APPENDIX A

Appendix

A.1 Ergebnisse des Prozessvergleichs

Verwendete Abkürzungen in den nachfolgenden Tabellen:

WF: Wasserfallmodell

V: V-Modell

W: W-Modell

Spiral: Spiralmodell

RUP: IBM Rational Unified Process

OUP: Open Unified Process

XP: Extreme Programming

Crystal: Crystal Family

MDD: Model Driven Development

FDD: Feature Driven Development

TDD: Test Driven Development

-	WF	V	W	Spiral	RUP	OUP	XP
Unit	3	4	4	2	4	4	4
Service	1	3	3	1	2	2	2
Integration	2	3	3	2	4	4	4
System	2	4	4	2	4	4	3
Security	0	1	1	1	1	2	1
Governance	0	1	1	1	1	2	1
Regression	1	2	3	0	2	2	1
TDD	0	0	1	0	0	1	4
Iterationen	1	1	2	3	3	3	4
Monitoring	0	0	0	0	0	1	0
Summe	10	19	22	12	21	25	24
Durchschnitt	**1.0**	**1.9**	**2.2**	**1.2**	**2.1**	**2.5**	**2.4**

Tabelle A.1: Ergebnisse des Vergleichs aus Kapitel 5 (Tabelle 1/2)

-	Crystal	Scrum	Kanban	MDD	FDD	TDD
Unit	4	2	2	3	3	4
Service	1	1	1	3	1	3
Integration	2	1	1	3	3	1
System	3	1	1	2	2	0
Security	1	0	0	0	0	1
Governance	1	1	0	0	0	0
Regression	1	2	2	4	1	1
TDD	1	4	4	0	0	4
Iterationen	4	4	2	1	1	3
Monitoring	0	0	0	0	0	0
Summe	18	16	13	16	11	17
Durchschnitt	**1.8**	**1.6**	**1.3**	**1.6**	**1.1**	**1.7**

Tabelle A.2: Ergebnisse des Vergleichs aus Kapitel 5 (Tabelle 2/2)

Literaturverzeichnis

[Aal98] AALST, W.M.P. van d.: *The Application of Petri Nets to Workflow Management*. Eindhoven University of Technology, 1998

[Ama] AMAZON (Hrsg.): *Amazon Elastic Compute Cloud*. `http://aws.amazon.com/ec2/`, Abruf: 2011-07-28

[Amb02] AMBLER, S.W.: *Agile Modeling: Effective Practices for XP and the UP*. John Wiley and Sons, 2002

[Bar05] BARTELT, Christian: *Modellbasierte Entwicklung mit dem V-Modell XT*. OBJEKTspektrum, 2005

[BD04] BRÜGGE, Bernd ; DUTOIT, Allen: *Objekt-orientierte Softwaretechnik mit UML, Entwurfsmustern und Java*. Pearson Studium, 2004

[BDS10] BRATANIS, Konstantinos ; DRANIDIS, Dimitris ; SIMONS, Anthony J. H.: *An extensible architecture for run-time monitoring of conversational web services*. ACM, 2010

[Bec99] BECK, Kent: *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999

[Bec02] BECK, Kent: *Test Driven Development by Example*. Addison Wesley, 2002

[BGS01] BEYDEDA, S. ; GRUH, V. ; STACHORSKI, M.: *A graphical class representation for integrated black- and white-box testing*. Management of Innovation and Technology, 2006 IEEE International Conference, 2001

[BKM07] BIANCO, Phil ; KOTERMANSKI, Rick ; MERSON, Paulo: *Evaluating a Service-Oriented Architecture*. Software Engineering Institute, 2007

[BL06] BRYNE, Jan C. ; LANZEN, Anders: *Web Services Tutorial*. Center for Scientific Computing, Espoo Finland, 2006

[Boe79] BOEHM, B.W.: *Guidelines for verifying and validating software requirements and design specifications, Pages 711-719*. TRW, 1979

[Boe88] BOEHM, B.W.: A spiral model of software development and enhancement. In: *Computer*, IEEE Computer Society, 1988. – ISBN 0018–9162, S. 61–72

[Bor] BORLAND (Hrsg.): *Borland SilkPerformaer*. http://www.borland.com/resources/en/pdf/products/silk/Borland-SilkPerformer.pdf, Abruf: 2011-06-15

[BP09] BERTOLINO, A. ; POLINI, A.: *SOA Test Governance Enabling Service Integration Testing across Organization and Technology Borders*. Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference, 2009

[BSGS04] BORN, Marc ; SCHIEFERDECKER, Ina ; GROSS, Hans-Gerhard ; SANTOS, Pedro: *Model Driven Development and Testing A Case Study*. Enschede Netherlands, 2004

[BSS11] BLANK, Christoph ; STEINMETZ, Nathalie ; SCICLUNA, James: *seekda! Enriched Services - A Tool for Semantic Service Annotation*. seekda GmbH, Grabenweg 68, A-6020 Innsbruck, Austria, 2011

[BW08] BRUNNER, F. J. ; WAGNER, K. W.: *Qualitätsmanagement Leitfaden für Ingenieure und Techniker*. Carl Hanser Verlag München Wien, 2008

[CLL99] COAD, Peter ; LEFEBVRE, E. ; LUCA, Jeff de: *Java Modeling in Color with UML*. Prentice Hall, 1999

[Coc03] COCKBURN, Alistair: *Agile Softwareentwicklung*. mitp, 2003

[Coc05] COCKBURN, Alistair: *Crystal Clear*. Addison Wesley, 2005

[dev] DEVX.COM (Hrsg.): *A Practical Guide to Seven Agile Methodologies Part 2*. http://www.devx.com/architect/Article/32836/0/page/2, Abruf: 2011-06-11

[DGH03] DUSTDAR, Schahram ; GALL, Harald ; HAUSWIRTH, Manfred: *Softwarearchitekturen für verteilte Systeme*. Springer Verlag, 2003

[Ecl] ECLIPSE.ORG (Hrsg.): *eclipse.org - Introduction to OpenUP*. http://www.eclipse.org/epf/general/OpenUP.pdf, Abruf: 2011-01-31

[EN10] ETZION, Opher ; NIBLETT, Peter: *Event Processing in Action*. Manning Publications, 2010

[Eri] ERIK CHRISTENSEN, FRANCISCO CURBERA, GREG MEREDITH, SANJIVA WEERAWARANA (Hrsg.): *W3C Web Services Description Language (WSDL) 1.1*. http://www.w3.org/TR/wsdl, Abruf: 2011-08-04

[Evi] EVIWARE (Hrsg.): *Eviware soapUI*. http://www.eviware.com/soapUI/features.html, Abruf: 2011-06-15

[Fie00] FIELDING, Roy: *Architetectural Styles and the Design of Network-based Software Architectures*. University of California Irvine, 2000

[FMG02] FIEGE, Ludger ; MÜHL, Gero ; GÄRTNER, Felix C.: *Modular Event-Based Systems*. Cambridge University Press, 2002

[Gar96] GARTNER: *Service Oriented' Architectures, Part 1*. SSA Research Note SPA-401-068, 1996

[Ger07] GERHARDT, Chris: *Enterprise Mashups*. Universität Karlsruhe, 2007

[GKZ04] GRECHENIG, Thomas ; KÖHLE, Monika ; ZUSER, Wolfgang: *Software Engineering mit UML und dem Unified Process 2. Auflage*. Pearson Studium, 2004

[Gre] GREENHAT (Hrsg.): *Greenhat GH Tester*. `http://www.greenhat.com/products/ghtester.html`, Abruf: 2011-06-15

[GW03] GEORGE, Boby ; WILLIAMS, Laurie: *A structured experiment of test-driven development*. Elsevier B.V., 2003

[Ham04] HAMILL, Paul: *Unit Test Frameworks*. O'Reilly Media, 2004

[Har07] HARRIS, Torry: *SOA Test Methodology*. Torry Harris Business Solutions, 2007

[HB04] HAAS, Hugo ; BROWN, Allen: *W3C Web Services Glossary*. W3C, 2004

[IEE91] IEEE: *IEEE Standard Glossary of Software Engineering Terminology*. IEEE, 1991

[IEE08] IEEE: *IEEE Standard for Software and System Test Documentation*. IEEE, 2008

[ISO08] ISO: *ISO 9001:2008 Quality Management Systems Requirements*. International Standards for Business, Government and Society, 2008

[JBR99] JACOBSON, Ivar ; BOOCH, Grady ; RUMBAUGH, James: *The Unified Software Development Process*. Addison-Wesley, 1999

[JfSJb+06] JUN-FENG, Yao ; SHI, Ying ; JU-BO, Luo ; DAN, Xie ; XIANG-YANG, Jia: *Reflective Architecture Based Software Testing Management Model*. Management of Innovation and Technology, 2006 IEEE International Conference, 2006

[Kec11] KECHER, Christoph: *UML 2 Das umfassende Handbuch*. Galileo Computing, 2011

[Kru04] KRUCHTEN, Philippe: *The Rational Unified Process - An Introduction (Third Edition)*. Addison-Wesley, 2004

[Kur00] KURSCHL, Werner: *Forschungsbericht Monitoring von verteilten System*. Johannes Kepler Universität, 2000

[Lad08] LADAS, Corey: *Scrumban - Essays on Kanban Systems for Lean Software Development*. Modus Cooperandi Press, 2008

[Lee09] LEE, Youngkon: *SOA Test Framework Based on BPA -Simulated Event Proxy*. IEEE Computer Society, 2009

[LLZ+09] LIU, Hehui ; LI, Zhongjie ; ZHU, Jun ; TAN, Huafang ; HUANG, Heyuan: *A Unified Test Framework for Continuous Integration Testing of SOA Solutions*. IEEE, 2009

[Luc02] LUCKHAM, David: *The Power of Events An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002

[MS95] MANSOURI-SAMANI, Masoud: *Dissertation Monitoring of Distributed Systems*. University of London, 1995

[MWW05] MICHAEL WICK, Daniel S. ; WAGNER, Paul: *Using Testing and jUnit Across The Curriculum*. University of Wisconsin-Eau Claire, 2005

[NCL10] NING, JingFeng ; CHEN, Zhiyu ; LIU, Gang: PDCA process application in the continuous improvement of software quality. In: *Computer, Mechatronics, Control and Electronic Engineering (CMCE), 2010 International Conference on* Bd. 1, 2010, S. 61 –65

[Nie05] NIEBUHR, Dirk: *Erfolgreiche IT-Projekte mit dem V-Modell XT*. OBJEKTspektrum, 2005

[OAS] OASIS CONSORTIUM (Hrsg.): *Reference Model for Service Oriented Architecture 1.0,Committee Specification 1, 2 August 2006*. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm`, Abruf: 2011-04-16

[Par] PARASOFT (Hrsg.): *Parasoft SoaTest*. `http://www.parasoft.com/jsp/products/soatest.jsp`, Abruf: 2011-06-15

[Par97] PARHAMI, Behrooz: *Defect, Fault, Error, ..., or Failure*. University of California Santa Barabara, 1997

[Pfe01] PFEIFER, Tilo: *Qualitätsmanagement. Strategien, Methoden, Techniken*. Hanser Verlag München, 2001

[PFP02] PALMER, Stephen R. ; FELSING, Mac ; PALMER, Steve: *A Practical Guide to the Feature-Driven Development*. Prentice Hall International, 2002

[PTN+07] PIETREK, Georg ; TROMPETER, Jens ; NIEHUES, Benedikt ; KARMANN, Thorsten ; HOLZER, Boris ; KLOSS, Michael ; THOMS, Karsten ; BELTRAN, Juan Carlos F. ; MORK, Steffen: *Modellgetriebene Softwareentwicklung. MDA und MDSD in der Praxis. 1. Auflage*. Entwickler-Press, 2007

[Pus] PUSHTOTEST (Hrsg.): *PushToTest TestMaker*. `http://www.pushtotest.com/products.html`, Abruf: 2011-06-15

[Roy70] ROYCE, Winston: *Managing The Development of Large Software Systems, Pages 328-338*. TRW, 1970

[Sch95] SCHROEDER, B.A.: *Online monitoring: a tutorial Page 72 to Page 78*. IEEE Computer Society, 1995

[Sch07] SCHWABER, Ken: *Agiles Projektmanagement mit Scrum*. Microsoft Press Deutschland, 2007

[SRWL06] SPILLNER, Andreas ; ROSSNER, Thomas ; WINTER, Mario ; LINZ, Tilo: *Praxiswissen Softwaretest - Testmanagement, Kapitel 3.4 (W-Modell)*. dpunkt Verlag, 2006

[SS07] SCHMELZER, Hermann ; SESSELMANN, Wolfgang: *Geschäftsprozessmanagement in der Praxis*. Carl Hanser Verlag, 2007

[SUN] SUN (Hrsg.): *SOA Benefits Overview*. `http://www.sun.com/products/soa/benefits.jsp`, Abruf: 2011-02-07

[SVEH07] STAHL, Thomas ; VÖLTER, Markus ; EFFTINGE, Sven ; HAASE, Arno: *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management. 2. Auflage*. dpunkt, 2007

[Tha02] THALLER, G.E.: *Software-Test*. dPunkt, 2002

[TS07] TANENBAUM, Andrew S. ; STEEN, Maarten van: *Verteilte Systeme 2. Auflage*. Pearson Studium, 2007

[VKZ05] VÖLTER, Markus ; KIRCHER, Michael ; ZDUN, Uwe: *Remoting Patterns*. WILEY, 2005

[W3C] W3C CONSORTIUM (Hrsg.): *W3C Input to the United Nations Working Group on Internet Governance*. `http://www.w3.org/2005/04/dd-wsig.html`, Abruf: 2010-11-30

[Wel99] WELLS, Don: *ExtremeProgramming.org*. Internet: zuletzt gesichtet 04.November 2010, 1999

[Wes98] WESKE, Mathias: *Deadlocks in Computersystemen*. Redline, 1998

[WK03] WILLIAMS, Laurie ; KESSLER, Robert: *Pair Programming Illumninated*. Addison-Wesley, 2003

[ZfXp09] ZHI-FANG, Liu ; XIAO-PENG, Gao: *SOA Based Mobile Device Test*. ICICTA, 2009

[ZQ06] ZHANG, Jia ; QIU, R.G.: *Fault Injection-based Test Case Generation for SOA-oriented Software*. IEEE, 2006