FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Planung und Entwicklung eines Tools für Reporting, Monitoring und Unterstützung der Durchführung von Tests von Schnittstellen in heterogenen Systemlandschaften

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Rene Schakmann
Matrikelnummer 0400767

an der
Fakultät für Informatik der Technischen Universität Wien

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

Wien, TT.MM.JJJJ

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung

Schakmann Rene
Kudlichstrasse 50
A-3100 St.Pölten

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.

Ort, Datum, Unterschrift

Abstract

Business processes using information technology are no longer bound to local or company boundaries. Distribution of company sites, stratetic partnerships or outsourcing of IT leads to a heterogeneous IT-landscape. These systems need to be reintegrated using various interfaces. Besides the development of distributed systems, testing is also a known issue in this area. Technical as well as organizational aspects need to be considered. In tests of interfaces, using various systems to transport data to a certain destination, errors are often difficult to find and locate. The knowledge of testers is often limited to business related topics. The system under test is seen as a blackbox and due to this fact finding errors often affords lot of time and personel costs. This work will show a framework concerning this problem of testing distributed systems. It will provide a concept of dealing with specific problems in this area and introduce a framework to support testers with their task.

Kurzfassung

Im Gegensatz zu früher sind Geschäftsprozesse, die durch IT unterstützt werden, nicht mehr an lokale oder unternehmerische Grenzen gebunden. Zahlreiche Partnerschaften, Auslagerung von Systemen oder die Verteilung von Standorten führen dazu, dass sich heterogene Systemlandschaften bilden, die durch unterschiedlichste Schnittstellen miteinander verbunden werden müssen. Nicht nur die Entwicklung solcher Systeme stellt Herausforderungen, auch der Test dieser verteilten Anwendungen beinhaltet sowohl organisatorische als auch technische Aspekte. Bei Tests, die Schnittstellen betreffen, welche Daten in mehreren Schritten über unterschiedliche Systeme transportieren, sind Fehler oft nur schwer nachzuvollziehen, da sich das System für den meist fachlichen Tester ohne technisches Hintergrundwissen als Blackbox darstellt. Das Auffinden des Fehlers ist oft nur mit hohem zeitlichem und organisatorischem Aufwand möglich. Unterschiedliche Systeme können unter unterschiedlichen organisatorischen Verantwortungen stehen oder unterschiedliche Technologien verwenden. Diese Arbeit zeigt einen Ansatz wie diese Integrationstests sowohl organisatorisch als auch technisch durch ein Framework unterstützt werden können um eine Beschleunigung und Vereinfachung des Testvorgangs zu erreichen und die Tests von verteilten Systemen transparenter zu gestalten.

Für meinen Vater

Inhaltsverzeichnis

Abbildungsverzeichnis

KAPITEL 1

Einführung

1.1 Motivation

Geschäftsprozesse in und zwischen Unternehmen haben durch die Unterstützung von IT-Systemen eine andere Charakteristik erhalten als dies früher der Fall war. Durch die zusätzliche Verbreitung von der leicht verfügbaren Vernetzung durch das Internet zwischen den Unternehmen kommt eine zusätzliche Dynamik in die Prozesse der Unternehmen. Abläufe die früher über Jahre hin fixiert waren können heutzutage innerhalb von wenigen Stunden einfach durch die Umstellung von Services in der IT geändert werden. Die Einbindung neuer Geschäftspartner, der Austausch bestehender oder die Erweiterung oder Auslagerung eines Geschäftsprozesses sind heutzutage in wesentlicher kürzer Zeit möglich als in vergangenen Jahrzenten. Kollaborationen zwischen Unternehmen stellen im modernen Geschäftsumfeld mittlerweile eine wichtige Funktion um Wettbewerbsfähig zu bleiben, neue Produkte anbieten zu können oder die Qualität von Dienstleistungen zu verbessern. IT Landschaften sind über Jahre hinweg gewachsen und diverse Insellösungen sind entstanden. Der Zusammenschluss diverser Softwaresysteme und deren Interaktion über Grenzen, seien es Standortgrenzen oder organisatorische Grenzen, ist zu einem essentiellen Bereich der Softwareentwicklung bzw. der Architektur von Systemen geworden.

Abbildung 1.1 illustriert, dass es in wesentliche Themengebieten rund um verteilte Systeme wie SOA(Service orientierte Architektur) oder Cloud Computing zur Zeit starke Weiterentwicklungen gibt, bzw. speziell im SOA Bereich ein Abschnitt erreicht wurde in dem es kurz davor steht eine breite Akzeptanz zu finden. Durch die darauf folgende breite Anwendung ist es wichtig, diese Gebiete im Bereich Qualitätssicherung, speziell im Fall von Softwaretests zu unterstützen.

Wie wichtig die Schnittstellen zwischen Unternehmen mittlerweile geworden sind, zeigt unter anderem, dass eigene Unternehmen wie APIGEE[Api] gegründet werden die sich nur diesem Verbindungsaspekt zwischen Geschäftspartnern widmen. Das Monitoring, Debugging und die Analyse dieser Schnittstellen hat zu einem neuen Geschäftszweig geführt der sich auf diese Thematik spezialisiert hat. Aufkommende Trends wie Cloud Computing und die damit noch stärkere Verteilung der einzelnen Systeme führen zu weiteren Schwierigkeiten beim Test und Betrieb von komplexen Softwaresystemen. Cloud Computing nimmt zwar dem Betreiber von Softwaresystemen die Verantwortung des Betriebs der Infrastruktur [WPG+10], nicht aber die Verantwortung des Betriebs der auf dem Knoten laufenden Software. Web 2.0 fällt ebenfalls in diesen Themenbereich. Viele neue Dienste werden durch das Zusammenfügen von anderen, bereits existierenden Diensten zur Verfügung gestellt.

Integrationstests stellen einen der wichtigsten Aspekte in verteilten Systemen dar. Viele kritische Fehler können erst durch einen ausführlichen Integrationstest aufgedeckt werden[LLZ+09]. Testen mittels BlackBox oder WhiteBox Tests wird nur auf Unit oder Komponentenebene angewandt. Insbesondere Blackboxtests werden häufig auf Komponentenebene verwendet[NRH10] um deren Funktionalität zu überprüfen. Für ganze Softwaresysteme sind diese Tests noch bedingt geeignet, da keinerlei Informationen über interne Zustände vorliegen und so die Fehlersuche erschwert. Bei großen komplexen Softwaresystemen kommt erschwerend hinzu, dass die zuständigen Personen für den Test, meist Fachtester sind. Daher ihr Wissen beschränkt sich auf ein Gebiet, das für die fachliche Richtigkeit der Software notwendig ist. Technische Hintergründe, die Verteilung der Software und andere Aspekte des Systems sind ihnen meist nur schemenhaft bekannt und es liegt auch nicht in ihrem Aufgabenbereich dies zu kennen. Dennoch sind diese

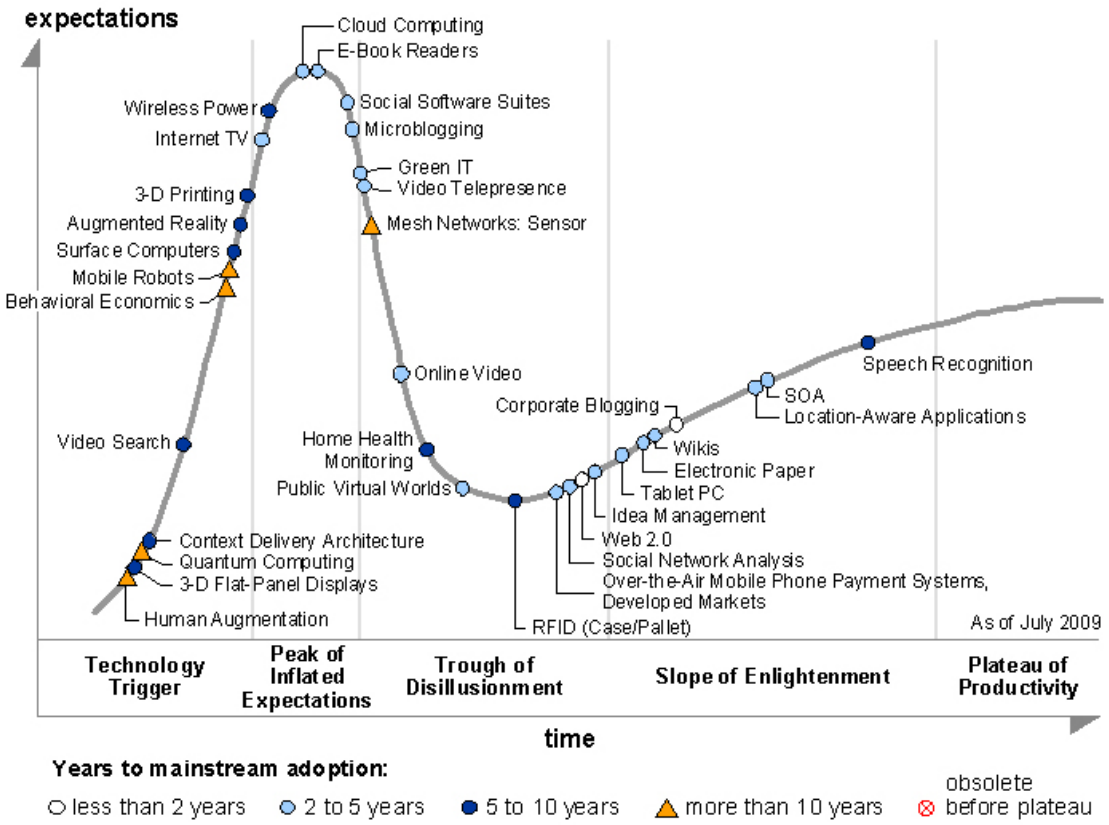

Abbildung 1.1: Hype Cycle Emerging Technologies 2009 [Gar]

Fachtester dafür verantwortlich die Korrektheit des Systems zu überprüfen. Softwaresysteme die sich über verschiedene organisatorische Grenzen bewegen erschweren die Suche nach Fehlern bzw. die Sicherstellung der korrekten Funktion des zu testenden Systems. Oft es ist es nur mit erheblichem organisatorischem Aufwand möglich Tests durchzuführen, da hierzu an verschiedenen Stellen im System Aktionen von Nöten sind, die nicht in der Herrschaft des Testers liegen. Dies gilt ebenfalls für die mögliche darauffolgende Fehlersuche. Der Tester kann nur lokal observierbare Zustände kontrollieren, und diese sind im Falle eines Fachtesters oft sehr beschränkt, da selbst wenn der lokale Zugriff auf eine Datenbank etc. möglich wäre, dieser nicht über die nötige Kompetenz verfügt, Überprüfungen dort durchzuführen. Diese Arbeit stellt ein Framework bzw. ein Tool vor das diese Problematik adressiert. Tester sollen sich auf ihre Kernkompetenz beschränken, und sich nicht über organisatorische Aspekte, Systemverteilung oder Problemen in der Beschaffung von Kontaktinformationen von zuständigen Personen beschäftigen müssen. HEKATE, das hier vorgestellte Framework, bietet den Testern die Möglichkeit ihre Testfälle zentral zu verwalten und über das ganze System hinweg zu verfolgen und diese zu validieren oder Fehler aufzudecken. Durch die gesammelten Informationen ist ein genaues Monitoring des Systems möglich und durch die Benutzung von historischen Daten die Erstellung von Reports über unterschiedliche Aspekte des Testvorganges.

Die Arbeit gliedert sich wie folgt:

Kapitel 2 bietet einen Überblick über Themengebiete wie verteilte Systeme, Testen bzw. Testmanagement. Geteilt in zwei Bereiche beschreibt es zuerst konzeptionelle Grundlagen. Der zweite Teil widtmet sich den technischen Grundlagen die in diesen Bereichen Anwendung finden bzw. die notwendig sind um die in dieser Arbeit erstellte Lösung zu beschreiben.

Kaptel 3 fasst Arbeiten zu diesen Themengebieten bzw. zu verwandten Bereichen zusammen und diskutiert Gemeinsamkeiten und Unterschiede zwischen den besprochen Ansätzen und dem hier erstellen Framework.

Kapitel 4 stellt eine genaue Spezifikation der Anforderungen an das erstelle Framework dar. Es beinhaltet die Beschreibung der Komponenten, die Architektur und Umsetzungsdetails.

Kapitel 5 zeigt durch die Anwendung des Frameworks in einem konkreten Anwednungsfall die Vor und Nachteile des Frameworks im Einsatz in einer praxisnahen Umgebung.

Kapitel 6 fasst die gewonnen Erkenntnisse zusammen, diskutiert diese und gibt einen Ausblick auf zukünftige Weiterentwicklungen.

1.2 Definitionen

Als Fachtester wird in dieser Arbeit eine Person definiert, die über ein erhebliches Wissen in einer bestimmten Domäne verfügt. Dieses Wissen wird für die fachliche Verifikation eines Softwaresystems eingesetzt. Ein Fachtester verfügt allerdings nicht über technisches Hintergrundwissen.

KAPITEL 2

Grundlagen

Dieses Kapitel gibt einen Überblick sowohl über konzeptionelle Grundlagen als auch über die technische Basis auf der das Framework aufsetzt. Es soll jeweils eine kurze Einführung erfolgen um ein grundsätzliches Verständnis der Rahmenbedingungen zu schaffen.

2.1 Konzeptionelle Grundlagen

Im folgenden Abschnitt soll eine kurze Einführung in die Charakteristika verteilter Systeme gegeben werden. Besondere Rücksicht wird dabei auf etablierte Bereiche wie Service orientierte Architekturen gelegt, aber auch aktuelle Trends wie Cloud Computing wird Rechnung getragen. Abschließend wird der Bereich Reporting und Monitoring in verteilten Systemen näher vorgestellt.

Verteilte Systeme

Als ein verteiltes System wird allgemein eine Ansammlung von autonomen Rechnern gesehen die mittels eines Netzwerk verbunden sind und entsprechende Software ausführen. Beinahe alle Systeme die heute in Verwendung sind, seien es moderne Handybetriebssysteme wie Android[Goob], Google Docs[Gooc] oder das wohl berühmteste Beispiel das Internet[W3Cf], sind Teil eines verteilten Systems. Neue Trends wie Cloud Computing[WPG+10] sind ohne Verteilung nicht realisierbar. Es gibt zahlreiche Gründe für den Einsatz von verteilten Systemen die sich in zwei Kategorien einteilen lassen, Problem-orientierte und Systemeigenschaften-orientierte. Problem-orientierte Gründe ergeben sich aus dem Problem selbst, wenn ein Rechner in Europa auf Dateien in den USA zugreifen möchte, muss zwangsläufig ein verteiltes System zum Einsatz kommen. Andere Gründe für den Einsatz können jedoch auch aus den nötigten Eigenschaften für ein gewünschtes System resultieren, diese werden im Folgenden aufgeschlüsselt[VKZ05]:

Performance & Skalierbarkeit Oft treten Systemlasten auf die von einer einzelnen Maschine nicht mehr kosteneffizient gehandhabt werden können. Die Aufteilung auf mehrere physische Rechner setzt voraus, dass die Zusammenarbeit dieser sichergestellt ist. Dieser Vorgang wird Load Balancing genannt und resultiert in einem verteilten System.

Fehlertoleranz Jede Hardware wird in absehbarer Zeit defekt, auch Softwarekomponenten können ausfallen und so zu einer Nichterreichbarkeit des Systems führen. Um diesem Szenario vorzubeugen verteilt man die Software auf unterschiedliche Systeme, dies resultiert wiederum in einem verteilten System.

Serivce und Client Ortsunabhängigkeit In vielen Systemen sind Clients und Services nicht im Vorhinein bekannt. Sie können sich jederzeit hinzuschalten und das System wieder verlassen.

Wartbarkeit und Deployment Wartung und Deployment bei einer großen Anzahl an Clients erhöht die Kosten einer Software(TCO - Total Cost of Ownership) enorm. Thin Clients stellen hier einen besseren Ansatz dar. Hierbei wird die Business Logic remote auf einem Server ausgeführt und nur die Ergebnisse beim Client angezeigt.

Sicherheit In den meisten Systemen ist es sinnvoll sicherheitsrelevenate Informationen wie Zugangsdaten, Zugriffsbeschränkungen etc. an einem zentralen Ort zu speichern und zu verwalten. Auf der anderen Seite ist es ebenfalls sinnvoll wichtige Daten an geografisch unterschiedlichen Orten zu verteilen um bei einem Ausfall eines Standorts die Funktionsfähigkeit des ganzen Systems nicht zu gefährden.

Business Inegration Das Unternehmen ihre Systeme verbinden ist längst Standard. Enterprise Applicaion Integration(EAI) lässt sich nicht mehr aus heutigen Businessumfeldern wegdenken ohne massive Einschnitte in der Produktivität hinzunehmen.

Verteilte Systeme sind in den unterschiedlichsten Ausprägungen vorhanden. Je nach den speziellen Anforderungen kommen unterschiedliche Architekturen zum Einsatz. Um diese besser einordnen zu können werden im Folgenden Software-Architekturstile kurz beschrieben[DGH03]:

Daten zentrierte Software-Architekturen(data centered) sind auf den Zugriff und die Aktualisierung von Daten ausgelegt(Repository). Das zentrale Element hierbei ist der Datenbehälter der mit den Clients kommuniziert. Dieser kann rein passiv sein, daher nur der Datenhaltung dienen, oder als aktiver Datenbehälter (blackboard) implementiert sein. Handelt es sich um einen aktiven so sendet dieser, sollten sich Daten ändern, von sich aus Nachrichten an alle Clients(Subscriber) die sich vorher für diese Änderungen angemeldet haben. Sowohl die Clients untereinander als auch Client und Server sind voneinander nicht abhängig, daher können neue Clients hinzugefügt werden ohne andere bereits vorhandene Clients zu beeinflussen.

Datenfluss-Architekturen (dataflow architectures) fokussieren sich auf die Wiederverwendung und Modifizierbarkeit. Das System wird als eine bestimmte Reihenfolge von unterschiedlichen Transformationen gesehen. Werden nun Daten in diesem System verarbeitet, wandern sie Schritt für Schritt durch die unterschiedlichen Komponenten die die Daten manipulieren. Am Ende werden diese Daten als Ausgabestrom wieder bereitgestellt und können von anderen Systemen weiter verarbeitet werden. Hier gibt es zwei verschiedene Ausprägungen. Bei Batch-Sequential sind die Komponenten unabhängig und jede fängt erst an die Daten zu verarbeiten nachdem der gesamte Inputstrom gelesen wurde, die Komponenten arbeiten also eine nach der anderen die Daten ab. Im Gegensatz dazu wird beim Pipe-and-Filter Stil nicht gewartet bis die gesamte Eingabe gelesen wurde bevor Daten auf den Ausgabestrom geschrieben und weiter verarbeitet werden. Der große Vorteil dieser Architekturen ist ihre Einfachheit. Es müssen keine komplexen Interaktionen verwaltet werden. Jeder Filter kann als Black-Box gesehen werden und ist dadurch leicht ersetzbar. Es ergeben sich dadurch allerdings auch einige Nachteile, zum Beispiel verlangen die Filter natürlich nach dem kleinsten gemeinsamen Nenner der Datenrepräsentation. Auch kann es zu Overheads kommen wenn z.B.: jeder Filter in einem eigenen Prozess läuft.

Call-and-Return Architekturen besitzen in der Softwareentwicklung schon eine lange Tradition, dadurch haben sich bereits diverse Substile ausgeprägt. Die klassische Hauptprogramm-Unterprogramm-Architektur beschreibt das typisch Programmierparadigma. Es gibt einen singulären Kontrollfluss der nach und nach jede Komponente, die weiter unten in der Hierarchie angesiedelt ist, aufruft. Remote Procedure-Call(RPC) Architekturen verteilen unterschiedliche Komponenten auf unterschiedlichen Rechnern, wodurch eine parallele Abarbeitung ermöglicht wird. Objektorientierte oder abstrakte Datentyp-Architekturen sind hauptsächlich auf die Kapselung von Daten und das Prinzip von information hiding fokussiert. Bei Schichten-System-

Architekturen(layered architectures) werden die Komponenten unterschiedlichen Schichten zugeordnet. Diese sind wiederum hierarchisch organisiert. Eine Schicht verwendet dabei immer die darunter liegende Schicht und gibt Daten nur an die direkt darüber liegende Schicht weiter. Dieser Aufbau ermöglicht eine zunehmende Abstraktion im Design, doch nicht jedes System lässt sich sinnvoll in Schichten zerlegen.

Unabhängige Komponenten-Architekturen bestehen aus diversen Prozessen oder Objekten die über den Austausch von Nachrichten untereinander kommunizieren. Das primäre Ziel dieses Stils ist es, eine möglichst leichte Modifizierbarkeit durch die Entkopplung zu erreichen. Unter diesen Typ fallen auch ereignisgesteuerte(Event-basierte) Systeme. Hierbei geben einzelne Komponenten Informationen bekannt(publish). Andere Komponenten die diese Informationen erhalten wollen, müssen sich zuerst bei dieser registrieren(subscribe). Der Publisher und der Subscriber kennen sich in diesen Systemen nicht direkt, sondern nur über eine dritte Komponente die diese Kommunikation verwaltet. Der Publisher gibt seine Informationen an dieses Eventsystem weiter und dieses wiederrum leitet die Nachrichten zu allen Komponenten die sich hierfür angemeldet haben. Diese asynchrone Kommunikation erlaubt eine weitgehende Entkopplung der Systeme. Eine weitere Ausprägung der Unabhängige Komponenten-Architektur findet sich im Peer-to-Peer-Stil wieder. Die Interaktion zwischen den Komponenten erfolgt hierbei im Request/Reply Stil, allerdings ohne die Symmetrie wie in Client-Server Systemen.

In der Praxis können Softwaresysteme jedoch nie mit nur einem einzigen architekturellen Stil aufgebaut werden. Unterschiedliche Stile übernehmen unterschiedliche Aufgaben. Diese Heterogenität lässt sich in drei Arten aufteilen[VKZ05]:

- Bereichsabhängige Heterogenität: Ein Subsystem folgt nicht dem generellen Stil des Hauptsystems
- Hierarchieabhängige Heterogenität: In unterschiedliche Hierarchieebenen geteilte Systeme verwenden unterschiedliche Ansätze auf den verschiedenen Ebenen
- Simultane Heterogenität: Ein einziges System kann durch mehrere Stile gleichzeitig beschrieben werden

Daraus resultierend ergeben sich unterschiedlichste Herausforderungen speziell für verteilte Systeme[VKZ05]:

Netzwerk Latenz Ein Aufruf einer Komponente die auf einem entfernten System liegt, benötigt signifikant mehr Zeit als ein Aufruf auf dem lokalen System. Für zeitkritische Systeme muss diese Verzögerung genau beachtet werden.

Vorhersagbarheit Die Zeit die ein Aufruf benötigt variiert bei verteilten Systemen auf Grund des zwischen geschalteten Netzwerkes stärker als bei lokalen Systemen. Des Weiteren müssen Faktoren wie die Bandbreite, Netzwerklast, Leistung der Maschine auf der das Remote Service liegt und etliche andere Parameter beachtet werden. Außerdem können während der Kommunikation jederzeit Fehler auftreten.

Nebenläufigkeit Wann welche Komponenten welche Funktion ausführen ist in verteilten Systemen oft nur schwer vor herzusagen. Nebenläufigkeit führt zu einer Reihe von Problemen wie Nicht-Determinismus, Deadlocks und Race Conditions.

Skalierbarkeit Auf Grund der Eigenheiten von verteilten Systemen kann nicht immer vorhergesagt werden welche Systeme zu welcher Zeit verfügbar sein werden oder welche Anzahl an Clients eine gewisse Ressource anfordern (z.B.: wie viele Besucher gerade eine gewisse Website betrachten wollen). Deswegen muss die Hardware als auch die Software in beide Richtungen skalierbar sein.

Fehler in Teilsystemen Systeme die auf einer einzigen Hardware laufen, wohlmöglich noch in einem Prozess werden einfach gestoppt, sollte es zum Beispiel zu einem Hardwarefehler kommen. Verteilte Systeme erfordern ein ausgereifteres Fehlerhandling, da jederzeit Teilsysteme nicht mehr ihre Funktion erfüllen könnten.

SOA

Eine Ausprägung der Architektur eines verteilten Systems ist die Service orientierte Architektur(SOA). SOA wurde 1996 in der Research Note von Gartner[gar96] das erste Mal erwähnt. SOA ist im Vergleich mit anderen Ansätzen wie zum Beispiel JMI[Orab] Technologie unabhängig. In den früheren Jahren war das Stichwort SOA oft nur ein Verkaufsargument für jedes System das Services anbot. Doch in den letzten Jahren hat sich SOA zu einem fest etablierten Architekturparadigma für verteilte Systeme entwickelt.

SOA ist ein Ansatz um ältere monolithische Softwaresysteme abzulösen. Der Fokus hierbei liegt auf der Gestaltung der Systemlandschaft mit kleineren und wiederverwendbaren Services. Hierdurch erreicht man eine deutliche Reduktion der Kosten der Wartbarkeit und des Betriebs. Durch die Wiederverwendbarkeit der Services, kann Funktionalität die früher oft doppelt implementiert wurde, an einer zentralen Stelle verwaltet und gewartet werden. Dieses Service wird dann in den entsprechenden Softwarekomponenten eingebunden. Die Aufteilung in kleinere Einheiten reduziert des Weiteren auch die Komplexität von größeren Softwaresystemen erheblich. Dies führt zu einer Reduktion der Durchlaufzeiten für Änderungen in der Wartung oder beim Hinzufügen neuer Funktionalität. Ein weiterer erheblicher Vorteil ist, dass neue Geschäftsprozesses nun durch das simple zusammenfügen von bestehenden Services unterstützt werden können, dies führt zu einer höheren Agilität in der Softwareentwicklung und des Weiteren des ganzen Unternehmens. SOA ist also keine spezielle Technologie oder eine spezifische Software, sondern ein Paradigma um IT Umgebungen zu strukturieren.

SOA spezifische Services verfügen über folgende Eigenschaften:

- Ein Service ist eine abgeschlossene und eigenständige Einheit.
- Services sind verteilt, müssen daher über das Netzwerk zugänglich sein.
- Services müssen ihre Interfaces öffentlich zugänglich machen.
- Services müssen Interoperabilität gewährleisten.

- Services müssen auffindbar sein, daher es muss eine zentrale Stelle geben in der man nach Services suchen kann.

- Services müssen sich dynamisch, daher zur Laufzeit, binden lassen.

In der Realität werden selten all diese Vorgaben eingehalten[soa07]. Vielmehr besteht der Fokus darin einzelne Services zu funktionsfähigen Geschäftsprozessen zusammenzufügen. Die Services sind dabei grob granular und verfügen über eine geringe Bindung untereinander. Ein konzeptioneller Überblick einer solchen Architektur ist in der Abbildung 2.1 dargestellt.

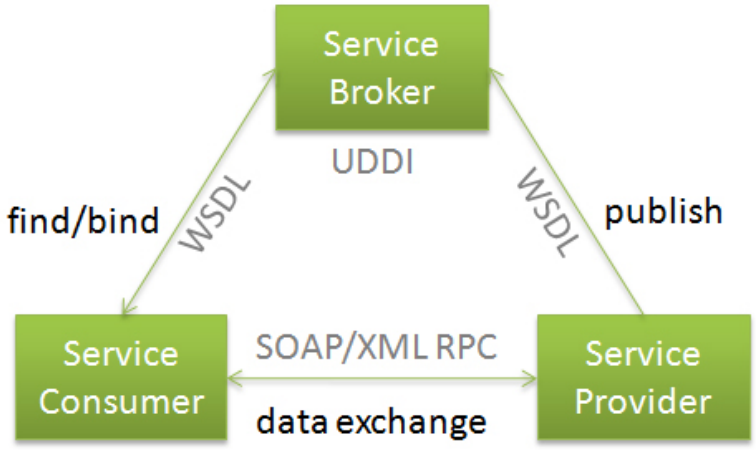

Abbildung 2.1: SOA Architektur

Die Grafik zeigt, an Hand von Technologien die bei Web Service fokussierten Umgebungen eingesetzt werden, ein Beispiel für eine konkrete Implementierung. Die Abbildung gliedert sich in drei Teile. Der Service Provider bietet das Service an und registriert sich damit beim Service Broker. Dieser bietet einen Dienst an, mittels dem der Consumer dieses Service finden kann. Der Client kann dieses Service nun suchen und binden. Die eigentliche Transaktion findet dann ohne den Broker, nur mehr zwischen Provider und Consumer statt.

Abbildung 2.2 verdeutlicht den Unterschied einer Systemlandschaft vor und nach der Umstellung auf SOA. Die linke Seite zeigt Prozesse, Aktivitäten und Daten die auf unterschiedliche unabhängige und inkompatible Anwendungen aufgeteilt sind. Die Ausführung eines Geschäftsprozesses erfordert, dass der Benutzer zwischen diesen Anwendungen wechseln und unterschiedliche Datenstrukturen kennen muss. Abgesehen von diesen Aspekten ist die Wartung durch replizierte Funktionalität aufwendiger und teurer, als im Vergleich zur rechten Seite der Grafik. Diese illustriert eine sauber nach SOA Richtlinien gestaltete IT Umgebung.

Durch das Zusammenfügen von unterschiedlichen Services ist es dem Benutzer nun möglich seine Aufgaben in einer einheitlichen Applikation durchzuführen. Die wiederverwendbaren Services müssen nur mehr an einer Stelle gewartet werden, dies reduziert die Kosten und die benötigte Zeit erheblich.

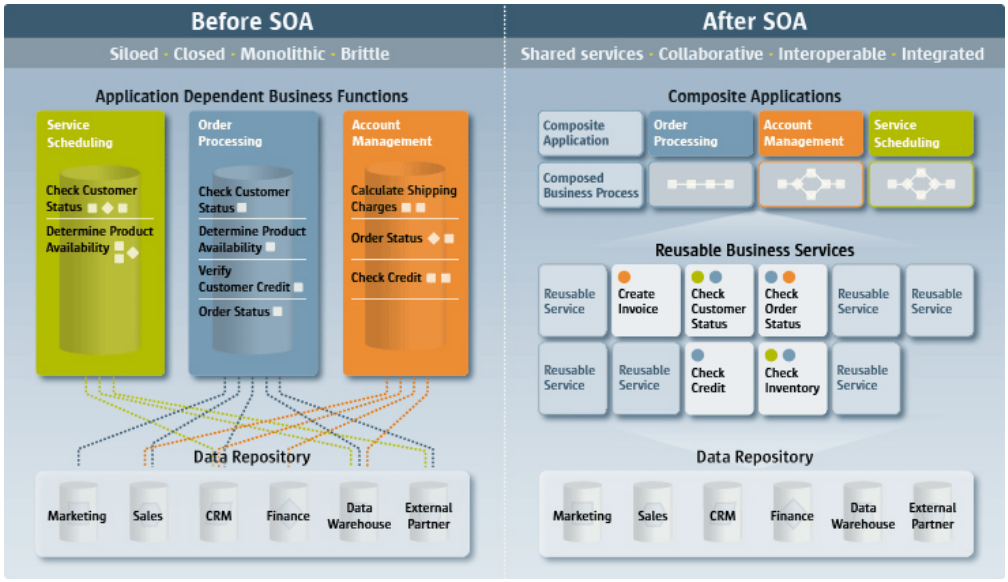

Abbildung 2.2: Bevor und nach der Einführung von SOA [SUN]

Monitoring & Reporting

Monitoring bzw. Reporting von verteilten Systemen lässt sich laut [Mas95] in drei unterschiedliche Aktivitäten aufschlüsseln:

Erzeugung Zustandsinformationen, z.B.: der Aufruf einer Methode, das Empfangen einer Nachricht, etc, werden während der Ausführung eines Prozesses identifiziert und lösen die Erzeugung eines Berichtes über diese Aktivität aus. Mit Hilfe dieser lässt sich der Ablauf eines Prozesses in einem verteilten System verfolgen.

Verarbeitung Die im ersten Schritt erzeugten Daten müssen zentral gesammelt und verarbeitet werden. Einzelne Berichte müssen zu zusammenhängenden Einheiten verknüpft werden.

Visualisierung Die gewonnenen Daten müssen dem Benutzer präsentiert werden und zwar in einer Form die ihn in seiner Tätigkeit unterstützt.

Generell kann so auch die Architektur eines Monitoring Systems in Komponenten eingeteilt werden[Wer00]:

In der Datenakquisitionsschicht werden relevante Ereignisse in dem System registriert das unter Beobachtung steht. Kommt der Steuerfluss eines Systems an einem Beobachtungspunkt vorbei so wird überprüft ob alle Bedingungen für das Auslösen eines Ereignisses gegeben sind. Trifft dies zu wird ein Ereignisbericht erstellt und Metainformationen über den Kontext hinzugefügt. Diese können zum Beispiel der Ort, der Zeitpunkt und einige Zustandsinformationen sein.

In der Schicht Datenzugriff werden die Daten gespeichert. Es wird entschieden wo welche Daten persistiert werden. Diese sind später für die Analyse von Bedeutung. Die internen

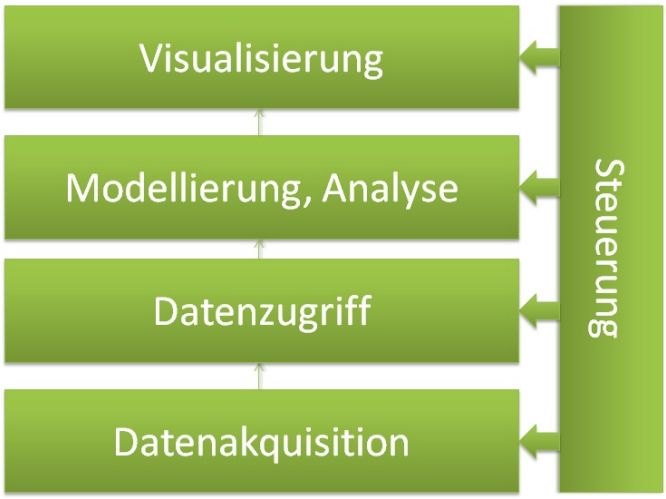

Abbildung 2.3: Monitoring Komponenten

Strukturen werden hier allerdings von den darüber liegenden Schichten verborgen, dadurch wird eine Entkopplung zwischen Messung und Auswertung erreicht. Je nach Anforderung werden diese Daten direkt auf dem Knoten gespeichert, gesammelt und später an eine zentrale Stelle übermittelt oder direkt ohne temporäre Zwischenspeicherung weitergeleitet. Hier sind vor allem Performanceüberlegungen zu beachten.

Die Modellierung und Analyseschicht ermittelt den globalen Zustand des zu beobachteten Systems aus einer Sequenz von Ereignissen. Um dies zu erreichen werden die Ereignisberichte von der darunter liegenden Schicht übernommen und in das Modell eingebettet. Hier können verschiedene Ereignisattribute auf Plausibilität geprüft werden um einen inkonsistenten Zustand des Systems zu vermeiden. Des Weiteren können Filter verwendet werden um die Anzahl der Ereignisse die verarbeitet werden müssen zu reduzieren. Durch das zusammensetzen von bereits zusammengesetzten Ereignissen lassen sich hier verschiedenste Abstraktionsschichten realisieren.

Die Visualisierung kann in unterschiedlichen Formen erfolgen, zum Beispiel sind textuelle oder grafische Repräsentationen der Daten möglich. Häufig werden hier Gantt Diagramme zur Darstellung von Aktivitäten, oder unterschiedlichste Statistiken eingesetzt[HE91]. Auch das Verhalten entlang einer Zeitachse kann hier animiert werden auf dieser sich der Benutzer vor und zurück bewegen kann.

Die Steuerungsschicht dient lediglich dazu um die anderen Schichten den Bedürfnissen entsprechend anzupassen. Typische Aufgaben sind Änderungen bei den Sensoren bzw. Änderungen bei der Strategie der Datensammlung.

Das Monitoring von Systemen bringt Vorteile in verschiedensten Bereichen[Sch95]:

- Performance Evaluierung
- Performance Verbesserungen
- Debugging und Testen

- Kontrolle von Systemen
- Sicherheit
- Überprüfung der Korrektheit
- Verlässlichkeit

Ereignisse die in verteilten Systemen auftreten lassen sich in drei Kategorien unterteilen:

- Hardware-Level Events
- Prozess-Level Events
- Anwendungsabhängige Events

Hardware-Level Events werden durch Ereignisse ausgelöst die auf Hardwareebene stattfinden wie Speicherlesefehler, oder I/O Fehler. Ereignisse auf Prozessebene lassen sich nur von außerhalb eines Prozesses betrachten, wie sie zum Beispiel bei der Terminierung auftreten. Das Monitoring auf dieser Ebene zählt wohl zu den Wichtigsten im Bereich der verteilten Systeme, denn nur hier sind Kommunikationsfehler, etc. zu identifizieren. Ereignisse auf Anwendungsebene sind sehr fein granular, zum Beispiel die Änderung von Variableninhalten oder der Aufruf einzelner Methoden. Generell können drei verschiedene Implementierungskonzepte unterschieden werden: Hardware-Monitore, Software-Monitore und Hybrid-Monitore.

Hardware-Monitore benutzen dezidierte Hardware um Ereignisse in verteilten Systemen zu observieren. Die Erkennung der Ereignisse erfolgt durch die hierbei durch die Beobachtung des Steuerbusses, des Adressbusses und des Datenbusses. Es werden zum Beispiel Daten über Lese- und Schreibvorgänge und über Speicheradressen aufgezeichnet[BMN89]. Hardware-Monitore haben den Vorteil das beobachtete System nicht oder nur wenig zu stören. Dies wird als nonintrusive bezeichnet. Dem gegenüber stehen auch einige Nachteile. Jede Änderung endet zwangsläufig in einer Hardwareneuentwicklung. Diese Anpassungen sind teuer und benötigten viel Zeit. Daraus resultiert auch eine geringe Flexibilität. Weitere negative Faktoren sind die geringe Portabilität und die relativ hohen Kosten. Da der Hardware-Monitor des Weiteren maschinennahe Daten liefert, müssen diese noch aufwendig aufbereitet werden um dem Benutzer zur Verfügung zu stehen[Mas95].

Software-Monitore sind reine Softwareimplementierungen. Das Programm, das es zu beobachten gilt wird mit zusätzlichen Aufrufen versehen die das Monitoring ermöglichen. Damit verbunden ist auch, dass Ressourcen die eigentlich dem System zur Verfügung stehen sollten für Monitoring Zwecke abgezweigt werden. Dennoch bietet die softwarebasierte Variante einige wesentliche Vorteile. Die Portabilität und die Flexibilität sind deutlich erhöht. Anpassungen können in der Software vorgenommen werden und erfordern keine Neuentwicklung von Hardware. Das reduziert die Kosten erheblich. Die durch Software-Monitore gewonnenen Daten werden auf Anwendungsebene beobachtet. Daraus folgt, dass sich das Abstraktionsniveau der Daten bereits in einem verwendbaren Bereich befindet.

Hybride Ansätze kombinieren die Vorteile von Hardware- und Software-Monitoren. Es werden zwar spezielle Anweisungen in den Programmcode der zu beobachteten Anwendung

eingefügt, diese werden allerdings von spezieller Hardware erkannt und ausgewertet umso die Störungen des Programms so gering wie möglich zu halten. Die Auswertung erfolgt schließlich wieder mittels Software.

QoS-Parameter

Quality of Serivce Parameter werden ein immer wichtigeres Thema im Bereich der verteilten Systeme[VKBG94]. QoS Management muss auf allen Ebenen stattfinden, dies wird in Abbildung 2.4 verdeutlicht[MMVA02]. Besonders wenn der Endbenutzer eines verteilten Prozesses ein menschlicher Akteur ist, müssen zum Beispiel Antwortzeiten über alle Ebenen hinweg eingehalten werden um eine flüssige Interaktion nicht zu stören.

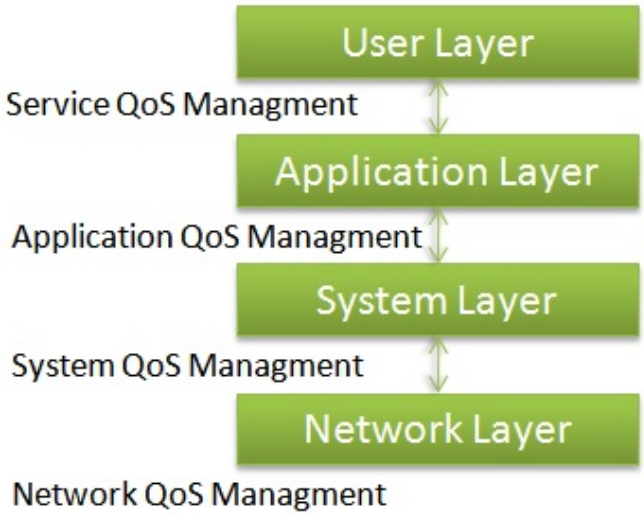

Abbildung 2.4: QoS Schichtenmodel[MMVA02]

Im Folgenden werden ausgewählte QoS-Parameter beschrieben um einen Überblick über diese Thematik zu gewinnen:

Effizienz ist die Eigenschaft eines Systems unter bestmöglicher Ausnutzung aller ihm zur Verfügung stehender Ressourcen seine Aufgaben zu erfüllen[PB96]. Ressourcen können hier bei zum Beispiel Zeit, Speicher oder Rechenkapazität sein. Folgende Merkmale lassen sich hier ermitteln[Mas95]:

Ressourcenbedarf Der Ressourcenbedarf beschreibt die Ausnutzung der CPU, Speicher oder Bandbreite

Durchsatz Der Durchsatz ist jene Datenmenge die in einer bestimmten Zeitspanne abgearbeitet werden kann.

Antwortzeit Die Antwortzeit ist jene Zeit die bis zur vollständigen Erledigung eines Auftrags benötigt wird.

Latenz Die Latenz ist die von einem physikalischen Gerät verursachte Verzögerung.

Effektivität misst die Wirksamkeit einer bestimmten Maßnahme[Bla67]. Daher gilt es zu messen ob die getätigten Schritte auch jeweils einen Beitrag zum Erreichen des Gesamtzieles leisten.

Einer der wichtigsten QoS Parameter von verteilten Systemen ist die Skalierbarkeit. Skalierbarkeit beschreibt die Fähigkeit eines Systems durch das Hinzufügen von neuen Ressourcen die Laufzeit zu verkürzen[JW98]. Die Skalierbarkeit wird durch unterschiedliche Faktoren beeinflusst. Je mehr Arbeitsschritte des Systems parallelisierbar sind umso besser skaliert das Gesamtsystem. Folgende Faktoren üben darüber hinaus Einfluss darauf aus[DG92]:

Das Starten(Startup) zusätzlicher Prozesses benötigt Ressourcen und Zeit. Sollten zum Beispiel tausende Prozesse gestartet werden kann dies einen beträchtlichen Anteil der Systemressourcen verbrauchen.

Interferenzen zwischen den Prozessen verlangsamen zusätzlich die Durchführungszeit. Je mehr Prozesse auf gemeinsam genutzte Ressourcen zugreifen müssen umso länger muss jeder Prozess warten diesen zu erhalten. Des Weiteren müssen diese Prozesse synchronisiert werden, dies führt wiederum zu weiterem zusätzlichem Aufwand.

Je mehr parallele Schritte durchgeführt werden umso kleiner ist jeder Schritt. Dies führt allerdings zu einem Engpass an dem Schritt der am langsamsten ausgeführt wird.

Die Zuverlässigkeit setzt sich aus den beiden Parametern der Verfügbarkeit und der Korrektheit zusammen[PB96]. Die Zuverlässigkeit zählt zu einem der wichtigsten Designparametern von verteilten Systemen. Durch die hohe Anzahl an Bereichen in den Fehler auftreten können, muss auf diese Eigenschaft besondere Rücksicht genommen werden. Messbare Attribute die hier verwendet werden sind oft die Zeit bis zum Auftreten des ersten Fehlers (mean time to first failure) oder die Zeit zwischen den Fehlern(mean time between failure)[RKH88]. Im Gegensatz zu Hardware die meist eine konstante Fehlerrate aufweist, kann es bei Software durch Modifikationen zu unterschiedlichen Verhalten kommen.

Fehlertoleranz bezeichnet die Fähigkeit eines Systems trotz auftretender Fehler, sowohl im Hardware- als auch im Softwarebereich, die weitere Nutzung zu ermöglichen[Gro94][GH99]. In verteilten Systemen wird Fehlertoleranz vor allem durch Redundanz erreicht. Sowohl Hard- und Software als auch alle benötigten Informationen werden redundant gehalten. Folgende Maßnahmen erhöhen die Fehlertoleranz[Nel90][Wer00]:

Fehlererkennung Um mit einem Fehler umgehen zu können muss dieser erfolgreich vom System als solcher erkannt werden.

Schadensbegrenzung Ein Fehler darf sich nicht auf andere Komponenten im System ausbreiten.

Diagnose Wurde ein Fehler festgestellt musst dieser einer Komponente zugeordnet und analysiert werden.

Fehlerbehandlung Der erkannte Fehler muss schließlich entsprechend entfernt werden, zum Beispiel durch den Austausch einer Komponente

Wiederanlauf Der Systemzustand muss wieder hergestellt und die Verarbeitung fortgesetzt werden.

Speziell in verteilten Systemen können an unterschiedlichsten Stellen Fehler auftreten, zum Beispiel Netzwerkfehler, Softwarefehler, zu starke Netzwerklast oder Nichterreichbarkeit eines Datenbankservers. Im Folgenden werden mögliche Fehlerursachen näher betrachtet[Bir97]:

Halting Failure Ein aktiver Prozess terminiert oder stürzt ab ohne eine fehlerhafte Operation auszuführen. Diese Art von Fehlern ist nur über ein Timeout festzustellen.

Fail-stop Failure Ein Prozess terminiert und steht nicht mehr für weitere Eingaben zur Verfügung oder reagiert nicht mehr auf diese. Es handelt sich um einen entdeckbaren Spezialfall des Halting Failure.

Send-omission Failure Dieser Fehler tritt auf wenn laut definierter Programmlogik eine Nachricht hätte gesendet werden müssen, dies aber nicht geschieht.

Receive-omission failures Ähnlich dem Send-omission Failure nur tritt dieser Fehler beim Empfänger auf. Die Nachricht wird verworfen oder enthält zum Beispiel nicht valide Daten.

Network failures Fehler im Netzwerk liegen dann vor, wenn die Nachrichten beim Austausch zwischen den Prozessen verloren gehen.

Timing failure Dieser Fehler tritt auf wenn die Uhrzeit der einzelnen Subsysteme nicht mehr synchron ist. Dadurch kann es zu unvorhersehbaren Timeouts und abweichendem Verhalten kommen.

Byzantine failures In diesem Fall liefert ein Prozess falsche oder teilweise falsche Resultate. Der Prozess läuft allerdings ohne Anzeichen auf fehlerhaftes Verhalten. In diesen Bereich fallen hauptsächliche unentdeckte Fehler der Software.

Aufwand der zur Fehlertoleranz eines Systems betrieben wird schlägt sich auch in anderen QoS Parametern nieder wie der Zuverlässigkeit und der Verfügbarkeit.

Ein weiterer QoS-Parameter ist die Korrektheit des Systems. Diese beschreibt den Grad der Übereinstimmung des Systems mit der Spezifikation laut der es entwickelt wurde[PB96]. Eine vollständige Fehlerfreiheit ist weder bei der Analyse, Spezifikation noch bei der anschließenden Implementierung eines Systems möglich. Es existieren aber unterschiedlichste Testfahren die mittels dynamisch, statischer oder formaler Methoden versuchen die Fehleranzahl soweit wie möglich zu senken. In verteilten Systemen kann jede Komponente für sich auf Korrektheit geprüft werden. Allerdings ist ein Integrationstest bei dem die Funktionsfähigkeit des gesamten Systems überprüft wird unerlässlich.

Fehler in verteilten Systemen sind auf Grund der notwendigen Kommunikation und der Verteilung oft nur schwer zu lokalisieren. Es ist besonders wichtig verschiedene Aktionen während des Zusammenspiels der Komponenten genau zu analysieren und zu testen. Dies kann allerdings nur mit einem dynamischen Testansatz sinnvoll durchgeführt werden.

Der letzte ausgewählte QoS-Parameter ist der Begriff der Sicherheit. Unter ihn Fallen Bereiche wie die Vertraulichkeit, Integrität aber auch die Verfügbarkeit.

Testen

Im Folgenden werden die Lebensphasen eines Lebenszyklus eines Softwaresystems kurz erklärt um klarzustellen in welchen Phasen Testen welchen Nutzwert hat. Abbildung 2.5 illustriert die verschiedenen Abschnitte.

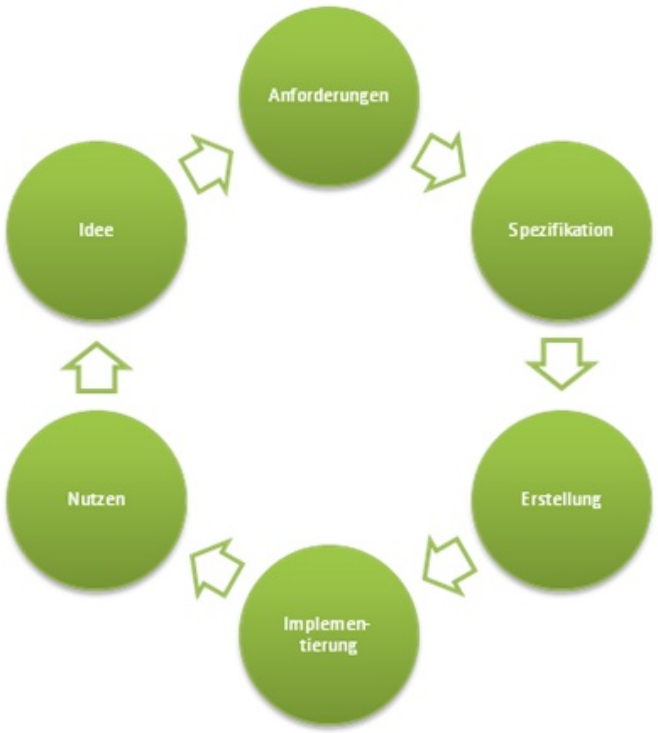

Abbildung 2.5: System Lifecycle vgl.: [Lon08]

Nach der initialen Idee kommt die Phase der Anforderungensdefinition. Business Analysten müssen genau festlegen welche Bereiche durch die Software abgedeckt werden bzw. welche Funktionalität umgesetzt wird. Diese muss exakt festgehalten und dokumentiert werden. Oft werden hier Geschäftsprozesse, Transformationen oder verschiedene Datenflüsse aus einem bestimmten Blickpunkt festgehalten. In dieser Phase kann noch nicht sinnvoll getestet werden. Tests, wenn man überhaupt davon sprechen kann, können höchstens in Form von Reviews oder ähnlichem ansetzen. Die nächste Phase der Spezifikation definiert bereits Teile des Systems. Das Systemdesign wird erstellt, allerdings beruhen die meisten Details noch auf Annahmen oder Vermutungen. Tests in dieser Phase können in Form von Static Tests erfolgen um sicherzustellen, dass das Design alle Anforderungen korrekt abdeckt. In der Erstellungsphase können erste dynamische Tests durchgeführt werden da die Implementierung bereits begonnen hat. Hier kann das Testteam bereits Testpläne erstellen, Tests erstellen und erste Tests durchführen. Fehler die dadurch in dieser Phase bereits bereinigt werden wirken sich so nicht auf spätere Phasen aus. In der Implementierungsphase ist das System bereit in verschiedenen Umgebungen getestet zu werden. So können Systemtests und Akzeptanztests durchgeführt werden. Des Weiteren

kann das System auf in einer Produktionsnahen Umgebung getestet werden. Hier beginnen Fachtester ihre Arbeit und testen auf fachliche Richtigkeit der Implementierung. Auch die Tests der nichtfunktionalen Anforderungen werden hier durchgeführt, wie zum Beispiel Performance oder Sicherheitstests. In der letzten Phase wird schließlich die Nutzung des Systems in der Produktionsumgebung fokussiert. Das Testen sollte hier abgeschlossen sein und die Risiken für Fehler auf ein wirtschaftlich sinnvolles Maß reduziert sein.[Lon08]. In Anlehnung das das V-Model wird in Abbildung 2.6 dargestellt wann welcher Schritt der Tests stattfinden.

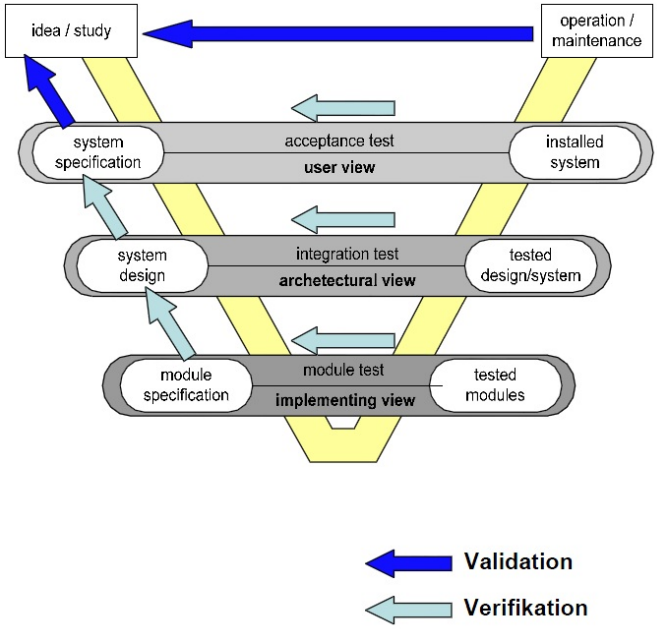

Abbildung 2.6: Validation und Verifikation[Rud10]

Die Abbildung verdeutlicht die unterschiedliche Fragestellung die auf den unterschiedlichen Ebenen gelöst wird. Während auf der obersten Ebene(blauer Pfeil) die Frage im Vordergrund steht ob man das richtige Produkt gebaut hat, ist in den Unterbereichen die Spezifikation im Fokus, daher ob das Produkt richtig gebaut wurde.

Tests können des Weiteren in unterschiedliche Levels eingeteilt werden[Rud10]:

1. private Tests der Entwickler
2. Unit Tests
3. Komponenten Tests
4. Integrationstests
5. System Level Tests
6. Regression Tests (Testautomatisierung)

7. Akzeptanztests

Private Tests finden vor der Auslieferung einer Komponente zum Beispiel an andere Tester statt. Es gibt keine dokumentierten Testfälle oder ähnliches. Allerdings lassen sich hier durch Coverage Analysen Verbesserungen in der Testabdeckung erzielen. Unit bzw. Komponenten Tests haben als Ziel die Identifizierung von Fehlern während der Implementierung. Es wird jeweils ein bestimmter Input spezifiziert und darauf ein bestimmter Output erwartet. Sie ermöglichen eine frühe Identifikation von Fehlern und deren Lokalisation. Integrationstests haben die Aufgabe die Interaktion zwischen den Komponenten auf Architekturebene zu überprüfen. Hier können verschiedene Integrationstrategien verfolgt werden, zum Beispiel Big-Bang, Top-Down, Bottom-up oder Build Integration, wobei die Empfehlung zu dem letzten Ansatz tendiert. System Level Tests haben als Ziel die Verifikation und die Validation des Gesamtsystems. Regressiontests sind Tests die bereits ausgeführt wurden aber die nach einer Änderungen oder Hinzufügen eines neuen Features erneut ausgeführt werden. Eng gekoppelt hiermit ist die Testautomatisierung. Tests die öfter durchgeführt werden müssen wie zum Beispiel im Zuge von Regressiontests können automatisiert werden. Dies führt zu einem Kostenvorteil da manuelle Tests erheblich teuer sind als automatisch ausgeführte Tests. Akzeptanztests überprüfen ob das System den Benutzeranforderungen gerecht wird.

In Hinblick auf das Testen von verteilten Systemen sind vor allem die Bereiche der Komponenten und Integrations- bzw. Systemtests interessant.

Testtechniken lassen sich in drei unterschiedliche Kategorien einteilen[Rud10]:

1. Black-box
2. White-box
3. Experience-based

Im Allgemeinen sind Black-Box Tests, Tests die keine Einsicht in das System haben, es also als Blackbox betrachten und nur auf den Input und den Output des jeweiligen Testes abzielen. Entsprechend sind auch die Techniken die dieser Kategorie zugeordnet sind darauf fokussiert Inputdaten zu liefern die eine möglichst große Testabdeckung ermöglichen. Solche Techniken sind zum Beispiel Äquivalenzklassen, Grenzwertanalyse oder Übergangstests(State transition testing)[CP98]. White-Box Tests hingegen haben Einsicht in das System, dies ermöglicht Fehler genau zu lokalisieren und neue Methoden anzuwenden. Typische White-Box Methoden sind zum Beispiel Statement Coverage, Decision Coverage oder Path Coverage Analysen. All diese Methoden geben den genauen Abdeckungsgrad der Testfälle über die Applikation die getestet werden soll in dem jeweiligen Bereich an. Experience-based verlässt sich beim Testen auf die Erfahrung des Testers. Hier ist es besonders wichtig, dass der Tester Erfahrung in dem jeweiligen Fachgebiet hat um zielgerichtet nach Fehlern suchen zu können.

Testdokumentation & Reporting

Im folgenden Abschnitt soll die Testdokumenation und Reporting im Testbereich näher erklärt werden. Hierbei wird auf den IEEE Standard for Software and System Test Documentation

zurückgegriffen[IEE08]. Abbildung 2.7 stellt einen Überblick über die Testfalldokumentation innerhalb eines Projektes dar. Eine genaue Beschreibung aller Elemente kann an dieser Stelle nicht erfolgen, da dies nicht im Scope dieser Arbeit liegt. Wichtige Elemente im Bereich dieser Arbeit sind vor allem die Reporting Knoten.

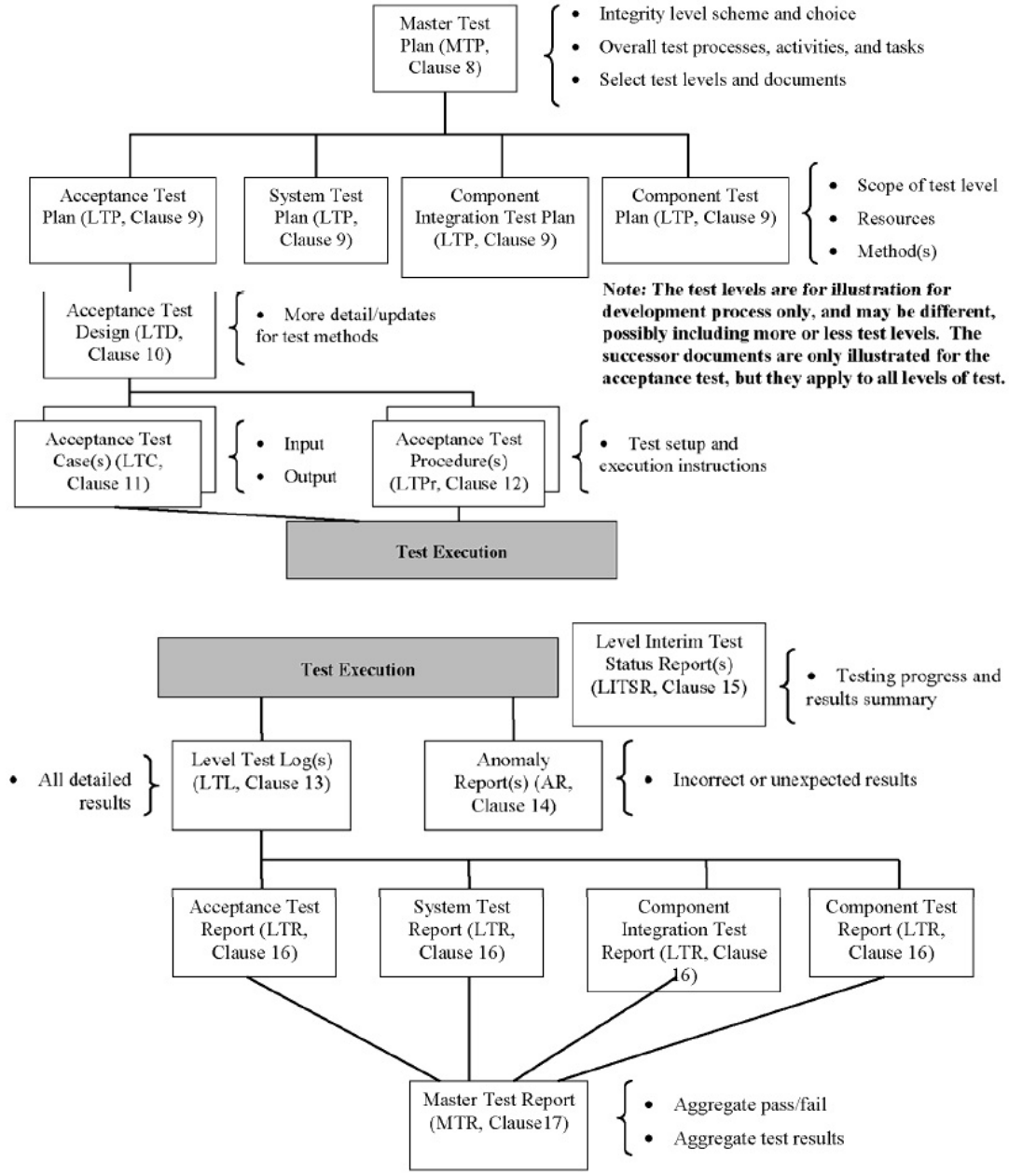

Abbildung 2.7: Test Dokumentation Überblick[IEE08]

Anomaly Reports, Level Interim Test Status Reports, Level Test Reports und Master Test

Reports werden im Weiteren näher beschrieben.

Der Zweck eines Anomaly Reports ist es, alle möglichen Ereignisse die während des Testprozesses auftreten und näher behandelt werden müssen. Für dieses Ereignis müssen Daten wie eine passende Beschreibung, Datum des ersten Auftretens oder die Auswirkungen festgehalten werden. Wichtig ist auch, dass der Status dokumentiert wird, damit im weiteren Projektverlauf mitverfolgt werden kann ob diese Anomalie behoben wurden. Level Interim Test Status Reports fassen die Resultate explizierter Testaktivitäten zusammen. Basierend auf diesen Ergebnissen können Auswertungen und Empfehlungen getätigt werden. Diese Reports können natürlich auf allen Testlevels erstellt werden. Wichtig hierbei ist es verschiedene Metriken, Änderungen vom Testplan und eine Zusammenfassung über den Teststatus zu bieten. Level Test Report kann als abschließender Level Interim Test Status Report gesehen werden. Der Master Test Report fasst schließlich alle anderen Reports zusammen. Er bietet auf deren Ergebnissen Auswertungen und gibt Empfehlungen über die weitere Vorgangsweise. In ihm werden Alle Resultate zusammengefasst dargestellt, Entscheidungen mit ihren Begründungen angegeben und Empfehlungen festgehalten.

Das Durchführen der einzelnen Tests wird ebenfalls schriftlich festgehalten. Hierbei sind bei der Planung ebenfalls unterschiedliche Ebenen zu beachten.

Master Test Plan (MTP) Der Master Test Plan koordiniert die gesamte Testplanung und das gesamte Testmanagement für die unterschiedlichen Levels der Tests. Dies kann sich auf ein Projekt beschränken, oder auch projektübergreifend angelegt sein. Der MTP legt den Gesamtumfang der Testaktivitäten fest, die Ressourcen bzw. die benötigte Arbeit und die dazu benötigte Zeit. Des Weiteren legt er die Abhängigkeiten unter den einzelnen Aktivtäten fest und muss eventuelle Risiken identifizieren und analysieren. Je nach Komplexität der Software legt er die Anzahl der Tests, die auszuführenden Arbeiten und deren Dokumentation fest. Typische Inhalte eines Master Test Planes sind eine Zusammenfassung der benötigten Ressourcen, Verantwortlichkeiten und eingesetzt Tools sowieso angewandte Metriken.

Level Test Plan (LTP) Level Test Plans definieren innerhalb ihres Bereiches die verwendeten Ansätze, Ressourcen und die zeitliche Planung für die Durchführung der Testaktivitäten. Sie definieren darüber hinaus genau welche Features von welchen Personen getestet werden und welche Risiken damit verbunden sind. LTP werden zum Beispiel jeweils für Komponenten Tests, Integrationstests und Akzeptanztests erstellt.

Level Test Design (LTD) Das Level Test Design verfeinert die Ansätze für die verschiedenen Tests. Es werden darüber hinaus die zu testenden Features, die von einem Testfall getestet werden, genauer spezifiziert.

Level Test Case (LTC) Auf diesem Level wird jeder Testfall genau. Das erfordert, dass für jeden Testfall der genau Input festgehalten wird und der erwartet Output ebenfalls schriftlich festgehalten wird. Für jeden Testfall wird das genaue Testziel festgehalten und welche Vorrausetzungen erfüllt werden müssen, damit dieser korrekt durchgeführt werden kann. Dies können zum Beispiel bestimmte Systemeigenschaften sein oder andere Testfälle sollten Abhängigkeiten bestehen.

2.2 Technische Grundlagen

Das folgende Kapitel beschreibt Technologien bzw. Standards die im Bereich der verteilten Systeme bzw. in dem in dieser Arbeit erstellen Framework zum Einsatz kommen. Das Ziel ist es, einen generellen Überblick zu geben umso die dahinterliegenden Konzepte kennen zu lernen. Die einzelnen Abschnitte fassen jeweils nur für diese Arbeit relevante bzw. für das Verständnis wichtige Informationen zusammen. Es werden unterschiedliche technische Aspekte mit unterschiedlichem Fokus abgedeckt. Auch generelle Konzepte wie der REST Architekturstil werden hier erklärt.

XML

Die Entstehung der Extensible Markup Language(XML) beginnt in den 1970er Jahren. Damals wurde GML, das von Goldfarb entwickelt wurde, verwendet um technische Dokumente mit strukturierten Tags zu versehen. Daraus entwickelte sich schließlich XML als ein vom W3C anerkannter Standard um (semi-)strukturierte Daten zu repräsentieren. XML an sich ist eine vereinfachte Version von SGML(Standardized Generalized Markup Language), welches wiederrum 1986 zum ISO Standard erklärt wurde[Cha]. Der Fokus lag dabei auf der Vereinfachung des Standards, allerdings sollte gleichzeitig ein Großteil der Funktionalität beibehalten werden. In der heutigen Form wurde es schließlich 1996 von Sun Microsystms entwickelt und wurde anschließend 1998 in der Version 1.0 zur W3C Recommendation[W3Cd]. Die aktuelle Version liegt mit 1.1 vor. Ähnliche wie HTML, verwendet XML Tags(z.B.: <html>) um Elemente zu repräsentieren. Jede Anwendung kann so ihre eigene Struktur und Element spezifizieren und mittels XML Schema validieren lassen.

XML hat sich zu einem Standard in vielen Bereichen entwickelt, da es über eine Vielzahl von Vorteilen verfügt. Ein wichtiger Faktor stellt seine Plattformunabhängigkeit und die Möglichkeit der einfachen Einbindung in unterschiedlichste Programmiersprachen dar. Die Möglichkeit, dass jede Anwendung ihre eigenen Strukturen definieren kann ermöglich es den Entwicklern die optimale Repräsentation für ihr spezifisches Problem zu wählen. Es existieren darüber hinaus eine große Anzahl an unterschiedlichen XML Parsern wie SAX[SAX], JDOM[JDO] oder JAX[JAX]. XML lässt sich durch das Hinzufügen von Elementen leicht erweitern und verfügt so meist über Abwärtskompatibilität zu bestehenden Anwendungen. Dateien die XML Format verfügen durch die Tags über selbst beschreibende Elemente. Dies vereinfacht das Lesen und Verarbeiten für Menschen.

XML Dokumente die folgende Kriterien erfüllen gelten als well-formed[Ref]:

1. Jedes Dokument muss mit einer XML Deklaration beginnen.
2. Jedes XML Dokument enthält exakt ein Root Element.
3. Jedes Element hat einen Start und einen Endtag.
4. Die Elemente müssen korrekt geschachtelt sein.
5. Attribute müssen unter Anführungszeichen stehen.

6. Elementnamen sind case-sensitive.

7. Jede Entität die referenziert wird muss all diese Kriterien ebenfalls erfüllen.

Das Codebeispiel 2.1 zeigt ein einfaches Beispiel einer XML Datei. Später wird diese XML Datei verwendet um die Funktionalität von XML Schema und XSLT zu demonstrieren, auf Grund dessen die Struktur sehr einfach gehalten wurde. Das Root Element ist in diesem Beispiel Testcases, jedes Element wird von ‹›umschlossen. Das Root Element enthält vier weitere Elemente die geschachtelt in diesem enthalten sind. Die Testcase Elemente enthalten jeweils einen Wert, der zum Beispiel deren Titel oder Ähnliches enthalten könnte. Das Root Element enthält des Weiteren zwei Attribute Tester und Wichtigkeit. Auch die Unterelemente können jeweils wieder Attribute enthalten. Das letzte Element demonstriert die Möglichkeit eines leeren Elementes, bei dem eine entsprechend verkürzte Schreibweise möglich ist.

```
<?xml version="1.0" encoding="utf-8"?>
<Testcases Tester="Schakmann Rene" Wichtigkeit="2">
  <!-- Kommentar: Wichtige Testfaelle -->
  <Testcase>XML</Testcase>
  <Testcase>Web Services</Testcase>
  <Testcase>BPEL</Testcase>
  <Testcase/>
</Testcases>
```

Listing 2.1: XML Beispiel

Um den Unterschied zu HTML zu verdeutlichen befindet sich im Codebeispiel 2.2 valider HTML Code. Da die Elemente allerdings falsch geschachtelt sind handelt es sich hierbei nicht um gültiges XML. An dieser Stelle sei XHTML[W3Cc] erwähnt. XHTML Dokumente basieren auf HTML genügen allerdings den XML Regeln.

```
<b><i>Testcases Auflistung</b></i>
```

Listing 2.2: HTML Beispiel

XML Schema

XML Schema Dateien werden verwendet um für XML Dokumente eine bestimmte Struktur festzulegen. 2001 wurde die erste Version als W3C Recommendation publiziert und war die erste eigenständige Schemasprache für XML. Der Vorgänger DTDs(Document Type Definition) wurde aufgrund zahlreicher Vorteile der neuen Schemadefinition erfolgreich abgelöst. XML Schema Dateien sind leichter zu erweitern und erlauben die Definition neuer Strukturen ohne Nebeneffekte auf bestehende. Des Weiteren ist es ausdrucksstärker als sein Vorgänger[MNSB06]. XML Schemata sind selbst wiederum in XML geschrieben und erlauben die Verwendung von Namespaces. Dies erlaubt zum Beispiel die Verwendung von Datenstrukturen deren Namen ident sind aber unterschiedliche Domains haben gemeinsam in einer Anwendung. Diese Datenstrukturen können entsprechend Definiert und an den entsprechenden Stellen wiederverwendet werden. Dies erleichtert die Strukturierung der Schemas. Es werden 19 primitive Datentypen unterstützt wie boolean, string, decimal oder hexBinary. Mit deren Hilfe können komplexe Strukturen aufgebaut werden. Zusammenfassend lassen sich folgende Schlüsselfunktionen festhalten:

- defines elements and their attributes that can appear in a document
- specifies the number and order of child elements
- declares if an element is allowed to contain text
- defines data types for elements and attributes
- sets default and fixed values for elements and attributes

Die Verwendung von XML als Sprache für die Schema Dateien bietet darüber hinaus den Vorteil, dass existierende Tools mit XML Unterstützung verwendet werden können, um Schema Definitionen zu erstellen.

Das Codebeispiel 2.3 ist ein Beispiel für eine gültige Schema Definition für das XML Beispiel 2.1. Als erstes müssen immer die Version und das korrekte Encoding festgelegt werden, ausserdem wird die Verwendung von Namespaces demonstriert. Alle Elemente mit dem Präfix xs: gehören zu dem definierten Namespace. Danach wird genau ein Root Element 'Testcases' spezifiziert. Bei Dezem Element handelt es sich um einen komplexen Datentyp, da er noch weitere Unterelemente enthält. Im angeführten Beispiel enthält er eine Sequenz von Elementen die wiederrum als komplexer Datentyp deklariert sind. Die Unterelemente 'Testcase' sind hier direkt definiert. Es wäre allerdings auch möglich durch eine Referenz bereits bestehende Elemente einzubinden. Die Häufigkeit des Vorkommens eines Elementes kann mittels minOccurs bzw. maxOccurs definiert werden. In diesem Beispiel kann das Element theoretisch null bis unendlich mal vorkommen.

```
1  <?xml version="1.0" encoding="utf-8"?>
   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3    <xs:element name="Testcases">
       <xs:complexType>
5        <xs:sequence>
           <xs:element name="Testcase" nillable="true" minOccurs="0" maxOccurs="
               unbounded">
7            <xs:complexType>
               <xs:simpleContent>
9                <xs:extension base="xs:string">
                 </xs:extension>
11             </xs:simpleContent>
             </xs:complexType>
13         </xs:element>
         </xs:sequence>
15       <xs:attribute name="Tester" type="xs:string" />
         <xs:attribute name="Wichtigkeit" type="xs:int" />
17     </xs:complexType>
     </xs:element>
19 </xs:schema>
```

Listing 2.3: XML Schema Beispiel

Die Einbindung eines erstellen Schemas in eine XML Datei erfolgt wie der folgende Beispielcode demonstriert2.4.

```
<?xml version="1.0" encoding="utf-8"?>
<Testcases Tester="Schakmann Rene" Wichtigkeit="2" xmlns:xsi="http://www.w3.
    org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="schema.xsd">
...
</Testcases>
```

Listing 2.4: Include Schema

Um auch unerfahrenen Benutzern die Struktur leichter verständlich zu machen lassen sich XML Schema Definition leicht grafisch darstellen. Die Abbildung 2.8 zeigt eine solche grafische Repräsentation mittels Altova XMLSpy[1].

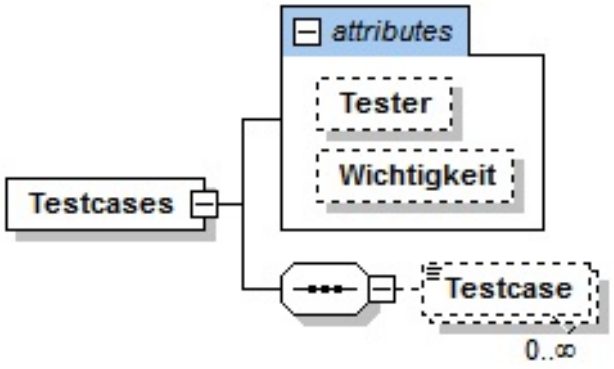

Abbildung 2.8: XSD Grafische Representation

DOM

Das Document Object Model(DOM) wird bei validen XML und HTML Dokumenten dazu verwendet um Elemente oder deren Inhalte anzusprechen[**?**]. Zurzeit liegt es in der Version 3 vor, die seit 2004 den Status Recommendation des W3C inne hat[W3Ca]. DOM profitierte stark von der Beliebtheit von JavaScript in den verschiedenen Browsern. Durch JavaScript und DOM konnten die bisher nur statischen Webseiten dynamisch verändert werden. Durch deren Einsatz können einzelne Teile eines Dokuments gezielt verändert, hinzugefügt oder gelöscht werden. Anfänglich wurden zwei unterschiedliche document object models von Netscape und Internet Explorer implementiert. DOM wurde schließlich als einheitlicher und plattformübergreifender Standard ausgearbeitet[W3Cb]. Bei validen HTML bzw. XML Dokumenten, wie in Codebeispiel 2.5 aufgeführt, kann das Document Object Model grafisch dargestellt werden wie in Grafik 2.9 veranschaulicht wird.

[1] http://www.altova.com/xml-editor/

```
<html>
    <head>
        <title>Link Liste</title>
    </head>
    <body>
        <h1>Links</h1>
        <a href="http://www.isis.tuwien.ac.at/">Click</a>
    </body>
</html>
```

Listing 2.5: DOM Beispiel

Wie bei jedem HTML Document ist die Wurzel des gesamten Inhaltes. Dieser beginnt erst mit dem Root Element wie es aus XML bekannt ist, in diesem Fall der html-Tag. Dieser enthält alle weiteren Elemente. In der Darstellung wird jedes von Ihnen als Kinder-Knoten dargestellt, wie auch jedes Attribut.

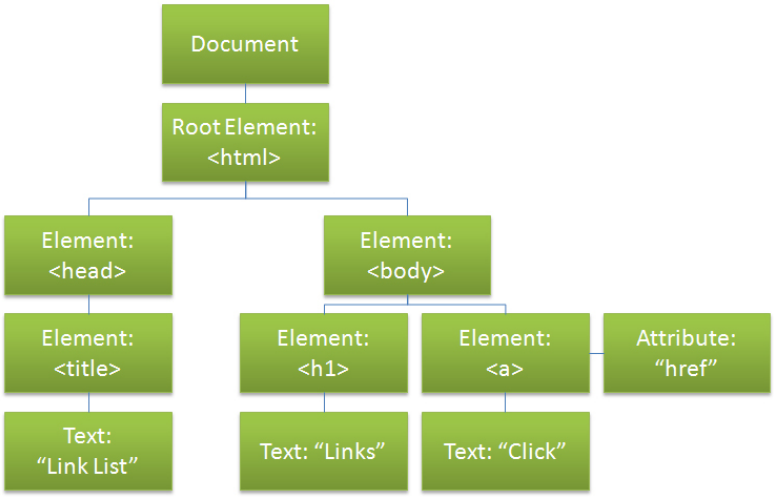

Abbildung 2.9: DOM Tree

XPath

XPath ermöglicht es nur gewisse Teile eines XML Dokumentes anzusprechen. Mittels verschiedener Suchparameter können so nur gewünschte Teile gefiltert und zurückgegeben werden. XPath liegt in der aktuellen Version 2.0 vor und wurde erstmals 1999 in der Version 1 eine W3C Recommendation[W3Ce]. Neben den Suchfunktionen enthält es ebenfalls die Möglichkeit für simple Manipulationen von Strings. Für die Adressierung der einzelnen Bereiche verwendet XPath Pfadausdrücke des darunter liegenden DOM Baumes. Unter der Berücksichtigung von Namespaces unterscheidet es weiter zwischen drei verschiedenen Knotenarten: Elemente, Attribute und Textknoten. Ausgewertete XPath Ausdrücke können unterschiedliche Objekte

zurückliefern wie NodeSets, Boolean, Nummern oder Strings. Einige XPath Standardausdrücke sind im Folgenden aufgelistet:

/ Wählt das Root Element aus

.. Wählt das Elternelement aus

Elementname Wählt ein Element mit dem entsprechenden Namen aus

@ Wählt ein bestimmtes Attribut aus

child:node() Wählt alle Kinderknoten aus.

Beispiele für korrekte XPath Ausdrücke werden in Codebeispiel 2.10 gegeben. In Anlehnung an unser XML Beispiel2.1 würde der erste XPath Ausdruck den ersten Testcase auswählen, der Zweite würde alle Attribute mit dem Namen Wichtigkeit selektieren und der dritte würde schließlich alle Testfälle zurückgeben.

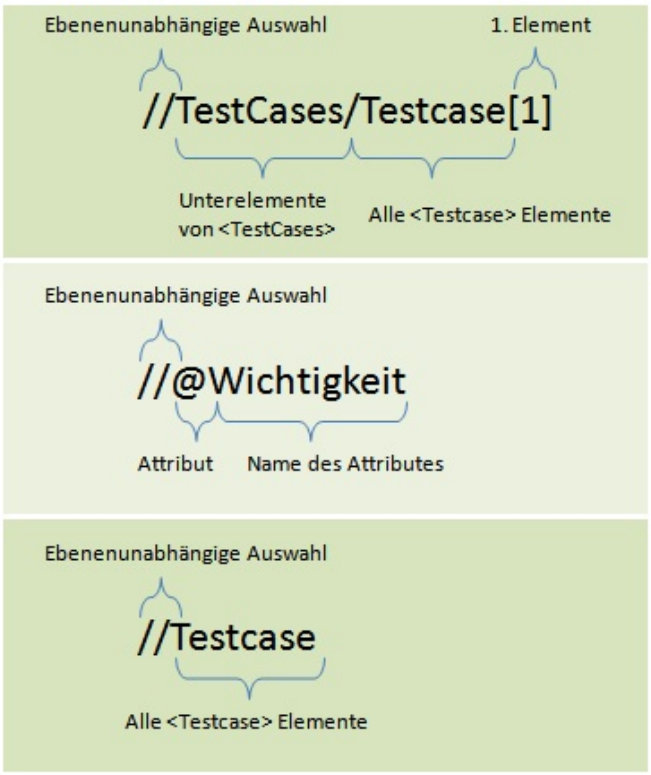

Abbildung 2.10: XPATH Beispiel

Web Services

Geschäftsprozesse geben die Rahmenbedingungen für die modernen IT Landschaften vor. Da Unternehmen nicht mehr geschlossen für sich arbeiten sondern über Unternehmensgrenzen hinweg aggregieren muss auch die IT diesen Anforderungen gerecht werden und einen Standard zur Kommunikation über verschiedene Unternehmen hinweg bereitstellen. Wichtig hierbei ist es, dass alle beteiligten Parteien dieselbe Sprache sprechen. EDI (Electronic Data Interchange)[UNE] war in den 80ern einer der ersten Versuche diesen Standard zu schaffen. Nachfolgestandards wie CORBA (Common Object Request Broker Architecture) von der Object Management Group[Hen08] versuchten ebenfalls diese Problematik zu adressieren. Dennoch konnten sich diese Ansätze genauso wie neuere Versuche wie RMI (Java Remote Method Invocation) nicht richtig durchsetzen. Mit dem Aufgekommen des Internets mit HTTP und dem darunterliegenden TCP/IP wurde nun ein offener Standard für die Kommunikation zugänglich. Das W3C publizierte darauf im Jahr 2000 einen XML basierenden Standard mit Technologien wie SOAP und WSDLs. Unterstützt durch Microsoft und IBM etablierten sich Web Services schnell als Standard.

Das W3C definiert ein Web Service als ein Stück Software, das auf einer Maschine läuft und über das Netzwerk mit anderen Maschinen kommuniziert und interagiert. Dessen Interface muss in einem maschinen-lesbaren Format, der Web Service Description Language (WSDL) vorliegen. Die Interaktion der Services erfolgt über definierte Simple Object Access Protocol (SOAP) Nachrichten. Als Standard hat sich hier das Hyper Text Transfer Protocol (HTTP) etabliert. Die Nachrichten werden mittels XML serialisiert[W3Cf].

Die Verwendung von Web Services verfügt über einige Vorteile. Durch die Offenheit des Standards besteht keine Abhängigkeit mehr von einem gewissen Lieferanten. Das gefürchtete Vendor Lock lässt sich so umgehen. Web Services sind per Definition Plattform unabhängig. Dadurch ist eine hohe Interoperabilität zwischen unterschiedlichen Systemen gewährleistet. Durch die Verwendung von WSDL und der Definition des Austauschformates bzw. der Datenstruktur muss der Client keinerlei Informationen über die tatsächliche Datenstruktur besitzen die dem Service zu Grunde liegt. Ein großer Vorteil ist die Verwendung von HTTP und die dadurch einfache Integration in bestehende Netzwerk. Dies kann aber auch als Nachteil gesehen werden, wenn man Sicherheitsaspekte in Betracht zieht. Weitere Nachteile sind die geringere Performance verglichen mit stärker gekoppelten Systemen wie CORBA[Irm02].

Ein wichtiger Aspekt der Web Services ist die standardisierte Beschreibungssprache der Interfaces. Für diesen Zweck wurde WSDL entwickelt. Ursprünglich von Microsoft und IBM vorangetrieben ist es mittlerweile eine W3C Recommendation die seit 2007 in der Version 2.0 vorliegt. Die am meisten genutzte Version ist allerdings WSDL 1.1, daher werden alle folgenden Beschreibungen auf diesen Standard referenzieren.

WSDL Dokumente beschreiben Web Services als eine Sammlung von Endpunkten. Die Definition der Services erfolgt mittels folgender Elemente:

- Types: Datentypen werden mittels XML Schema Dateien beschrieben.
- Message: Ein- und ausgehende Nachrichten sind eine strukturierte Ansammlung von Datentypen.
- Operation: Operationen die hier definiert werden können von Clients aufgerufen werden.

- Port Type: Ein Port Typ besteht aus einer oder mehrerer Operationen.
- Binding: Das Binding verbindet einen abstrakten Porttyp mit einem konkreten Protocol wie HTTP.
- Port: Ein Port ist die Verbindung aus einem bestimmten physischen Port und einer Netzwerkadresse.
- Service: Das eigentliche Web Service fasst alle anderen Bereiche zu einem Service zusammen.

Abbildung 2.11 verdeutlicht das Zusammenspiel der verschiedenen Elemente.

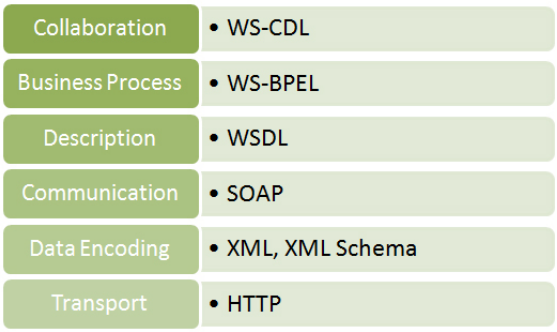

Abbildung 2.11: Web Services Stack

Für den Scope dieser Arbeit sind nur die unteren fünf Layer von Interesse. Die Basis, der Transportlayer, ist standardmäßig mittels HTTP realisiert. Allerdings sind hier auch andere Übertragungsprotokolle wie SMTP möglich. Das Data Encoding erfolgt mit den bereits vorgestellten Technologien XML und XML Schema. Die Kommunikation erfolgt mittels SOAP. Die Service Definition bzw. die Interfacebeschreibung verwendet WSDL zur genauen Spezifikation. Der letzte, für diese Arbeit wichtige, Layer ist der Business Prozess Layer. Mittels BPEL (Business Prozess Execution Language) können Web Services zu funktionierenden Business Prozesses zusammengefügt werden. Im nächsten Abschritt wird dieser Standard näher erklärt.

REST

Das Prinzip von Representational State Transfer (REST) wurde erstmals im Jahr 2000 von Roy Fielding erwähnt[Fie00]. REST versteht sich als ein bestimmter Softwarearchitekturstil für verteile Systeme. Es werden hier vor allem einfache Schnittstellen unter Verwendung des HTTP Protokolls adressiert. Der Architekturstil beschreibt folgende sechs Elemente[Fie00]:

Client Server Die Trennung von Client und Server stellt einen der wichtigsten Punkte dar. Durch die Aufspaltung der Zuständigkeiten kann die Implementierung des Servers unabhängig vom Userinterface gehalten werden. Diverse Serverimplementierungen können geändert oder komplett ersetzt werden unter der Voraussetzung, dass das Interface unverändert bleibt. Dies ist wichtig für die Portabilität des Clients.

Stateless Die Interaktion des Clients mit dem Server muss zustandslos erfolgen. Jeder Aufruf des Clients muss dazu die volle Information übertragen, die für den Server notwendig ist um die Anfrage korrekt zu beantworten. Es wird daher kein Kontext (zum Beispiel Sessiondaten) auf dem Server gespeichert. Diese werden ausschließlich auf dem Client verwaltet. Diese Anforderungen haben entscheidende Auswirkungen auf nicht funktionale Anforderungen wie Sichtbarkeit, Zuverlässigkeit und Skalierbarkeit.

Cache Zur Erhöhung der Netzwerkeffizients muss der Client einen Cache implementieren. Daten können explizit oder implizit für die Möglichkeit einer Zwischenspeicherung in einem Cache markiert werden. Sollten Daten zwischengespeichert werden so müssen ähnliche Requests aus diesem Cache bedient werden.

Uniform Interface Ein einheitliches Interface stellt den zentralen Punkt der REST Architektur dar. Das Interface eines Services wird so von der eigentlichen Implementierung abstrahiert. Dies führt zu einer deutlichen einfacheren Wiederverwendung in den anderen Systemen. Dies führt zu einem Trade-Off im Bereich Performance, da die übertragenen Daten in einer bestimmten standardisierten und nicht mehr in der systemspezifschen nativen Form übertragen werden können. REST eignet sich daher eher für grob-granulare Systeme zur Kommunikation.

Layered System Um den speziellen Anforderungen der Skalierbarkeit von Internetapplikatonen gerecht zu werden verwendet REST horizontale Layer die das System entsprechend seiner Funktionen aufteilen. Eine Eigenheit von layerbasierten Systemen ist es, dass jeweils nur der direkt darunter gelegene Layer bekannt ist und aufgerufen werden kann. Hierdurch erreicht man eine Reduzierung der Komplexität des Systems und eine Unabhängigkeit von anderen Layern. Ein Nachteil dieser Architektur ist es, dass es ein gewisser Overhead bei der Abarbeitung zu Stande kommt. Durch die Verwendung von Caches an den Layergrenzen können allerdings Performanceverbesserung erzielt werden.

Code on Demand Der Client soll durch downloadbare Scripts oder Applets erweitert werden können. Diese Anforderung ist optional.

RESTful Web Services bieten folgende Optionen zu den bereitgestellten Ressourcen an:

- GET: Anzeige der Ressource inklusive ihrer Eigenschaften und eventuellen Subelemente.
- POST: Erzeugt eine neue Entität.
- PUT: Ersetzt die Ressource durch eine neue übermittelte Entität.
- DELETE: Löscht eine bestimmte Ressource.

Der Aufbau zur Adressierung einer Ressource mittels REST Service folgt dem abgebildeten Schema in Abbildung 2.12. Das Beispiel demonstriert wie eine Ressource vom Typ 'person' und der eindeutigen ID 42 adressiert wird.

Als Protokoll wird HTTP definiert. Dies ist zwingend notwendig da dies eine essentielle Eigenschaft von REST basierten Services ist. Host und Port adressieren den Serviceendpunkt. Der

Abbildung 2.12: Aufbau einer URI zu einer REST-ressource

Pfad der Anwendung ist optional, daher könnte die Ressource auch direkt im Root Verzeichnis zur Verfügung gestellt werden. Zur besseren Strukturierung wird hier allerdings der Name der Anwendung bzw. des Services mitgegeben. Unsere Ressource ist vom Typ 'person', daher spiegelt sich dies auch in dem URI Schema wieder. Als letztes wird die eindeutige ID angegeben. Auf diese URI können nun die verschiedenen Aktionen wie GET, PUT, etc. durchgeführt werden um entsprechende Manipulationen der Daten durchzuführen.

Wie bereits erwähnt, benötigt die Übertragung eine einheitliche Repräsentation der Daten die vom oder zum Service gesendet werden. Ein oft im Zusammenhang verwendetes Format zum Datenaustausch ist JSON.

JSON

JavaScript Object Notation(JSON) ist ein textbasiertes und sprachunabhängiges Datenaustauschformat. Es definiert ein minimales Set an Regeln für die Repräsentation von strukturierten Daten. Es ist eine Abwandlung von der Objekt literaren aus Java Script und wurde im ECMAScript Programming Language Standard definiert. Das Ziel von JSON in der Entwicklung war es, so minimal wie möglich gehalten zu sein. Weitere wichtige Aspekte waren die Portabilität und die Fixierung auf eine textbasierte Notation. JSON kann vier primitive und zwei strukturierte Datentypen repräsentieren. Als einfache Datentypen werden String, Number, Boolean und Null verwendet, strukturierte Daten können als Objekte oder Arrays dargestellt werden. Ein Objekt wird hierbei verstanden als eine nicht sortierte Sammlung von null oder mehreren Paaren aus Namen und Wert, wobei der Name immer ein String ist. Der Wert kann die bereits erwähnten primitiven Datentypen annehmen bzw. ein Array oder ein Objekt sein. Ein Array in JSON ist eine geordnete Sammlung von Werten. [Cro06]

Beispiel 2.6 beinhaltet ein Objekt das mit JSON serialisiert wurde. Das Objet 'testcase' verfügt über die Attribute 'id' und 'name'. Des Weiteren besitzt es ein Array das mit 'entry' drei Objekten befühlt ist. Das 'entry' Objekt wiederrum verfügt über ein 'state' und ein 'comment' Attribut.

```
{"testcase": {
  "id": "file",
  "name": "First Testcase",
  "history": {
    "entry": [
      {"state": "added", "comment": "someComment"},
      {"state": "changed", "comment": "someComment"},
      {"state": "closed", "comment": "someComment"}
    ]
  }
}}
```

Listing 2.6: JSON Beispiel

Adobe Flex

Adobe Flex stellt ein Framework für die Entwicklung von Rich Internet Applikationen zur Verfügung. Das Framework umfasst verschiedenste Werkzeuge:

- Flex Builder: Der Flex Builder ist eine grafische Entwicklungsumgebung für das grafische Design der Oberflächen von Flex Anwendungen. Er basiert auf Eclipse
- Flex SDK (Software Development Kit): Das Flex SDK setzt sich aus den Flex Bibliotheken, dem Compiler sowie einem Debugger zusammen.
- BlazeDS: Zur Anbindung von Flex Anwendungen an unterschiedlichste Back-End Systeme wie Java wird das Framework BlazeDS verwendet. Es ermöglicht unter anderem die Anbindung an Java Applikation Server. Darüber hinaus bietet es allerdings noch weitere Funktionen zum Reporting oder zur Client Synchronisierung.
- Flex Charting: Flex Charting ist eine Bibliothek zur Erstellung von Charts.

Grafik 2.13 verdeutlicht den Unterschied einer Flex Anwendung zu einer HTML basierten Internet Applikation. Die Abbildung stellt nur ein Beispiel dar, daher die Technologien die im Bereich Server zur Anwendung kommen können auch durch andere ersetzt werden. Die Abbildung illustriert ein typisches Setup. An dieser Stelle sei darauf hingewiesen, dass auch andere Ausprägungen möglich sind. So kann die Adobe Flex Anwendung auch nur aus eingebetteten HTML Dateien bestehen.

Während HTML basierte Applikation im Browser direkt ausgeführt werden, wird für eine Adobe Flex Anwendung der Adobe Flash Player[Ado] benötigt. Die Flex Anwendung wird innerhalb dieses zusätzlichen PlugIns ausgeführt. Die Definition des grafischen Benutzer Interfaces (GUI) erfolgt bei Flex Anwendungen nicht in HTML und CSS, sondern mittels Macromedia eXtensible Markup Language (MXML). Dies ist eine von Macromedia 2004 (seit 2005 in Adobe System eingegliedert) eingeführte Markup Sprache. Beispiel 2.7 illustriert wie die Darstellung von Elementen mittels MXML funktioniert. Die darzustellenden Elemente werden innerhalb des Applikation Root Elements angegeben. Dieses Root Element kann zusätzliche Attribute besitzen

Abbildung 2.13: Vergleich HTML basierte Web Applikation zu Adobe Flex Applikation

wie Verweise auf aufzurufende Methoden bei bestimmten Ereignissen. In diesem Beispiel wird die init Methode nach der vollständigen Erstellung der Anwendung im Client aufgerufen.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:mx="library://ns.adobe.com/flex/mx" minWidth="955" minHeight="
                   600" creationComplete="init()" >
                  <mx:Label text="MXML Beispiel für die Darstellung eines Labels!"
                      verticalCenter="0" horizontalCenter="0" fontSize="50">
    </mx:Label>
</mx:Application>
```

Listing 2.7: MXML Beispiel

Ein weiterer Unterschied zu HTML basierten Anwendungen ist, dass in Adobe Flex Action Script, und nicht wie bei HTML Java Script, zur Manipulation von Objekten verwendet wird. Abbildung 2.8 zeigt eine kurze Action Script Methode. Die Methode bearbeitet ein erzeugtes MouseEvent, jedes Mal wenn der Cursor bewegt wird aktualisiert diese Methode die Koordinaten des zu bewegenden Objekts. Danach wird die Ansicht im Adobe Flash Player aktualisiert. Dies verdeutlicht, dass sich mittels Adobe Flex und Action Script relativ leicht Funktionen wie Drag and Drop umsetzen lassen.

```
<fx:Script>
    function dragObject(event:MouseEvent):void
      {
        moving.x = event.stageX - offsetX;
        moving.y = event.stageY - offsetY;
        event.updateAfterEvent();
      }
</fx:Script>
```

Listing 2.8: ActionScript Beispiel

Die Erstellung einer Adobe Flex Anwendung ist in Abbildung 2.14 schematisch dargestellt. Mittels Flex Builder werden zuerst alle notwendigen Artefakte erstellt. Die daraus resultierenden Dateien, unter anderem MXML und ActionScript Dateien, werden zu einer einzigen SWF Datei kompiliert. Diese wird dann im Browser ausgeführt.[Mül09]

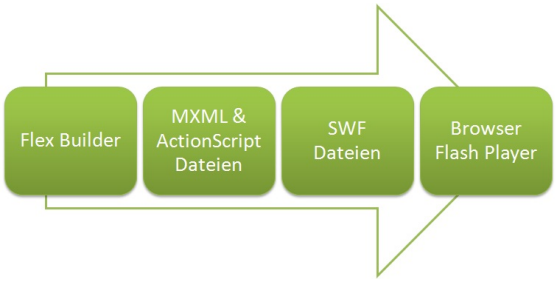

Abbildung 2.14: Erstellung einer Adobe Flex Anwendung

Flex und Flash unterscheiden sich hinsichtlich ihrer Ausrichtung. Während Flash in seiner Konzeption hauptsächlich auf Animationen und grafische Elemente abgezielt hat, wurde Flex mit dem Ziel entworfen in Enterprise-Applikationen zum Einsatz zukommen.

BlazeDS Server

Rich Client Internet Lösungen benötigen meist eine Anbindungen an einen Backend Server, die diverse Aufgaben wie die Bereitstellung der Daten oder Abarbeitung diverser Methodenaufrufe übernimmt. Adobe stellt hierfür die Adobe LiveCycle Data Services (LCDS) zur Verfügung. Die Hauptaufgabe ist die serverseitige Integration der Geschäftslogik in die Applikation. Ab einer gewissen Abstraktionsebene ist LCDS eine einfache J2EE Web Applikation, daher LCDS werden in Java Applikation Servern deployed und stellen so einen Anbindungspunkt für Flex Anwendungen bereit. Da die Implementierung von LCDS proprietär ist wurde 2008 eine Open Source Variante bereitgestellt die unter der Open Source Projekt(LGPL- Lizenz) namens BlazeDS veröffentlicht wurde. Sie umfasst einen Großteil der Funktionen von LCDS. Abbildung 2.15 stellt die einzelnen Komponenten innerhalb von BlazeDS da.

RPC (Remote Procedure Call) Services stellen eine Möglichkeit zum asynchronen Datenaustausch zwischen Client und Server bereit. Einfache Services können zum Beispiel mittels HTTP Protokoll und GET und POST aufrufen realisiert werden. Eine andere Möglichkeit ist der direkte

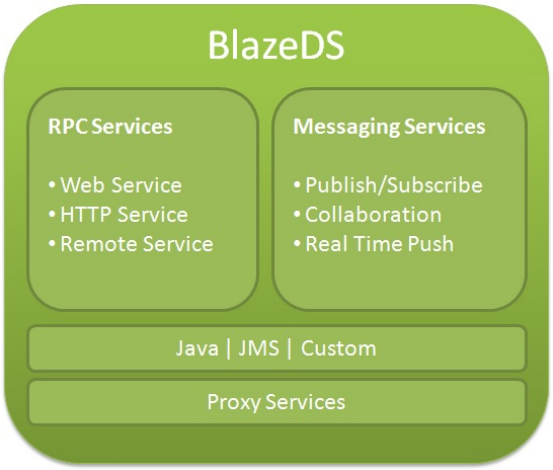

Abbildung 2.15: BlazeDS Komponenten

Aufruf von Java Klassenmethoden mittels der Verwendung von Remote Services. Messaging Services bieten die Möglichkeit die Flex Anwendung als Listener zu konfigurieren. Die Kommunikation läuft nach dem klassischen Message Producer und Message Consumer Prinzip ab. Der Producer erstellt Nachrichten die an den Consumer weitergeleitet werden. Hier kann sowohl die Serverseite als auch die Clientseite beide Rollen einnehmen. Der Einsatz von Standards wie Java Messaging Services(JMS)[Oraa] ist ebenfalls möglich. Mittels Service Adapter können beliebige Schnittstellen angesprochen werden. Dies ermöglicht die Anbindung auch an andere Systeme als Java bzw. JMS. All diese Schichten verwenden das Proxy Service. Dies dient als Vermittlungsschicht zwischen der Datenquelle und dem Client. Das Proxy Service kapselt alle notwendigen Funktionen und verwaltet die Verbindungen vom Client zum Server. Dadurch können Anforderungen an Sicherheit oder Verbindungsmanagement an einer zentralen Stelle verwaltet werden.

Android

Durch das Aufkommen von immer leistungsfähigerer Hardware für mobile Geräte, werden diese zunehmend stärker in Geschäftsprozesse eingebunden. Der Kunde kann dadurch auch von Unterwegs an diversen Prozessen partizipieren. Abbildung 2.16 zeigt die vergangene und geschätzte zukünftige Entwicklung der Branche der mobilen Applikationen[boo10].

Durch die Einbindung von mobilen Applikationen werden verteilte Systeme weiter an Bedeutung gewinnen. Google stellt mit seinem Betriebssystem Android[Goob] eine mobile Plattform zur Verfügung die vor allem im letzten Jahr rasch Marktanteile für sich gewinnen konnte. Abbildung 2.17 illustriert die Architektur von Android in der aktuellen Version 2.2.

An oberste Ebene befindet sich der Application Layer. Dieser beinhaltet Anwendungen die unter Verwendung der darunter liegenden Ebenen ausgeführt werden. Das Application Framework stellt verschiedenste Manager zur Verfügung die gewisse Funktionsbereiche kapseln und so die Entwicklung von Programmen vereinfachen. Des Weiteren stehen Libraries wie die

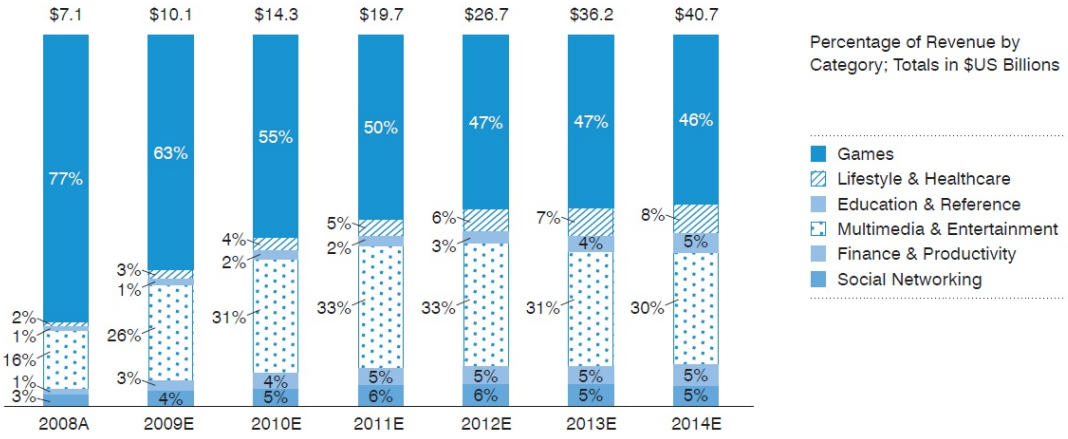

Abbildung 2.16: Globaler Markt für mobile Applikationen[boo10]

SSL Library bereit um Entwicklungsaufgaben zu beschleunigen. Die Android Runtime beinhaltet Bibliotheken die einen Großteil der Standardfunktionalität von Java bereitstellen. Auch die virtuelle Maschine (VM) ist hier angesiedelt. Jede Applikation läuft in ihrem eigenen Prozess in einer eigenen Instanz der Dalvik VM. Diese benutzt wiederrum einen Linux Kernel der Version 2.6. Dieser wird verwendet um die Basisfunktionalität wie Sicherheit, Speichermanagement oder Prozessmanagement bereit zu stellen.

Anwendungen auf der Androidplattform können in Java erstellt werden und können diverse bereitgestellt APIs benutzen. Der Java Code wird wird vom Kompilierer in nativen Code kompiliert.

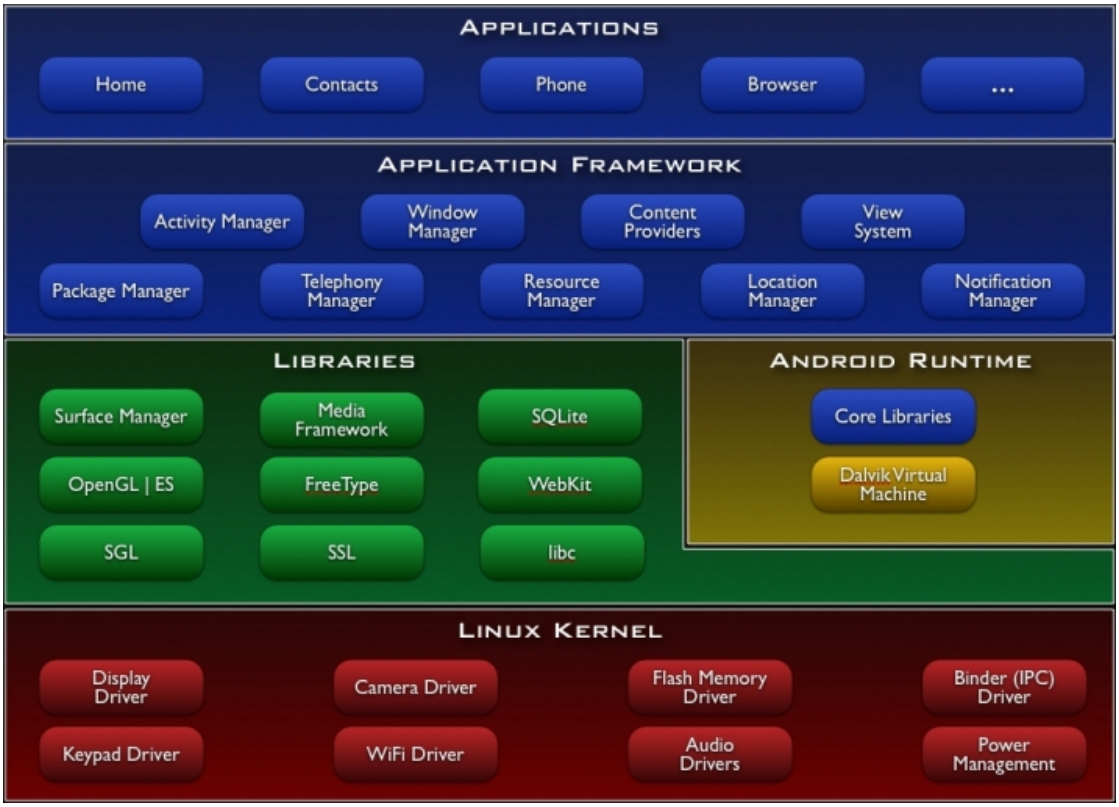

Abbildung 2.17: Android Architektur[Good]

KAPITEL 3

Related Work

Es existieren unterschiedlichste Ansätze zum Testen von verteilten Systemen. Sowohl in technischer als auch in organisatorischer Hinsicht bietet dieses Gebiet einige Herausforderungen zu denen bereits aktiv Forschungsarbeit betrieben wurde. Dieses Kapitel stellt einige dieser Ansätze näher vor und vergleicht sie mit dem in dieser Arbeit entwickeltem Framework.

In [JfSJb+06] wird die Problematik der steigenden Komplexität von verteilten Systemen adressiert. Diese resultiert vor allem aus den immer größeren Dimensionen der heterogenen Netzwerke in denen diese laufen. Der hier vorgestellte Ansatz des Reflective Architecture Based Software Testing Management Model (RATMM) verbindet die Reflektion Theorie mit Software Architektur Design Methoden und Softwaretestmanagement. Dies wird vor allem durch das Anreichern der Umgebung mit Metainformationen erreicht. Diese beschreibt zum Beispiel die Struktur, Sematnik oder den Grund einer gewissen Information. Dadurch soll der gesamte Softwaretestprozess vereinfacht werden. Der Hauptfokus liegt hier bei auf UNIT und Komponententests. Die Grundannahme hierbei ist, dass die Phase des Testens nicht separat gesehen werden kann wie sie zum Beispiel im V Model stattfindet. Die verschiedenen Phasen, vor allem das Testen, können nicht rein sequenziell abgearbeitet werden. Testen wird als ständig begleitender Prozess gesehen. Die reflektive Architektur teilt sich hierzu in ein Meta-Level und ein Basis-Level. Während die Meta-Ebene hält Metadaten um die Basis-Ebene zu kontrollieren, daher um ein sinnvolles Monitoring über diese zu gewährleisten. Meta-Objekte können hier zu einzelne Basis-Objekte überwachen. Im Vergleich zu HEKATE ist die vorgestellte Methode nur ein Model. HEKATE hingegen unterstützt als Framework aktiv Tester und zielt vor allem auf die Einfachheit der Tests ab um auch reinen Fachtestern Testabläufe transparent zu gestalten. Der grundlegende Ansatz ist allerdings ähnlich, beide Konzepte sehen die Notwendigkeit eines guten Monitorings um die Tests von verteilten Systemen entsprechend unterstützen zu können.

[BGS01] beschäftigt sich mit der grafischen Repräsentation von Black- bzw. White-Box Tests. Der Hauptfokus liegt hierbei auf der Integration beider Arten. Beide Testarten haben das gleiche Ziel, das Aufspüren von Fehlern im Programm. Dennoch werden Black- und White-Box Test meistens unabhängig voneinander erstellt und ausgeführt. Das hier vorgestellte Konzept soll diesen Umstand durch entsprechende Techniken verbessern und die Erstellung integriert ermöglichen. Dies soll durch eine neue grafische Repräsentation der Tests erreicht werden. Diese unterscheidet für jede Klasse zwei unterschiedliche Sichten, die Spezifikation und die Implementations-Sicht. Beide werden als Control-Flow Graphen dargestellt. Dies ermöglicht das Testen mittels strukturbasierten Techniken. Durch den Ansatz der Zusammenführung der beiden Testarten muss der Tester sich nur mehr mit einem einzigen Tool und einem einzigen Konzept vertraut machen. Des Weiteren ist die Wartung geringer da sie nur mehr einmal an einer zentralen Stelle durchgeführt werden muss. Diese hier vorgestellte Methode ist nicht für verteilte Systeme gedacht und konzipiert, da sie nur auf 'lokale' Software abzielt. Dennoch ist das Konzept der Zusammenlegung der Testarten ähnliche wie bei HEKATE. Unser Framework versucht ebenfalls die Komplexität der Tests zu reduzieren und einen zentralen Anlaufpunkt für alle Tests zu schaffen. Durch unterschiedliche Validierungspunkte können je nach Detailgrad Stufen zwischen reinen Komponententests oder WhiteBox-Tests abgedeckt werden.

[BP09] wendet sich dem speziellen Thema von verteilten Umgebungen zu die mittels SOA Referenz Architektur implementiert sind. Der Hauptfokus dieser Arbeit liegt auf SOA Test Governance bzw. Integrationstests über organisatorische und technologische Grenzen hinweg.

Es wird der Begriff der SOA Test Governance (STG) eingeführt. Software Testen benötigt zwei grundsätzliche Aspekte - die Kontrolle und die Beobachtung. Die Kontrolle beinhaltet, dass der Tester das System dem Test entsprechend manipulieren kann und so die entsprechenden Tests ausführen kann. Die Beobachtung wird in Hinsicht auf eingehende Anfragen von anderen Systemen gesehen - wie sich das System bei den entsprechenden Anfragen verhält und welche Rückgaben geliefert werden. Bei traditionellen Softwaretests verfügt der Tester über direkten Zugang zum System und kann sämtliche Teile beobachten und manipulieren. Bei SOA Umgebungen ist dies nicht mehr gegeben, die anderen Systeme könnten unter anderer organisatorischer Kontrolle stehen, nicht beobachtbar oder zur Zeit des Tests nicht verfügbar sein. SOA Test Governance versucht nun ein Konzept aufzuzeigen, dass Tests in diesen Umgebungen kollaborativ durchgeführt werden müssen. Alle Stakeholder entlang eines Prozesses müssen miteingebunden werden. STG legt hierfür verschieden Standards, Policies, und Regeln fest um dies zu ermöglich. HEKATE versucht dies ebenso zu erreichen, nur fokussieren wir unsere Arbeit auf die technische Unterstützung durch ein Framework während die hier vorgestellte Arbeit nur organisatorische Aspekte abdeckt.

[Lee09a][Lee09b] stellt ein Testframework für SOA Umgebungen vor das auf BPA(Business Process Activity)-Simulated Event Proxys basiert. Der Ansatz trennt SOA Systeme in folgende drei Ebenen: Business Process Layer, Service Layer und Computing Resource Layer. Der Fokus der Tests liegt hier vor allem auf dem Business Process Layer, da ein Test dieser Ebene einen Test der darunter liegenden Ebenen weitgehend mit einschließt. Um einen Test durchzuführen werden bestimmte Aktivitäten simuliert. So kann ein realer Geschäftsprozess in einem Test Szenario nachgespielt werden. HEKATE wird keine Generierung von Proxyservices etc. unterstützen. Das Ziel, im Gegensatz zu dem hier vorgestellten Konzept ist es, bereits fertige Umgebungen zu testen um Integrationstests durchzuführen. Der Ansatz auf der Business Prozess Ebene zu testen ist allerdings derselbe, auch HEKATE geht davon aus, dass wenn die Businesslogik auf oberste Ebene korrekt funktioniert die darunter liegenden Ebenen ebenfalls funktionieren.

In [LLZ+09] wird ein Framework für kontinuierliche Integrationstests von SOA Umgebungen vorgestellt. Continuous Integration Testing (CIT) wird hier als zentraler Bestandteil der Qualitätssicherung von SOA Umgebungen gesehen. Bei der Anwendung von CIT wird sofort nach der Entwicklung von Code ein Integrationstest durchgeführt. Das Framework teilt sich in zwei Ebenen. Der Behaviour Layer repräsentiert die Testlogik eines Testfalles und ist plattformunabhängig. Der Configuration Layer hingegen enthält Plattform spezifische Informationen und ist für verschiedene Programmiermodelle konfigurierbar. In SOA Umgebungen sind unterschiedliche Teile oft durch in unterschiedlichen Programmiermodellen wie EJB oder Web Services gelöst. Jedes dieses Modelle hat andere Mechanismen für interne und externe Aufrufe. Durch die Trennung in die zwei genannten Ebenen kann dieses Problem gelöst werden. Dies ist auch eine Ähnlichkeit zu HEKATE. Unser Framework kann an beliebigen Stellen im Prozess Daten einfügen umso bestimmte Features zu testen. Auch hier ist die Unabhängigkeit zu dem darunterliegenden Programmiermodel gegeben.

[ZQ06] beschäftigt sich mit der Generierung von negativen Testfällen für Web Serivces. Negative Testfälle sind Testfälle die absichtlich Fehler produzieren. Als Grundannahme wird hier von einer SOA Umgebung ausgegangen, in der Web Services mittels Universal Description, Discovery, and Integration (UDDI) gefunden und verwendet werden können. Des Weiteren

stehen einige Web Services mit beinahe der gleichen Funktionalität zur Verfügung zwischen denen entschieden werden muss. Der Ansatz ist nun, im Gegensatz zu vorherigen, nicht die Web Services zu validieren, sondern diejenigen die nicht spezifikationsgemäß funktionieren auszuschließen und nicht zu verwenden. Mittels Grenzwerten bei automatisch erstellen Testfällen wird so rasch die Richtigkeit eines Services überprüft. An den geeigneten Stellen innerhalb eines Prozesses werden dann diese generierten Daten injiziert. Im Vergleich zu HEKATE ist dies ein ganz ähnlicher Ansatz. Auch HEKATE wird an bestimmten Stellen Daten in einen Prozess einschleusen können um gewisse Subsysteme zu validieren. Allerdings generiert HEKATE nicht automatisch Daten, sondern diese werden vom Fachtester manuell eingegeben. Des Weiteren wird bei HEKATE davon ausgegangen, dass die benutzten Services bereits feststehen.

[ZfXp09] widmet sich dem Test von mobilen Endgeräten wobei es eine SOA Architektur nutzt um die Tests über multiple Plattformen hinweg zu unterstützen. Das Testen von mobilen Geräten hat verschiedene spezifische Charakteristika. Debugging und Testen ist auf den meisten Endgeräten nur eingeschränkt möglich. Anwendungen deren Funktionalität rund um Kommunikation angesiedelt sind haben gewisse Real-Time Anforderungen. Verschiedenste Anbindungen an das Internet (WLAN, Bluetooth, GSM, etc.) erhöhen des Weiteren die Komplexität der Tests. Im Gegensatz zu Desktopanwendungen gilt es bei mobilen Applikationen auch den Aspekt des Energieverbrauchs zu beachten. Es wird ähnlich wie bei vorigen Ansätzen eine Abstraktionsschicht zwischen den Testfällen und deren konkreter Ausführung eingeführt. HEKATE, wird nicht nur in Hinblick auf unterschiedliche mobile Geräte, auf Businessprozessebene das Testen auch auf unterschiedlichsten Plattformen unterstützen, da es durch die Bereitstellung einer Standardschnittstelle Monitoring auf den unterschiedlichsten Systemen zu lässt.

[BDS10] beschreibt eine erweiterbare Architektur für das Monitoring von Runtime Umgebungen die auf Web Services basieren. Web Services unterliegen ständigem Wandel. Durch die Einbindung von externen Web Services kann es zu unbemerkten Änderungen kommen die das Verhalten der eigenen Services unbemerkt beeinflussen. Web Services, die zusätzlich Zustandsinformationen speichern, sind eine zusätzliche Herausforderung bei der Planung einer gut skalierbaren Architektur für das Monitoring und die Validierung. Diese erfolgt durch die Ausführung verschiedener Sequencen von Testfällen, basierend auf realistischen Beispieldaten. HEKATE verwendet ebenfalls Beispielszenarien mit realistischen Daten für die Validierung von Systemen. Daher auch Änderungen an externen Systemen können so durch die Ausführung von Tests aufgedeckt werden.

In [KT04] wird ein Konzept behandelt, dass den REST Architekturstil für dezentralisierte Systeme erweitert. Das Hauptproblem, das hierbei adressiert wird die schnelle Änderung der Ressourcen in diesen Systemen. Ressourcen die sich häufig ändern bzw. schneller ändern als man deren Änderungen zu anderen Systemen übertragen kann stellen ein Problem in diesem Bereich dar. Diese Limitierung kann zum Beispiel durch physische Gegebenheiten wie die Netzwerklatenz gegeben sein. Als Lösungsansatz wird hier die Möglichkeit diskutiert, die einzelnen beteiligten Systeme keine eigenen Entscheidungen mehr treffen zu lassen. Einzelne Komponenten werden per Konfiguration bezüglich ihrer Handlungsfähigkeit auf unterschiedlichen Objekten eingeschränkt. Es wird ein eventbasierter Ansatz verwendet um Änderungen an einem Objekt so schnell wie möglich mittels REST Schnittstelle weiterzuleiten. HEKATE wird ebenfalls per REST Schnittstelle Änderungen an Komponenten verfolgen können. Sensoren werden

per REST Schnittstelle Zustände an das Hauptservice melden. Im Unterschied zu dem hier vorgestellten Ansatz ist dies allerdings bei HEKATE nicht zeitkritisch.

[LLB+09] beschäftigt sich mit dem Management von lose gekoppelten Systemen mittels REST. Software-as-a-Service weißt spezielle Faktoren auf. So sind verschiedene Anwendungen auf verschiedenen Knoten installiert die sich in unterschiedlichen organisatorischen Bereichen befinden. Die meisten Service Managementansätze gehen von einem zentralen Verwaltungsansatz aus. Die CMDB (centrally managed configuration management database) wird als Basis für die Orchestrierung der einzelnen Komponenten verwendet. Durch die Verteilung von Services über das Internet ist es allerdings notwendig geworden einen dezentralisierten Ansatz zur Steuerung zu finden. In diesem Paper wird hierfür ein REST Ansatz für die Konfiguration und ein ATOM-based Ansatz für Updates vorgestellt. Auch HEKATE widmet sich dem Thema unterschiedlicher organisatorischer Aspekte. Der hier vorgestellt Ansatz ist dem Steuerungsansatz von HEKATE insofern ähnlich, da beide für verteilte Services Steuerungsmöglichkeiten von außen vorsehen.

[AS08] behandelt einen Ansatz zur Umsetzung von datenintensiven REST basierten Softwarelösungen. Das Paper beschreibt die Umsetzung einer Use Case Studie mit einem MashUps unter der Verwendung von Yahoo! Maps, Amazon Associates Web Service, REST und dem Flex Framework. Basierend auf den Erfahrung der Erstellung und der Analyse der Daten gibt es Best Practice Tipps zur Umsetzung von MashUps. Die Visualisierung der Daten sollte immer als erstes erstellt werden. Das komplexe Design der Darstellung lässt sich leicht in sinnvolle Datenkomponenten herunterbrechen. Das Datenbankmodel sollte nur wirklich benötigte Daten beinhalten. Sollten von den einzelnen Teilnehmern des MashUps nicht benötigte Daten übertragen werden, so ist es sinnvoller diese zu verwerfen als zu speichern. Es ist wichtiger eine gute Datenqualität zu besitzen als eine hohe Quantität. Das generelle Design sollte darauf abzielen, die Latenz zwischen Benutzerinteraktionen und Serverantworten zu minimieren. Es sollten für verschiedene Benutzer unterschiedliche Darstellungen der Daten möglich sein. Die Interaktion mit den Daten sollte auf den jeweiligen Benutzer zugeschnitten sein und deren Fertigkeiten und Wissen miteinbeziehen. HEKATE selbst, kann auch als eine Art MashUp, betrachtet werden. Die hier gewonnenen Erfahrungen fließen bei der Planung und Erstellung von HEKATE ein.

[AGM08] stellt einen Ansatz zur Darstellung von MashUps vor. Als Basis dient das Mashlet. Dieses stellt die Möglichkeit zur Verfügung verschiedenste Datenquellen abzufragen, externe Web Services zu verwenden oder komplexe Interaktionsmuster zu implementieren. Der Zustand eines Mashlets ist durch die Verbindung zu anderen Mashlets und die interne Logik definiert. Die Interaktion mit dem User und anderen Anwendungen wird als Verlauf von Änderungen modelliert. Durch das eingeführte Datenmodel bzw. die damit verbundene deklarative Sprache für MashUp Spezifikationen soll ein Versuch zur Standardisierung von MashUps erfolgen. Der Ansatz zur Verwendung von Mashlets entspricht im weitesten Sinne den Sensoren in HEKATE. Beide liefern Daten unabhängig von der darunter liegenden Implementation.

KAPITEL 4

HEKATE - Heterogenes Environment Knowledge and Test Enhancer

4.1 Einführung & Konzept: Testfokussierung

Dieser Abschnitt stellt HEKATE näher vor, ein Framework zur Unterstützung von Tests in verteilten Systemen. HEKATE sieht sich nicht nur als Softwareunterstützung, sondern auch als generelles Paradigma zum Testdesign von verteilten Anwendungen. HEKATE verfolgt in diesem Bereich das Konzept des Test-Driven Designs. Tests schon bei der Planung von Softwaresystemen zu berücksichtigen führt später zu einer deutlich erhöhten Testbarkeit des Gesamtsystems. Durch die entsprechenden Anpassungen können Komponenten und Systeme entsprechend einfacher und schneller getestet werden. Systeme die bei ihrer Planung keinerlei Rücksicht auf die Testbarkeit von Funktionen legen, erschweren sich den Einsatz von Methoden wie Unit Tests oder Komponententests. Dies führt zu einer schlechteren Softwarequalität und einer erschwerten Wartung. Integrationstests, die über mehrere Anwendungen hinweg durchgeführt werden, müssen hier noch einen Schritt weitergehen. Es können keine einfachen Unit Tests mehr zum Einsatz kommen, bei denen Teile gemockt werden. Das gesamte System muss von einer weiteren Instanz überwacht und auf Korrektheit hin überprüft werden.

HEKATE teilt sich hierfür in unterschiedliche Komponenten. Das Frontend stellt das wichtigste Element der Interaktion mit den Benutzern dar. Es ermöglicht die Erstellung von Systemen, Testfällen und deren Ausführung. Auch das Monitoring und Reporting wird hier für den User aufbereitet angezeigt.

Das HEKATE Backend übernimmt die Verwaltung der gesamten Daten und stellt Services für das Frontend und die Sensoren bereit.

Die Sensoren sind ein von HEKATE eingeführtes Konzept. Sie dienen zur Sammlung von Daten und Informationen von den zu überwachenden Systemen. Sie werden an den entsprechenden Punkten eingebunden und liefern diverse Werte an das HEKATE Backend weiter. Sensoren dienen darüber hinaus auch zur Feststellung und Sammlung von Monitoring Daten.

Dieses Kapitel gibt einen Überblick über Anforderungen von HEKATE und die daraus resultierende Architektur. Das Kapitel stellt die unterschiedlichen Komponenten von HEKATE (Frontend, Backend und Server) genauer vor.

Die für eine grundlegende Unterstützung in diesem Bereich notwendigen Use Cases werden im folgenden Abschnitt näher beschrieben.

4.2 Use Cases

Der folgende Abschnitt beschreibt die wichtigsten Use Cases die identifiziert wurden. Abbildung 4.2 gibt mittels eines Use Case Diagrammes einen Überblick der Use Cases die durch HEKATE unterstützt werden. Des Weiteren wurden wie im Diagramm ersichtlich fünf wesentliche Rollen identifiziert die das Framework verwenden. Diese werden nun genauer beschrieben:

Fachtester Die wohl wichtigste Rolle aus Sicht von HEKATE stellt die Rolle des Fachtesters dar, denn die primäre Aufgabe des Framework ist es, diese Rolle in ihrer Tätigkeit soweit wie möglich zu unterstützen. Bei einem typischen Fachtester wird davon ausgegangen, dass dieser über nur sehr geringe Kenntnisse im Bereich der Implementation bzw. generell über die Systemarchitektur verfügt. Für ihn stellt sich das System bisher als BackBox da in das

er keinerlei Einsicht hatte. Seine Aufgabe ist es, ein System auf seine fachliche Richtigkeit zu überprüfen die anhand der Spezifikation festgelegt worden ist. Hierfür verwendet er Testfälle die ebenfalls von Fachtestern erstellt worden sind. Die Use Cases die dieser Rolle in HEKATE durchführen kann sind: Testfall erstellen & bearbeiten, Testfall ausführen, Testfallstatus tracken und System Monitoring.

Programmierer Unter Programmierer werden hier alle technisch versierten Beteiligten zusammengefasst, das gilt für Entwickler ebenso wie für System Architekten. Eine genaue Differenzierung ist an dieser Stelle nicht notwendig. Die Hauptaufgaben sind es eine Systemübersicht mittels HEKATE zu modellieren und anschließend in den externen Systemen entsprechende Sensoren zu setzen. Diese Daten können dann mittels Validierungspunkten ausgewertet werden.

Testmanager Der Testmanager, als Verantwortlicher über den gesamten Test, muss entsprechend auch über die Fortschritte der Integrationstests Bescheid wissen. HEKATE liefert ihm hierbei Statistiken bzw. Reports über die aktuelle Situation der Testfällestatus.

externes System Externe Systeme können in ihrer konkreten Ausprägung eine Vielzahl an Formen annehmen, von mobilen Endgeräten bis hin zu einer Komponente die in einem Applikationserver ausgeführt wird. Diese Systeme verfügen über Sensoren die an HEKATE Daten über aktuelle Ereignisse liefern wie die Verarbeitung von neuen Schnittstellendaten etc. Diese Daten werden an HEKATE geliefert und gespeichert um für spätere Reports zu dienen bzw. für das Life Monitoring aufbereitet.

Systemverantwortlicher Die Rolle des Systemverantwortlichens kann in der Praxis auch vielfältig ausgelegt werden. Wichtig ist nur, dass der Systemverantwortliche für seine Komponente über die jeweilige Kompetenz verfügt Informationen rasch zu beschaffen, bzw. eventuelle Fehler ausbessern zu können.

Im Weiteren sollen nun die Use Cases genau beschrieben werden wofür folgendes vereinfachtes Template verwendet wird:

Annahmen Beschreibt Bedingungen die vor dem Ausführen des Use Cases erfüllt werden müssen, damit dieser erfolgreich durchgeführt werden kann.

Rolle Rolle die diesen Use Case ausführt

Beschreibung Eine Beschreibung der Funktionalität des Use Cases. An dieser Stelle soll darauf hingewiesen werden, dass die Darstellung der Use Cases nicht mittels funktionaler Sprachen oder ähnlichen exakt spezifiziert wird. Das Ziel ist es eher dem Leser einen groben Überblick über die Funktionalität des Frameworks zu verschaffen.

Use Case: Testfall erstellen & bearbeiten

Annahmen Das System wurde von einem Programmierer modelliert. Validierungspunkte und Sensoren wurden erstellt und installiert.

Rolle Fachtester

Der User kann Testfälle erstellen, bearbeiten und löschen. Diese Testfälle beinhalten folgende Informationen:

1. Titel: Ein kurzer Titel, der treffend den Testfall beschreiben soll
2. Ziel: Das exakte Ziel, was mit dem Testfall getestet werden soll
3. Vorbedingungen: Vorbedingungen, die erfüllt sein müssen, damit der Testfall durchgeführt werden kann.
4. Beschreibung: Eine exakte Beschreibung der Durchführung
5. Validierungspunkte: Jeder Testfall wird zu einem bestimmten System eingetragen. Die Validierungspunkte können im Einzelnen (z.B.: bestimmte Werte) je nach Testfall adaptiert werden.

Beschreibung: Jeder Testfall erhält des Weiteren einen Status. Folgendes Zustandsdiagramm listet die möglichen Status und deren Übergänge auf:

Wie in der Grafik ersichtlich wird auf komplizierte Szenarien, in denen zum Beispiel Testfälle erst abgenommen werden etc. verzichtet. Das hier vorgestellte Szenario reicht aus um die Funktionalität von HEKATE zu demonstrieren.

Use Case: Testfall ausführen

Annahmen Das System wurde von einem Programmierer modelliert. Validierungspunkte und Sensoren wurden erstellt und installiert. Der Testfall wurde zuvor von einem Fachtester erstellt.

Rolle Fachtester

Beschreibung: Der Fachtester wählt einen gewünschten Testfall aus. Diesen Testfall kann er auf einer Instanz des modellierten Systems ausführen. Er bekommt dafür eine grafische Ansicht des modellierten Systems mit den verschiedenen Knotenpunkten. Diese Ansicht gibt ihm Feedback über den aktuellen Testverlauf sowie den Status der einzelnen Knoten. Schlägt ein Validierungspunkt in einem Knoten fehl, so kann sich der Fachtester hierzu die Details anzeigen lassen.

Use Case: System Monitoring

Annahmen Das System wurde von einem Programmierer modelliert. Validierungspunkte und Sensoren wurden erstellt und installiert. Der Testfall wurde zuvor von einem Fachtester erstellt.

Rolle Fachtester, Systemverantwortlicher

Beschreibung: Der Benutzer soll die Möglichkeit haben Daten der verschiedenen Systeme einzusehen. Hierzu sollen ihm die gesammelten Daten aus den Testfällen bzw. Livedaten angezeigt werden. Unter Livedaten werden Daten verstanden die in der Produktion gesammelt werden. Als Beispiel für Monitoring Daten wird die Durchlaufzeit definiert.

Use Case: Testfall Status tracken

Annahmen Das System wurde von einem Programmierer modelliert. Validierungspunkte und Sensoren wurden erstellt und installiert. Der Testfall wurde zuvor von einem Fachtester erstellt. Der Testfall wird oder wurde durchgeführt.

Rolle Fachtester

Beschreibung: Der Tester soll die Möglichkeit haben den Status eines Testfalles einzusehen. Daher Testfälle die unter Umständen längere Durchlaufzeiten haben, sollen die Möglichkeit bieten den aktuellen Fortschritt anzuzeigen.

Use Case: Reporting über Testfälle

Annahmen Das System wurde von einem Programmierer modelliert. Validierungspunkte und Sensoren wurden erstellt und installiert. Der Testfall wurde zuvor von einem Fachtester erstellt. Testfälle wurden durchgeführt.

Rolle Testmanager

Beschreibung: Der Testmanager soll die Möglichkeit haben, Statistiken von System abzufragen. Die Testfälle sollen entsprechened ihrem Status gegliedert dargestellt werden.

Use Case: Systemstatus melden

Annahmen Das System wurde von einem Programmierer modelliert. Validierungspunkte und Sensoren wurden erstellt und installiert. Der Testfall wurde zuvor von einem Fachtester erstellt.

Rolle externes System

Beschreibung: Das externe System muss die Möglichkeit haben mittels Sensoren dem Testserver verschiedene Statusmeldungen zu übermitteln. Folgende Beispielstatusmeldungen werden definiert:

1. Sensor passiert, die Daten werden übermittelt
2. Durchlaufzeit für einen Durchlauf

Use Case: Systemübersicht erstellen

Annahmen Validierungspunkte und Sensoren wurden erstellt und installiert.

Rolle Programmierer

Beschreibung: Der Programmierer kann eine grafische Systemübersicht erstellt. Dazu kann er Knoten anlegen, diese mit Validierungspunkten auf physischen Systemen verbinden, und die Knoten untereinander verbinden um so einen Datenfluss zu visualisieren.

Use Case: Validierungspunkte erstellen & bearbeiten

Annahmen Das System wurde von einem Programmierer modelliert.

Rolle Programmierer, Systemverantwortlicher

Beschreibung: Der Programmierer soll zu jedem Knoten verschiedene Validierungspunkte anlegen und diese Bearbeiten können. Jeder Validierungspunkt wird mit einem Sensor gekoppelt.

Use Case: organisatorische Informationen verwalten

Annahmen Das System wurde von einem Programmierer modelliert. Validierungspunkte und Sensoren wurden erstellt und installiert.

Rolle Testmanager, Systemverantwortlicher

Beschreibung: Der Testmanager kann zu jedem Knoten und Sensor einen oder mehrere verantwortliche Personen hinterlegen. Zu diesen können mehrere Kontaktinformationen gespeichert werden.

4.3 Anforderungen

Funktionale Anforderungen

Der folgende Abschnitt beschreibt die genauen funktionalen Anforderungen an das Framework In Abbildung 4.3 wird ein kurzer Überblick gegeben welcher Layer welche Anforderungen auszugsweise erfüllt.

Nicht-funktionale Anforderungen

Aus der speziellen Verwendung von HEKATE durch nicht technisch versierte Benutzer ergeben sich eine Reihe nicht-funktionaler Anforderungen.

Bedienbarkeit HEKATE muss einfach zu bedienen sein. Die Usability muss entsprechend den Benutzern angepasst sein. Für den Fachtester dürfen keinerlei Konfigurations- oder Installationsschritte notwendig sein um HEKATE verwenden zu können.

Portabilität und Wartung HEKATE kann durch unterschiedliche Personen in unterschiedlichen Unternehmen in demselben Projekt verwendet werden. Dadurch muss HEKATE auf unterschiedlichsten Systemen lauffähig sein. Neue Versionen von HEKATE sollen nicht für jeden Client einzeln installiert werden müssen.

Backup Das Backup aller Daten muss an einer zentralen Stelle erfolgen um die Benutzer nicht damit zu belasten.

Verfügbarkeit Die Verfügbarkeit des Backends muss auf einem hohen Maße gewährleistet sein.

Skalierbarkeit Sollten mehrere Systeme damit überwacht werden, muss eine entsprechende Skalierbarkeit von HEKATE gegeben sein.

Diese Anforderungen werden durch die Verwendung eines Thin-Clients sowie die Verwendung von REST Schnittstellen und der damit verbundenen Skalierbarkeit von Web Servern abgedeckt. Im Bereich der Usability wird durch die Verwendung von Adobe Flex Standard Komponenten soweit wie möglich versucht das Look and Feel zu erhalten das User gewöhnt sind um so eine vertraute Umgebung zu schaffen.

4.4 Architektur Übersicht

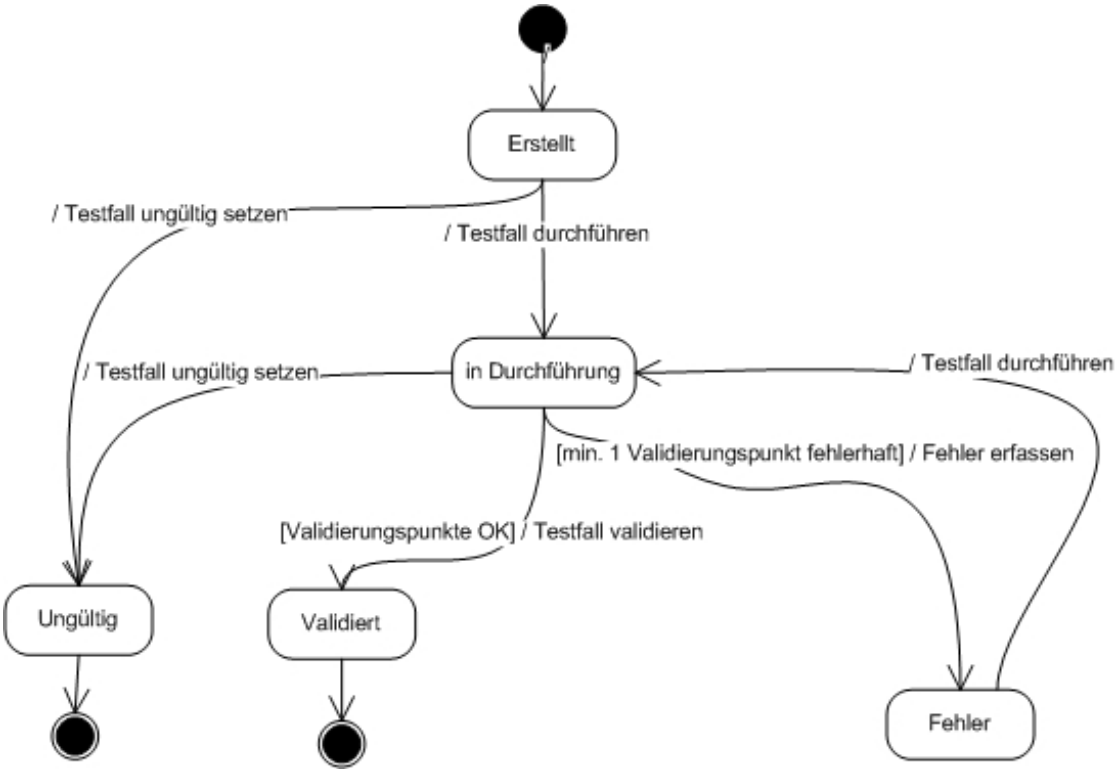

Abbildung 4.1: Übergänge von Testfallstatus

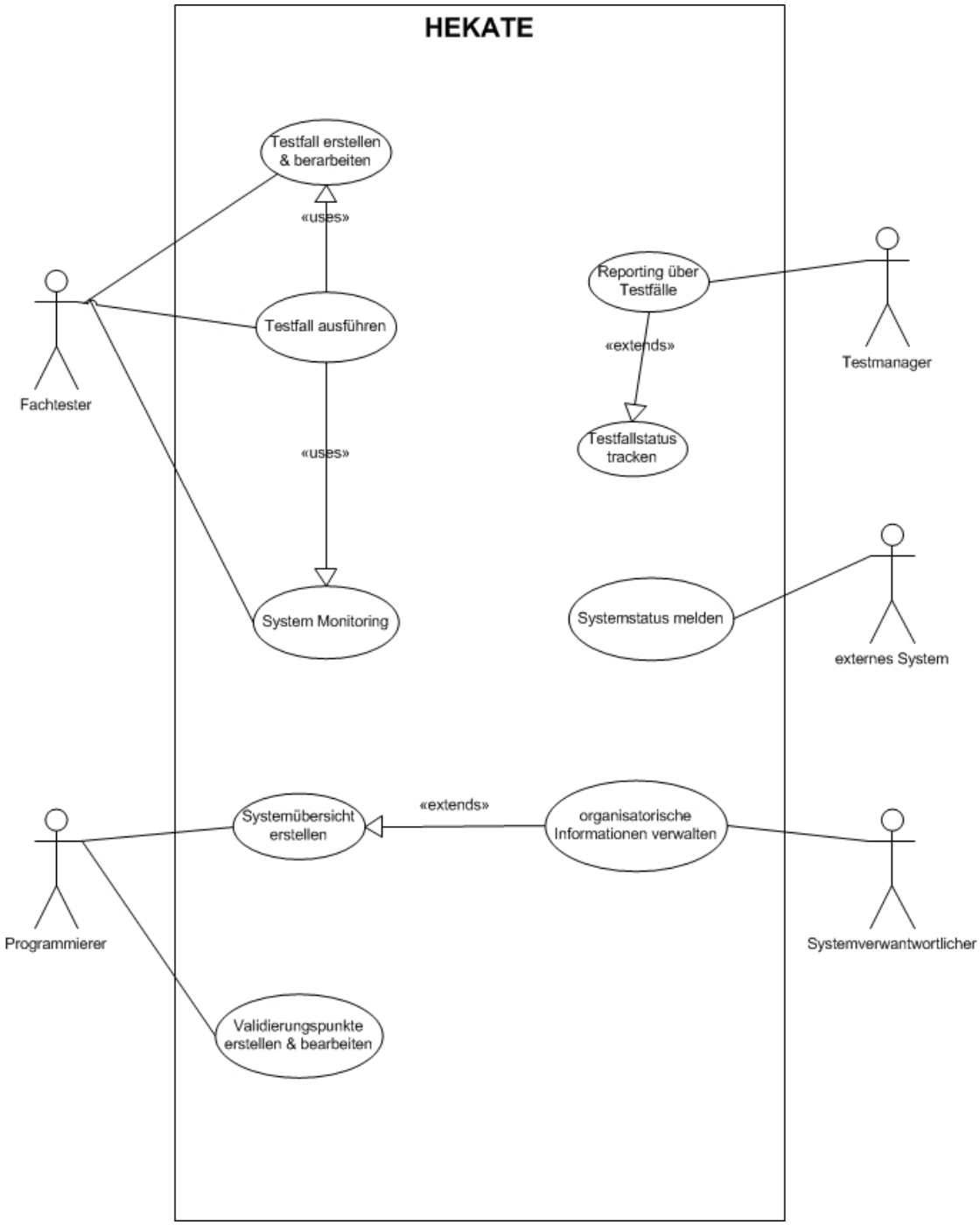

Abbildung 4.2: Use Case Diagramm

Abbildung 4.3: Anforderungen Übersicht

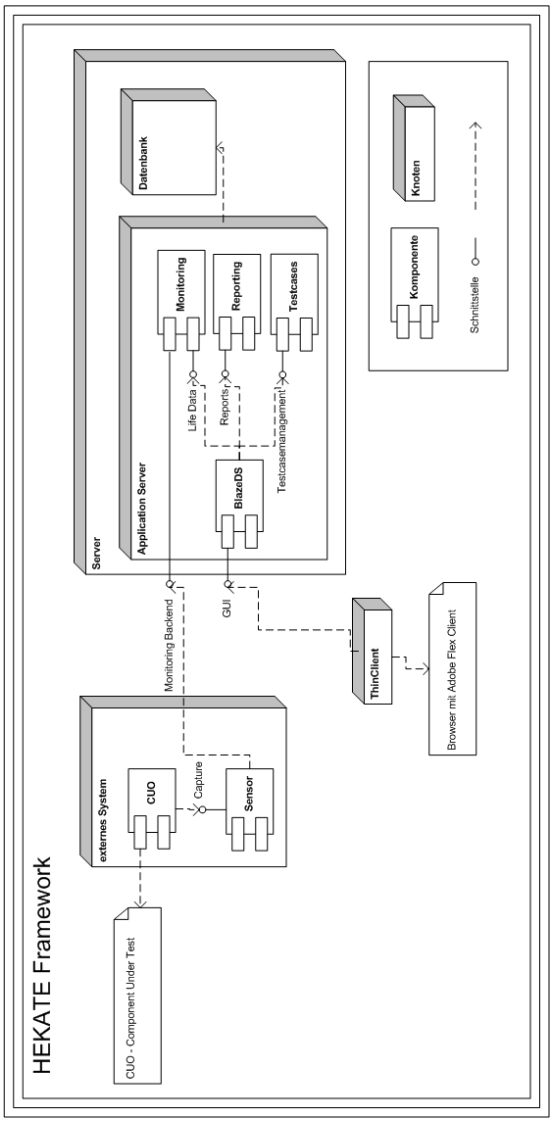

Abbildung 4.4: Architektur Übersicht

Domain-Model

Im folgenden Abschnitt werden die einzelnen Elemente die im Domain Model in Abbildung 4.5 dargestellt sind näher beschrieben. Das Domain Model enthält nicht alle Attribute die tatsächlich verwendet werden. Die nicht dargestellten Attribute wurden auf Grund der besseren Übersichtlichkeit weggelassen.

TestCase Der Testfall ist eines der zentralen Elemente des Domain Models. Jeder Testfall muss einem Benutzer zugeordnet sein. Dies ist aus organisatorischer Sicht unerlässlich. Jeder Testfall verfügt über einen Status. Der Testfall wird von dem User direkt erstellt. Zu jedem Testfall werden für die Organisation wichtige Daten wir Beschreibung oder Ziel gespeichert.

Status Der Status eines Testfalles indiziert die eindeutige Phase in der sich der Testfall gerade befindet. Es werden keine historischen Status gespeichert um die Komplexität zu reduzieren.

Role Die Rolle dient zur Vergabe von Berechtigungen bzw. zu organisatorischen Zwecken. Jeder Benutzer verfügt über eine eindeutige Rolle. Jede Rolle verfügt über eine Beschreibung die die Tätigkeit dieser Rolle innerhalb des Systems beschreibt.

ContactInformation Kontaktinformationen die jedem Benutzer zugeordnet sind, sollen vor allem zur schnelleren Abwicklung von Testfällen helfen. Da jeder Knoten ebenfalls über Kontaktinformationen verfügt, kann so beim Auftreten eines Fehlers innerhalb dieses Knotens rasch die richtige Ansprechperson gefunden und an einer Lösung gearbeitet werden.

System Ein System ist die Gesamtheit aller Knoten, die zusammen eine funktionsfähige Einheit bilden. Das System ist jene Einheit auf die sich die Testfälle beziehen und die getestet werden soll.

Node Ein System besteht aus mehreren Nodes. Jeder dieser Knoten ist eine abgeschlossene Komponente bzw. ein physisches System. Jeder Knoten enthält eine Beschreibung die dessen Aufgabe im Gesamtsystem umfasst.

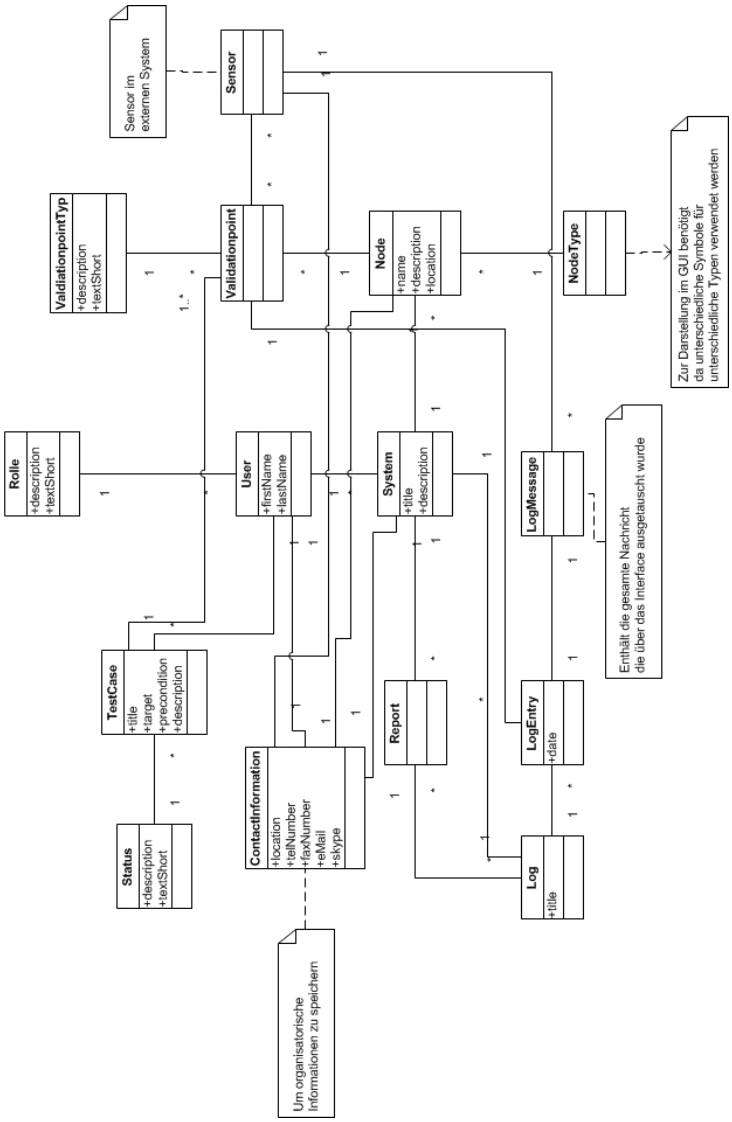

Abbildung 4.5: Domain Model

4.5 Komponente Server

Die Architektur des HEKATE Backends ist in Abbildung 4.6 dargestellt. Das Backend läuft als Dynamic Web Project in einem Applikation Server. Hier kommt Tomcat zum Einsatz, da hier die Integration mit BlazeDS problemlos gegeben ist. Für die Anbindung der Sensoren verfügt HEKATE über eine REST Schnittstelle. Die Controller sind für die Verarbeitung der Requests zuständig und rufen die Zuständigen Services auf um die benötigten Daten zu erhalten bzw. die eigentliche Verarbeitung der Business Logik anzustoßen. Die Service Schicht wird des Weiteren zur Anbindung der Flex Benutzeroberfläche verwendet. Die DAO Schicht ist für den Datenbankzugriff verantwortlich. Sie verwaltet sämtliche Operationen die auf persistenten Daten durchgeführt werden. Die einzelnen Schichten werden nun im Detail vorgestellt.

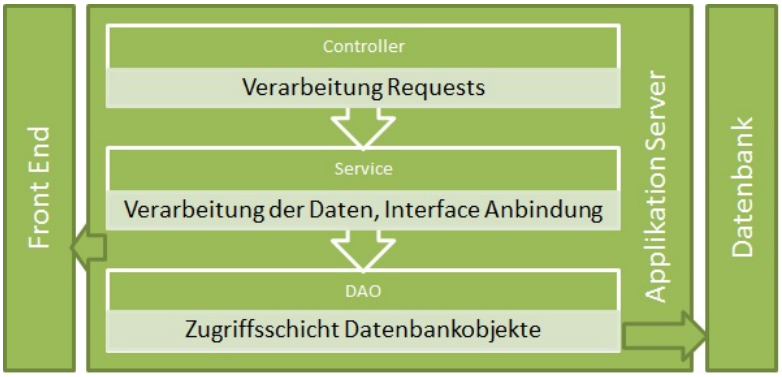

Abbildung 4.6: Architektur HEKATE Backend

Controller

Die Controller in HEKATE sind für die Bereitstellung der REST API verantwortlich. Jeder Controller stellt unterschiedliche Methoden bereit die von den Sensoren aufgerufen werden können. Diese REST Schnittstelle wurde mit Hilfe des Spring MVC Frameworks realisiert. Spring MVC übernimmt das Parsen der Requests und den Aufruf der zuständigen Controller. Das Mapping der URLs zu den entsprechenden Methoden erfolgt durch eine Annotation basierte Konfiguration direkt im Controller. Das Spring Servlet des MVC Frameworks nimmt die HTTP Requests der REST API entgegen und ordnet sie den entsprechenden Controllern und der entsprechenden Funktion innerhalb des Controllers zu. Die Rückgabewerte der Funktionen werden ebenfalls an das Spring MVC Framework übergeben. Dies serialisiert die übergebenen Werte bzw. Objekte in das JSON Format. Die einzelnen Methoden der Schnittstelle bzw. der korrekte Aufruf wird nun im Detail beschrieben.

Sensoren Registrierung

Die Registrierung der Sensoren erfolgt mittels API, und wird nicht per Hand im Front End erledigt. Dies hat den Vorteil, dass diese nicht manuell eingefügt werden müssen. Im Gegensatz

zu Systemen und Knoten werden Sensoren in großer Anzahl vorhanden sein. Aus diesem Grund wäre ein manueller Eintrag mit größerem Zeitaufwand verbunden. Die Sensoren erhalten bei der Registrierung eine GUID zugewiesen die von den Sensoren persistent in der Konfiguration gespeichert wird. Diese GUID wird für spätere Abfragen verwendet damit eine eindeutige Zuordnung erfolgen kann. Das Codebeispiel 4.1 stellt einen validen Input für dieses Inteface mit dem Mapping 4.2 dar. Diese Anfrage gibt die entsprechende ID zurück wie in 4.3 beispielhaft dargestellt. Die Sensoren teilen über diese Schnittstelle ausserdem ihre verfügbaren Methoden mit den entsprechenden Parametern mit, die als Service angeboten werden.

```
{
    "callableMethods": [
        {
            "methodParameters": [
                {
                    "name": "count",
                    "value": "3",
                    "type": "int"
                },
                {
                    "name": "stringToRepeat",
                    "value": "value",
                    "type": "string"
                }
            ],
            "methodName": "printRepeatedString"
        },
        {
            "methodParameters": [
                {
                    "name": "count",
                    "value": "3",
                    "type": "int"
                },
                {
                    "name": "stringToRepeat",
                    "value": "value",
                    "type": "string"
                }
            ],
            "methodName": "logRepeatedString"
        }
    ],
    "firstUpDate": 1299409749629,
    "sensorType": "XML",
    "systemId": "dc88e96f-67ed-4b4d-aa39-6776e690a1a4"
}
```

Listing 4.1: Registrierung Sensor JSON Beipsiel

```
/hekate/service
Methode: POST
```

Listing 4.2: Mapping Registrierung

```
{
    "regDate": 1299410678149,
    "id": "f44b4f7b-a570-41bc-85a1-c83e194001be"
}
```

Listing 4.3: Register Sensor JSON Response Example

Sensoren Command-Übermittlung

Die Methode 4.4 dient zur Steuerung der Sensoren. Diese poolen diese Adresse in regelmäßigen Abständen um Steuerungskommandos vom Backend zu erhalten.

```
Mapping: /hekate/sensor/{id}/command
Methode: GET
```

Listing 4.4: Mapping Commands

Sensoren Datenübertragung

Der Sensor übermittelt diesem Interface die erfassten Daten. Diese werden in die Datenbank geschrieben und von dort aus für weitere Verarbeitungszwecke verwendet. Es folgt keine Rückmeldung über die Verarbeitung. Der Statuscode des Responses ist immer 400 OK. Codebeispiel 4.6 demonstriert einen möglichen Input für das in 4.5 dargestellte Interface.

```
Methode: POST
```

Listing 4.5: Mapping Datenübertragung

"captureDate": 1299411218292,
"transmissionDate": 1299411218292,
"data": "NDI=",
"systemId": "dc88e96f-67ed-4b4d-aa39-6776e690a1a4"

```
"captureDate": 1299411218292,
"transmissionDate": 1299411218292,
"data": "NDI=",
"systemId": "dc88e96f-67ed-4b4d-aa39-6776e690a1a4"
```

Listing 4.6: Sensor Data JSON Response Beispiel

Sensoren Statusrückmeldung

Der Aufruf dieser Adresse erfolgt durch den Sensor in konfigurierbaren Zeitabständen und dient den Status des Sensors zu erfassen, daher ob die Komponente online oder offline geschalten ist. Diese Daten werden im Userinterface dazu verwendet dem User Feedback über die aktuellen Zustände der verschiedenen Sensoren und Komponenten zu geben. Es ist kein weiterer Input notwendig. Der Aufruf der URL aktualisiert automatisch den Status. Nach einem gewissen internen Timeout ohne Aufruf diese Schnittstelle durch den Sensor gilt dieser als offline. Codebeispiel 4.8 demonstriert einen validen Input für das Interface 4.7

```
Mapping: /hekate/sensor/{id}/status
Methode: GET
```

Listing 4.7: Mapping Statusrückmeldung

```
{
    "dataAmount": 356550,
    "upSince": 1303983972383,
    "cpuusage": 22.0,
    "ramusage": 32.0,
    "systemGUID": "63b72cbd-be60-4989-9b99-2e75ddcd0f00",
    "nodeGUID": "9c1f3846-9a89-458c-8d63-1ca54668c1dd"
}
```

Listing 4.8: Status ReportRequest Beispiel

4.6 Komponente Frontend

Im diesem Abschnitt wird das Flex Frontend näher beschrieben. Das Komponentendesign ist in Abbildung 4.7 abgebildet. Die einzelnen Schichten werden in den folgenden Abschnitten detaillierter erklärt.

Graphical User Interface-Layer

Abbildung 4.8 zeigt eine schematische Abbildung des Graphical User Interfaces(GUI). Bereich A wird für die Standard Menüleiste verwendet. Diese bietet Menüpunkte an für das Erstellen von Systemen und Testfällen. Dieser Bereich ist wie auch Bereich B statisch, daher diese werden nicht durch das Laden von Komponenten geändert bzw. sind selbst auch nicht als solche ausgeführt. Bereich C bietet hingegen Bereich für die Komponenten an. Komponenten sind Flex Module die je nach Bedarf geladen werden. Das Laden der Module wird durch die Auswahl der Menüpunkte angestoßen.

Subkomponenten die als eigene Module geladen werden sind wie folgt aufgeteilt:

System Diese Komponente dient zur Modellierung des Systems. Sie ermöglicht es dem Benutzer Knoten hinzuzufügen, grafisch anzuordnen und zu verbinden. Für das erstelle System können diverse Metadaten angegeben werden. Diese Ansicht erlaubt es dem Benutzer auch für die Knoten entsprechende Metadaten wie die organisatorische Zuständigkeit zu

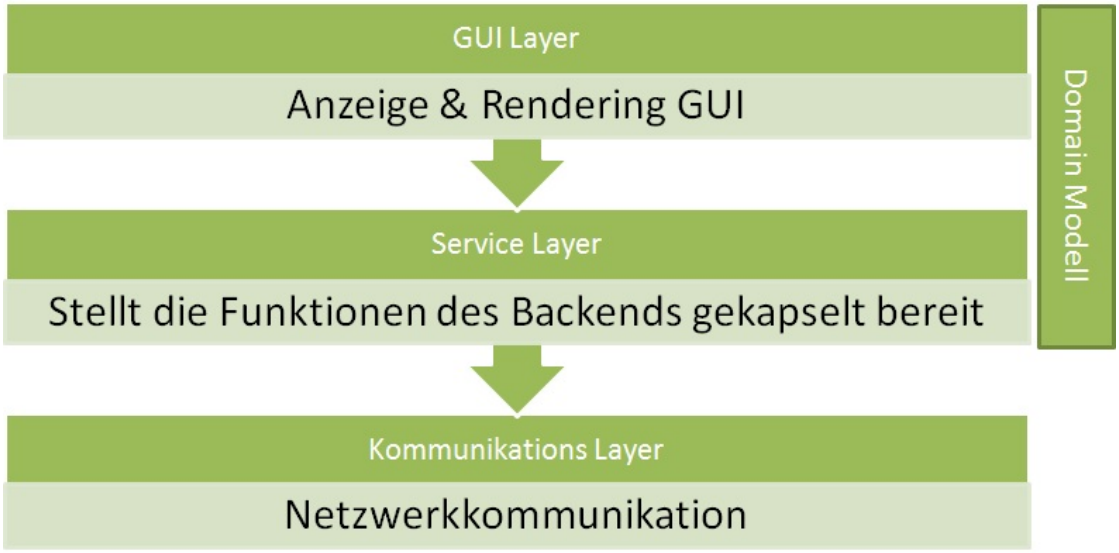

Abbildung 4.7: HEKATE GUI Architektur

hinterlegen. Es dient auch zur Anzeige der Monitoring Daten. Diese werden direkt bei dem entsprechenden Knoten dargestellt.

Reporting Das Reporting dient zur grafischen Darstellung der Reporting Daten. Es gibt den Benutzer die Möglichkeit für einen gewissen Zeitraum Daten grafisch in Form von Diagrammen darstellen zu lassen.

Testfälle Diese Komponente ermöglich das Anlegen und Ausführen von Testfällen.

Service-Layer

Der GUI Layer verwendet die gleichen Objekte (Objektnamen und Attribute) wie das eigentliche Backend. Durch diese Äquivalenz liegt deren Wiederverwendung im GUI Nahe. Durch den Bruch der Programmiersprachen zwischen Front und Backend ist dies allerdings nicht ohne weitere Modifikationen möglich. Adobe Flex ermöglicht hierfür das Binden von ActionScript Objekten direkt zu Java Objekten. Codebeispiel 4.9 und 4.10 zeigen zwei korrespondierende Objekte. Durch die Annotierung mit dem Remoteclass und dem Bindable Tag können so Java und Action Script Objekte automatisch gemapped werden. Da die Java Objekte in den Attributen zum größten Teil den im Front End verwendeten Klassen gleichen, ist es nicht praktikabel diese von Hand neu zu erstellen. Dies geschieht einmalig für neue Klassen bzw. wenn Änderungen an einer Java Klasse vorgenommen wurden.

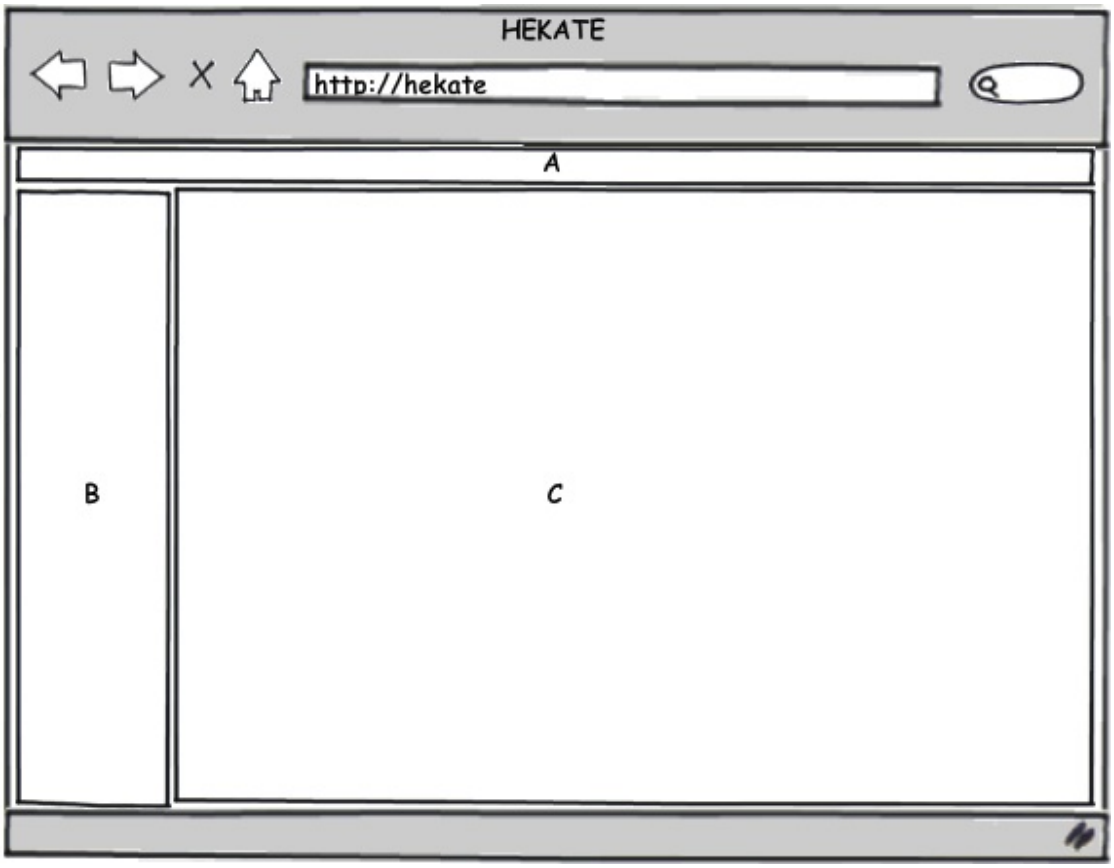

Abbildung 4.8: HEKATE GUI

```
{ package domain {
[RemoteClass(alias="at.hekate.core.node.Coordinates")]

public class Coordinates { [Bindable] public var x:int; public function
getX():int { return x; } public function setXvalue(value:int):void { x =
    value;
}

[Bindable] public var y:int; public function getY():int { return y; } public
function setYvalue(value:int):void { y = value; }


public function Coordinates() { }

} }
```

Listing 4.9: Action Script Objekt

```
2 package at.hekate.core.node;

4 import javax.persistence.Embeddable;
  import javax.persistence.Entity;
6
  @Entity
8 @Embeddable
  public class Coordinates {
10
    int x;
12  int y;

14    public int getX() {
          return x;
16    }
      public void setX(int x) {
18        this.x = x;
      }
20    public int getY() {
          return y;
22    }
      public void setY(int y) {
24        this.y = y;
      }
26

28 }
```

Listing 4.10: Java Objekt

Für HEKATE wurde hierzu ein Mapping Tool erarbeitet das aus den Java Klassen automatisch die äquivalenten Action Script Klassen erstellt. Das oben genannte Codebeispiel 4.9 wurde mittels des Tools aus 4.10 generiert. Hierbei kommt das Templatingframework Velocity zum Einsatz. Auf eine nähre Erklärung wird an dieser Stelle verzichtet da dies außerhalb des Themengebietes dieser Arbeit ist. Das entsprechende Velocity Template befindet sich im Anhang A.1.

Da der Kommunikationslayer zur Gänze von Adobe Flex mit BlazeDS gekapselt wird, soll an dieser Stelle nicht näher darauf eingegangen werden.

4.7 Komponente Sensoren

Sensoren sind Komponenten deren Aufgabe es ist bestimmte Aspekte von Systemen zu überwachen. Sie dienen zum Sammeln von Informationen in externen Systemen. Die Daten werden an das HEKATE Backend übermittelt und dort ausgewertet. Die Sensoren lassen sich ähnlich wie bei Loggingframeworks in unterschiedlichen Modi betreiben um den Overhead in der Produktionsumgebung so gering wie möglich zu halten. Für die Sensoren werden folgende Modi definiert:

Test Die Daten werden augenblicklich an das Backend geschickt.

Live Die Daten werden gesammelt und in konfigurierbaren Abständen an den Server gesammelt übertragen.

None Die Daten werden nicht gesammelt und nicht übermittelt.

Sensoren sollten an Punkten in den Anwendungen verwendet werden, die kritisch für die Ausführung eines Prozesses sind. Diese Punkte sind zum Beispiel am Beginn einer Methode die durch den Aufruf eines externen Interfaces angestoßen wird oder vor der Rückgabe. An diesen Stellen lassen sich für Integrationstest besonders wichtige Daten wie Eingangs und Rückgabewerte überprüfen. Sensoren können allerdings auch andere Aspekte wie Datenbankwerte, Daten auf dem Filesystem oder ähnliches überwachen. Die Einsatzmöglichkeiten und die konkrete Implementierung hängt stark von den Bereichen ab die durch den Integrationstest abdeckt werden sollen. Die Implementierung erfolgt mittels eines Eventsensores bei dem die gewünschten Sensoren registriert werden. Er leitet dann die eingetretenen Events an die bestimmten Sensoren weiter. Im Folgenden werden Beispielimplementierungen für Sensoren beschrieben.

Konfiguration

Die Konfiguration der Sensoren erfolgt für jeden Knoten einzeln mittels einer XML Datei die zu der XML Schema Definition in A.2 konform ist. Die Konfiguration besteht aus einem allgemeinen Teil und Teilen die für jeden Sensor spezifisch sind. Folgende Auflistung beschreibt die einzelnen Elemente der Konfiguration:

backendEndpoint Der Endpoint des HEKATE Backend Servers. Hier wird die Netzwerkaddresse im Format IP:PORT eingetragen. Der Port ist optional.

commandPoolingIntervalInSeconds Gibt die Zeitspanne an nach der die Sensoren erneut beim Backend nachfragen ob Commands für auszuführen sind.

nodeGUID Die eindeutige ID des Knotens. Wird vom HEKATE Backend bereitgestellt und muss manuell eingetragen werden.

overwriteSensorModus Überschreibt lokale Einstellungen der Sensoren.

sensorModus Überschreibt im Zusammenspiel mit overwriteSensorModus den Modus in dem sich die Sensoren befinden.

systemGUID Eindeutige ID des Systems. Wird vom HEKATE Backend bereitgestellt und muss manuell eingetragen werden.

liveConfig Dient zur Konfiguration sollte sich das System im live Modus befinden. intervalInMinutes definiert hier die Zeit nach der die gesammelten Daten an das Backend übertragen werden.

Für jeden Sensor können folgende Attribute konfiguriert werden:

callableMethods Definiert Methoden, die von außen aufgerufen werden können.

callableMethod Definiert eine bestimmte Methode, die vom HEKATE Backend aus, aufgerufen werden kann .

implementingClass Die Klasse welche die aufzurufende Methode implementiert.

methodName Der Name der Methode die aufgerufen werden kann.

methodParameters Die Parameter inklusive Name und Typ der Methode.

serviceName Der Name des Services das aufgerufen werden kann.

GUID Eindeutige ID des Sensors. Wird vom HEKAKTE Backend automatisch vergeben bei der Registrierung des Sensors.

name Ein Name des Sensors. Dient zur besseren Zuordnung für Fachtester.

sensorModus Definiert den Sensormodus.

sensorType Definiert den Sensortyp.

Code Beispiel 4.11 zeigt anhand eines Android Sensors einen Aufruf des Sensors zum Senden der Daten an das Backend von HEKATE. Die SensorFactory gibt den Sensor zurück der mit dem angegebenen Namen in der Konfigurationsdatei spezifiziert wurde. Das DataMapper Object ist ein Hilfsobject das Key-Value Paare hält. Diese können später im Frontend leicht abgefragt und ausgewertet werden. Das Code Beispiel fügt einen einzigen Wert eines Android Text-Views ein und sendet diesen anschließend an das Backend. Je nach Sensortyp sind die Daten die gesendet werden unterschiedlich.

```
AndroidSensor sensor = (AndroidSensor) SensorFactory.getInstance().getSensor(
    "android1");

ArrayList<DataMapper> data = new ArrayList<DataMapper>();

DataMapper mapper = new DataMapper();
mapper.setKey("bottom_field");
mapper.setValue(tx.getText().toString());
data.add(mapper);

sensor.sendData(data);
```

Listing 4.11: Sensor Java Beispiel

Generic-Sensor

Generische Sensoren können zum Beispiel ein- bzw. ausgehende Daten von Services ermittelten. Ein generischer Sensor übermittelt die ihm übergebenen Daten ohne weitere Überprüfungen. Abbildung 4.9 illustriert die Arbeitsweise eines generischen Sensors.

Als Nutzdaten werden binäre Daten entgegengenommen, daher können sämtliche Daten übertragen werden ohne Rücksicht auf deren Typ oder Format. Die Auswertbarkeit dieser Daten unterliegt dann dem Nutzer am Frontend.

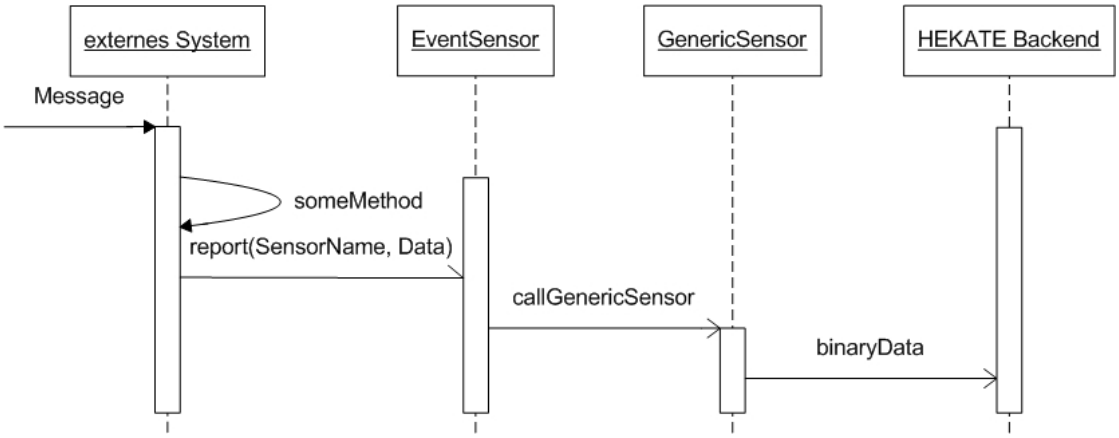

Abbildung 4.9: Generischer Sensor

Datenbank-Sensor

Ein Datenbank-Sensor liest bestimmte Daten aus der Datenbank. Dies kann zum Beispiel sinnvoll sein, wenn es zu überprüfen gilt ob der Aufruf einer bestimmten Methode die gewünschten Änderungen durchgeführt hat, oder wenn Batchimports ihre Daten in der Datenbank zwischenspeichern um die Validität der Daten zu überprüfen. Abbildung 4.10 zeigt die Arbeitsweise dieses Sensors.

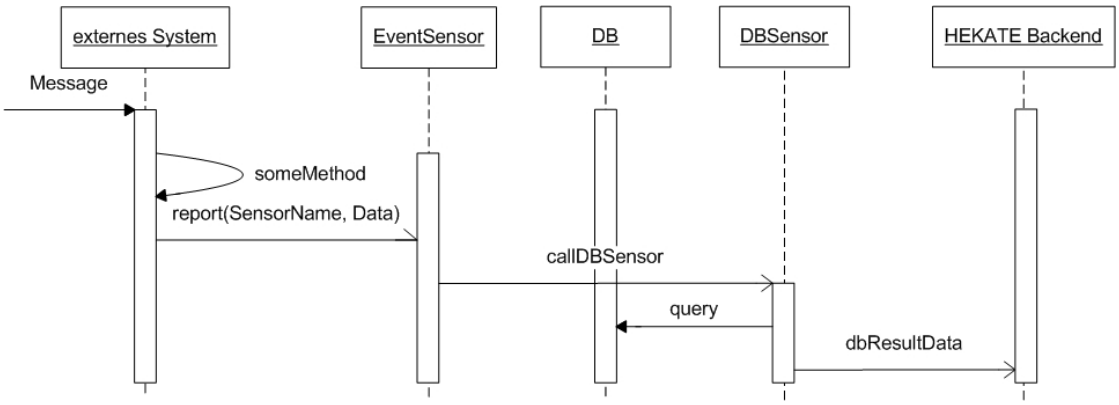

Abbildung 4.10: Datenbank Sensor

Der Sensor kann mittels Queries konfiguriert werden. Diese werden beim Aufruf des Sensors ausgeführt. Der Sensor liefert das Ergebnis der Query an das HEKATE Backend. Dort kann nun Validiert werden ob der Zustand des Systems valide ist.

XML-Sensor

Daten die über Schnittstellen ausgetauscht werden sind oft mittels XML serialisiert. Sei es direkt als Datei die per FTP übermittelt wird oder Daten die über eine Web Service Schnittstelle übermittelt werden. Datenstrukturen die in XML vorliegen, haben den Vorteil, dass sich Teile davon mittels XPATH auf bestimmte Kriterien hin auswerten lassen. Abbildung 4.11 stellt die Übermittlung der Daten an das Backend dar. Die Vorgehensweise ist mit dem des generischen Sensors gleich. Doch durch die zusätzliche Information, dass es sich hier um XML basierte Daten handelt, können im Frontend weitere Validierungen erfolgen.

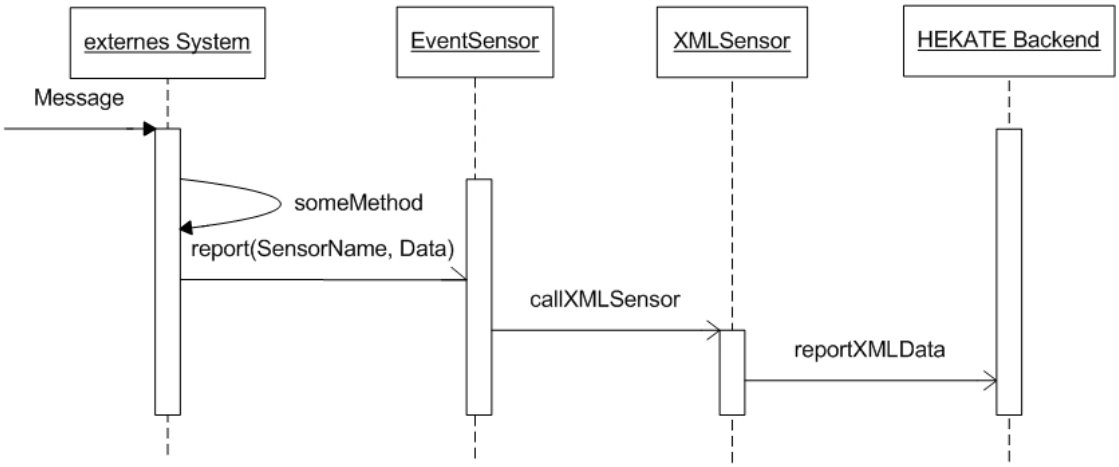

Abbildung 4.11: XML Sensor

Filesystem-Sensor

Vor allem bei Batchjobs die Datenexporten dienen werden oft Dateien auf das Filesystem geschrieben und dann per FTP, SFTP oder ähnlichen Protokoll zu einem externen System übertragen. Im Zuge eines Schnittstellentests ist es oft nötig zu wissen, ob dieses File erstellt wurde und über dessen Inhalt Bescheid zu wissen. Mittels dem in 4.12 abgebildetem Filesystemsensor können solche kritischen Punkte in Testfälle eingebunden werden. Der Sensor überwacht gewisse Verzeichnisse auf Änderungen und übermittelt diese an das HEKATE Backend zur Auswertung.

Der Filesystemsensor kann so konfiguriert werden, dass er nur bestimmte Daten miteinbezieht oder jede Änderung innerhalb eines Verzeichnisses weiterleitet.

Android: View-Sensor

Einer für den speziellen Einsatz auf der Androidplattform entwickelter Sensor überprüft bestimmte Elemente in der GUI Darstellung auf dem mobilen Endgerät. 4.13 zeigt hierbei die Vorgangsweise. Der Sensor greift auf den aktuellen View der Androidplattform zu und kann so einzelne Elemente wie ein Feld zur Anzeige von Daten auslesen und an das Backend weiterleiten.

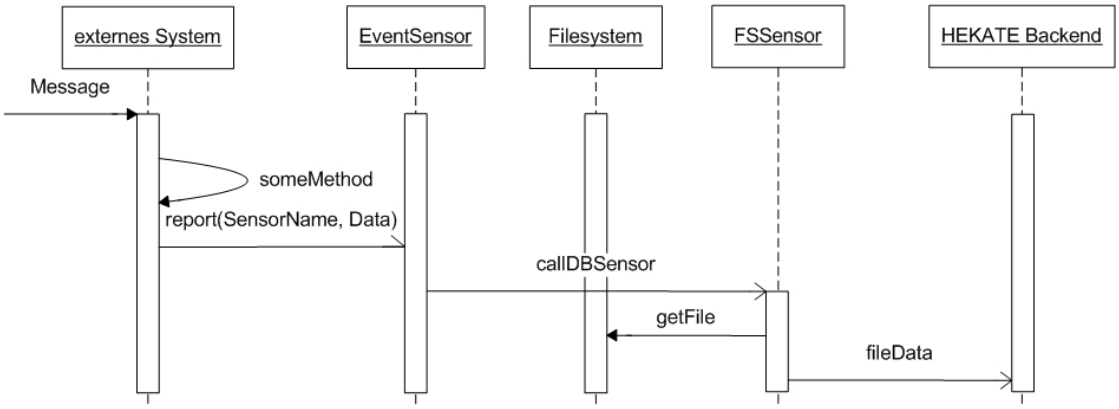

Abbildung 4.12: Filesystem Sensor

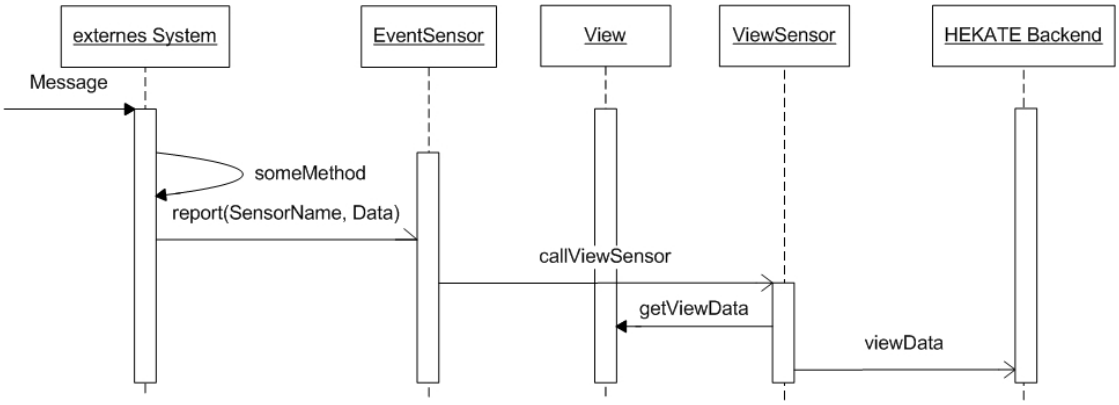

Abbildung 4.13: Android View Sensor

Durch diese direkte Überprüfung der Daten auf dem Endgerät kann ein vollständiger Integrationstest stattfinden.

4.8 Testfälle

Beschreibung

Der Komponente zum Erstellen von Testfällen kommt eine bedeutende Rolle innerhalb des Frameworks zu. Jeder Testfall verfügt über Metadaten und Daten die zur Validierung des Testfalles notwendig sind. Die Metadaten werden hauptsächlich zur Auswertung und zu organisatorischen Aktionen verwendet. Folgende Metadaten werden für jeden Testfall erfasst:

ID Jeder Testfall erhält eine eindeutige ID. Diese dient zur eindeutigen Identifizierung innerhalb des Systems.

Vor und Nachname des Erstellers Der Autor eines Testfalles muss in jedem Fall eindeutig identifizierbar sein. Dies dient nicht nur zur Zuordnung innerhalb des Systems sondern auch organisatorischen Gründen. Sollte der Testfall später durch jemand anderen ausgeführt werden, so ist der Ersteller die Ansprechperson an die sich der ausführende Tester wenden kann.

Erstellungsdatum Das Erstellungsdatum dient hauptsächlich dem Zweck bei späteren Reports eine datumsbasierte Statik erstellen zu können. Das Änderungsdatum wird dabei Standardmässig auf die Zeitzone eingestellt an dem sich das Server Backend befindet. Eine Unterstützung unterschiedlicher Zeitzonen kann bei der Auswertung erfolgen.

Status Der Status stellt für den Tester einen wichtigen Anhaltspunkt für seine Arbeitsorganisation dar. An Hand des Status erkennt der Tester ob ein Testfall noch bearbeitet werden muss oder ob dieser bereits ausgeführt und validiert wurde. Der Status ist weiters ein wichtiger Indikator für Reports.

Titel Jeder Testfall muss über einen Titel verfügen der den zu testenden Aspekt beschreibt.

Beschreibung Eine kurze Beschreibung die den Inhalt des Tests kurz beschreibt.

Ziel Das Ziel des Testfalles, daher das Ergebnis, das dieser Testfall bei erfolgreicher Ausführung produziert.

Vorbedingung Enthält eine Liste verschiedener Aspekte die vor dem Testfall erfüllt sein müssen.

Jeder ertstelle Testfall beinhaltet mehrere Validierungspunkte. Diese Validierungspunkte überprüfen die Daten die von den Sensoren geliefert werden auf bestimmte vom Tester definierte Aspekte.

4.9 Monitoring

Beschreibung

Das Ausführen von Tests bzw. deren erfolgreicher Abschluss ist auch im hohen Maße von der Information über den Status der am Test beteiligten Systeme abhängig. Sollte eine Komponente oder ein ganzes System zur Zeit der Testausführung nicht verfügbar sein, so kann dies den erfolgreichen Abschluss behindern. Fachtestern ist es oft nicht ersichtlich welche Systeme zur zeit betriebsbereit sind, bzw. ob die benötigten Komponenten entsprechend in Betrieb sind. Für Tester ist es besonders im Fall eines negativen Testresultates essentiell herauszufinden, ob des durch den Ausfall eines Systems oder durch einen Implementierungsfehler oder ähnlichem zu Stande kam. Da Fachtestern häufig der Systemüberblick bzw. das technische Know-How fehlt um die Fehler von Begin an einzugrenzen, muss erst in längeren Rücksprachen mit den Systemverantwortlichen der Fehler ermittelt werden. HEKATE bietet für diesen Fall eine Komponente zum Monitoring der am Test beteiligten Systeme. Die wichtigste Information bzw. Quality of Serivce Parameter ist die Verfügbarkeit des Systems, daher ist das System verfügbar und wann war das System nicht verfügbar.

HEKATE verwendet hierfür dieselbe Ansicht wie für das Ausführen von Testfällen. Fäll ein Knoten aus wird dieser entsprechend gekennzeichnet. Hierdurch erkennt der Tester sofort, dass dieser Knoten zur Zeit nicht zur Verfügung steht. Sollte dieser Knoten für einen Test benötigt werden, kann so direkt durch die hinterlegen Kontaktinformationen der zuständige Systemverantwortliche kontaktiert werden. Dies erspart die Suche nach dem Ort des Fehlers und lässt die Ursache auf einen einzigen Knoten einschränken.

Technische Umsetzung

Um das Monitoring von externen Systemen zu ermöglichen verwendet HEKATE die bereits erklärten Sensoren. Diese verfügen über einen speziellen Mechanismus der in den in der Konfiguration einstellbaren Abständen(Timeout) Statusreports an das HEKATE Backend liefert. Sollte ein Sensor dieses Timeout überschreiten gilt dieser als offline und wird entsprechend in der Monitoring Ansicht dargestellt. Sensoren die entsprechend dem Intervall Monitoring Daten liefern werden als online markiert. Der Statusreport der von den Sensoren erfolgt enthält folgende Daten:

Startdatum Durch dieses Datum kann der Tester ermitteln, wann dieser Sensor bzw. die damit konfigurierte Applikation gestartet wurde.

CPU Auslastung Sollte ein Knoten nicht mehr reagieren oder nur sehr langsam kann so durch den Tester eine Ursache der Systemlast entweder ausgeschlossen werden oder der entsprechende Systemadministrator informiert werden.

RAM Auslastung Wie bei der CPU Auslastung kann es durch eine zu starke Auslastung der Hardware zu Fehlern durch das nicht Einhalten von Timeouts, etc. kommen. Durch diese Kennzahlen kann der Tester so Fehlerquellen in diesem Bereich ausschließen oder weiterkommunizieren.

gesendete Daten Der Sensor kumuliert die gesendeten Daten. Dies dient zu statistischen Auswertungen.

Durch den Sensor erfolgt in dem konfigurierten Zeitintervall ein Callback über die REST Schnittstelle zum HEKATE Backend. Dies nimmt die entsprechenden Daten entgegen und persistiert diese. Des Weiteren ermittelt das Backend ob entsprechende GUI Instanzen darüber informiert werden müssen und leitet die Information an das entsprechende Frontend weiter.

4.10 Reporting

Als Reporting wird in HEKATE die Auswertung von historischen Testdaten angesehen. Durch die Verwendung von Testdaten bzw. den Einsatz der Live Daten über das Monitoring kann durch das Erstellen von Reports eine Aussage über die Zuverlässigkeit der einzelnen Komponenten getroffen werden. Die Daten der Testfälle können so kumuliert werden und eine Statistik über die entsprechenden Tests erstellt werden. Diese kann zum Vergleich der Komponenten dienen bzw. Aussagen über den aktuellen Qualitätszustand des gesamten Systems. HEKATE greift hierzu auf

die gesammelten Validierungsdaten pro Knoten zu. Die Auswertung kann zum Beispiel an Hand der Dimensionen Fehler gesamt System, Fehler pro Knoten oder Fehler pro Sensor dargestellt werden. Die einzelnen Aspekte können auf einen gewissen Zeitraum beschränkt gefiltert werden. Dies ermöglicht die Auswertung einzelner Phasen separat ohne die Verfälschung durch Fehler die in früheren Phasen entstanden sind.

KAPITEL 5

Anwendungsbeispiel

Das folgende Kapitel verdeutlicht anhand einer Anwendung die in einem Beispielprojekt den Funktionsumfang und den Nutzen des HEKATE Frameworks. Zuerst soll die Beispielapplikation beschrieben werden. Hier wird die Domain der Anwendung sowie die Requirements für diese Anwendung beschrieben. Aus diesen Anforderungen resultieren Architektur und Verteilung. Da es hier besonders auf die Verteilung und die organistorische Zuständigkeit ankommt wird ein besonders auf das Thema Deployment und Verteilung eingegangen. Des Weiteren soll eine Übersicht über die technische Implementierung gegeben werden. Um die Funktionen von HEKATE zu zeigen wird anschließend ein Testfall erstellt und ausgewertet. Basierend auf den Daten die durch den Betrieb gewonnen worden wird auch das Monitoring illustriert und ein Report für das System erstellt. Abschließend wird bewertet welchen Nutzen der Einsatz von HEKATE im hier beschriebnen Fall gebracht hat bzw. welche Nachteile dadurch entstanden sind.

5.1 Beispielprojekt: Produkt Check

Die hier vorgestellte Beispielanwendung soll es den Endbenutzern ermöglichen Produkte im Geschäften zu scannen und deren Hersteller zu identifizeren bzw. auf bestimmte Kriterien wie Fair Trade hin zu untersuchen. Hierzu sind einige unterschiedliche Systeme notwendig die gemeinsam schließlich die Daten für das Endgerät bereitstellen und dem Benutzer diese Funktion ermöglichen. Grafik 5.1 illustriert die hier vorgestellte Architektur. Das Unternehmen das diesen Dienst bereitstellen möchte ist selbst Zwischenhändler von Nahrungsmitteln und verfügt über mehere tausend Mitarbeiter und eine entsprechende komplexe IT Infrastruktur.

Die Daten werden von einem externen Lieferanten in das eigentliche System eingespielt. Der Datenexport erfolgt im XML Format und der Datentransfer erfolgt via SFTP (Secure File Transfer Protocol) auf einen Knoten innerhalb unserer Infrastruktur. Dieses XML File wird schließlich von der Core Anwendung verarbeitet und in die Datenbank eingespielt. Eine REST API wird von einem Web Server zur Verfügung gestellt damit die mobilen Endgeräte auf die Daten zugreifen können. Die Anwendung auf dem Web Server nimmt die Requests entgegen und verarbeitet dieses entsprechend. Basierend auf den erhaltenen Daten liefert das die entsprechenden Informationen an das Endgerät.

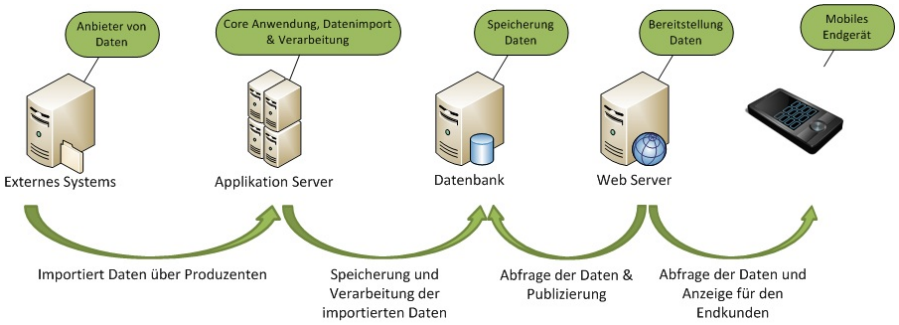

Abbildung 5.1: Beispielsystem

Des Weiteren nehmen wir ein fiktives Beispielzenario an, das ein beispielhaftes Setup der

Systemlandschaft wieder gibt. Mithilfe dieses Beispielszenarios sollen die Probleme verdeutlicht werden die in solchen verteilten Systemen bestehen.

5.2 Projekt Setup

Die hier vorgestellte Beispielanwendung ist nur ein Use Case in einem Projekt das je nach Phase zwischen 30 und 60 Vollzeitmitarbeiter beschäftigt. Dadurch kommt es zu einer starken Diversifizierung der Aufgaben. Es wurde eine eigene Testabteilung eingerichtet die das Erstellen, Durchführen und Validieren von Testfällen übernimmt. Diese Fachtester sind Fachmitarbeiter und verfügen über kein oder nur geringes Wissen und Interesse über die technischen Details. Dennoch müssen sie sicherstellen, dass die Software schlussendlich den fachlich Anforderungen entspricht. Sie erstellen Testfälle im Word, zum Tracking wird ClearQuest verwendet. Die Ausführung und Kontrolle erfolgt manuell.

Die korrekte Funktionsfähigkeit des Produktes ist essentiell. Sollte einem Kunden ein Produkt zum Beispiel als "Fair Trade" angezeigt werden obwohl es dies nicht ist, kann dies zu einem enormen Imageschaden für das gesamte Unternehmen führen. Aus diesem Grund ist der Test der Andwendug bei diesem Projekt besonders wichtiger Aspekt. Im folgenden Abschnitt werden die einzelnen Teile der Applikation vorgestellt. Technische Einzelenheiten, wie die eingesetzten Frameworks, werden dannach genauer beschrieben.

Vendor

Unser Vendor ist der Lieferant der Daten. Er verfügt über die Informationen bezüglich der Hersteller von diversen Produkten. Das für den Export verwendete Datenmodell ist in Abbildung 5.2 dargestellt. Jeder Produzent verfügt über mehrere Produkte. Jedes Produkt enthält mehrere Inhaltsstoffe die wiederum eine bestimmte Menge einer Substanz sind. Jedes Produkt kann des Weiteren in unterschiedlichen Versionen vorliegen. Für jeden Produzenten wereden Labels hinterlegt. Der Produzent kann diesen entsprechen oder nicht, bzw. die Information darüber kann unbekannt sein.

Als Produzent wird der Hersteller eines Produktes verstanden. Ein Produkt in diesem Beispiel ist ein Nahrungsmittel das von Endkunden gescannt wird. Oft gibt es eine gewissen Evolution bei Produkten und geringfügige Änderungen. Dies wird mit den Produktversionen abgedeckt. Als Label wird hier ein bestimmtes Gütekriterium verstanden. Diese Gütekriterien sind zum Beispiel Fair Trade, Sustainable oder Fair Work Conditions.

Abbildung 5.3 illustriert den Aufbau der Softwarekomponente des Vendors(der Lieferant der Daten). Die Datenschicht in unserer Beispielanwendung wird nur in Memory erzeugt, da dies absolut ausreichend ist um die Funktionalität zu demonstrieren. Basierend auf diesen Daten erfolgt der Export der genierten Daten durch einen Batchjob. Dieser Batchjob läuft innerhalb des Anwendungsservers und speichert die gegebenen Daten in ein XML File innerhalb der Infrastruktur unseres Kernsystemes. Das exportierte XML entspricht dem in A.3 gezeigtem XSD Format.

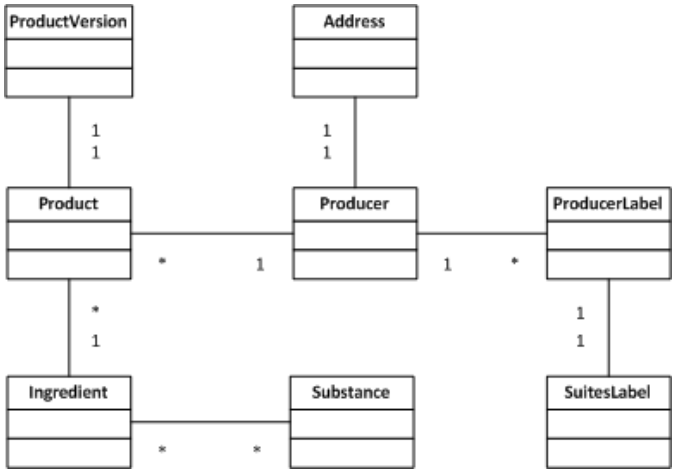

Abbildung 5.2: Domain Modell

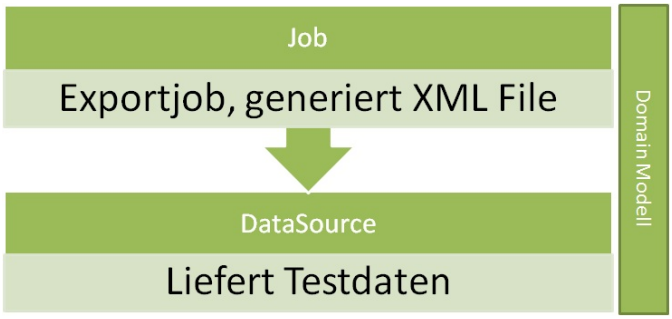

Abbildung 5.3: Vendor Beispielarchitektur

Import Anwendung

Abbildung 5.4 illustriert den Aufbau der Importanwendung. Der Job erledigt die Business Logik der Anwendung. Dieser verarbeitet die Daten und erstellt das entsprechende XML File und sorgt für die korrekte Verteilung des Files. Darunter liegt die Datenzugriffsschicht. Die DAOs (Database Access Objects) kapseln den gesamten Zugriff der Anwendung auf die Datenbank. Ihre Zuständigkeit liegt in der Verwaltung der persistenten Objekte. Diese Schicht speichert die importierten Daten in die MySql Datenbank. Das Datenbankschema entspricht dem Domain Model des Vendors. Als letzte Schicht übernimmt die Datenbank die persistente Haltung der Daten.

API

Der Aufbau der Anwendung zur Bereitstellung der API für das mobile Endgerät ist in Abbildung 5.4 dargestellt. Die API wird von den mobilen Endgeräten verwendet um Daten bzgl. der Produzenten bzw. Produkte zu erhalten. Diese API wird nicht von HEKATE verwendet, da

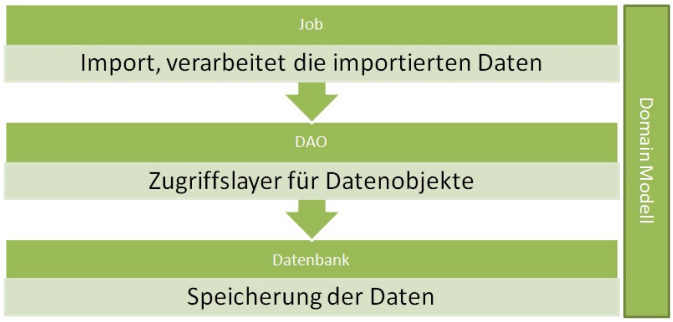

Abbildung 5.4: Import Beispielarchitektur

HEKATE von der eigentlichen Systemarchitektur der zu testenden Anwendungen unabhängig ist. Die Software ist als Dynamic Web Project aufgebaut, läuft daher in einem Application Server. In unser Beispielanwendung kommt Jetty 6[Coda] zum Einsatz. Für die Anwendung selbst kann als erste Schicht kann das Spring Frame MVC[sprb] Framework angesehen werden. Es übernimmt die korrekte Zuordnung der Requests zu den entsprechenden Controllern. An dieser Stelle sei für eine genauere Erklärung des Frameworks auf das Kapitel Hekate verwiesen. Als zweite Schicht übernehmen die Controller die Auswertung der Requests und rufen die entsprechenden Services auf. Die Controller sorgen auch für die korrekte Rückgabe der Resultate. Die Beispiel Anwendung verfügt über einen einzigen Controller der wie in 5.1 dargestellt gemappt ist. Das 5.2 zeigt die entsprechende Antwort des Servers.

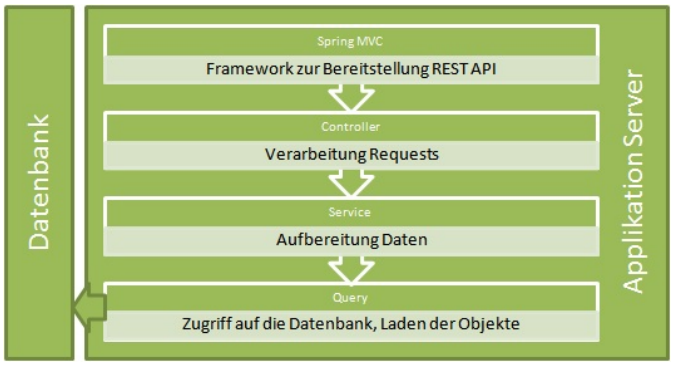

Abbildung 5.5: API Beispielarchitektur

```
http:/// localhost:port/hekate-example/hekate/producer/{id}
```

Listing 5.1: Mapping API Beispielanwendung

```
1
{
3     "address": {
          "country": "AT",
5         "city": "Vienna",
          "street": "Am Ring 3",
7         "postal": "1010"
      },
9     "name": "The Great Food Company",
      "labels": [
11        {
              "labelName": "Fair Trade",
13            "suitesLabel": "YES"
          },
15        {
              "labelName": "Sustainable",
17            "suitesLabel": "YES"
          },
19        {
              "labelName": "Fair Work Conditions",
21            "suitesLabel": "YES"
          }
23    ],
      "lastUpdate": 1304073413000
25 }
```

Listing 5.2: Response API Beispielanwendung

Der Service Layer enthält die Business Logik der Anwendung. In diesem Fall sorgt er dafür die korrrekten Daten vom nächsten Layer anzufordern und an den Controller zurückzugeben. Der Query Layer ist für das Laden der Objekte zuständig. Unsere Anwendung ist Read Only hat also keinen schreibenden Zugriff auf die Datenbank, weshalb keine DAOs verwendet werden und stattdessen ein Query Konzept verwendet wird.

Der Export der Daten erfolgt nicht equvivalent zum Domain Modell. Würde der gesamte Produzent exportiert werden Daten nach aussen freigegeben werden würden, die nicht dazu bestimmt sind. Des Weiteren würden interne Details wie Datenbank ID etc. publiziert werden die in externen Anwendungen nicht benötigt werden. Ausserdem gilt es, da die Schnittstelle zur Anbindung von mobilen Devices verwendet wird, die übertragene Datenmenge so gering wie möglich zu halten.

Android App

Der mobile Client wurde auf der Android Plattform realisiert. Die Architektur der Anwendung ist in Abbildung 5.6 dargestellt. Die erste Schicht ist der GUI Layer. Die Activity (entspricht einem angezeigten Bildschirm) ruft das entsprechende Service auf und stellt die Resultate für den User grafisch aufbereitet dar. Da das Interface für den späteren Test von Bedeutung ist wird hier näher darauf eingegangen. Der angezeigte Bildschirm (View), ist technisch als ListView implementiert, daher eine Liste mit einer Kopfzeile und einer Fusszeile die zwischen diesen beiden Elemente

weiter Zeilen enthält. Diese Zeilen werden verwendet um die Labels des Herstellers des Produktes anzuzeigen, das abgefragt wurde.

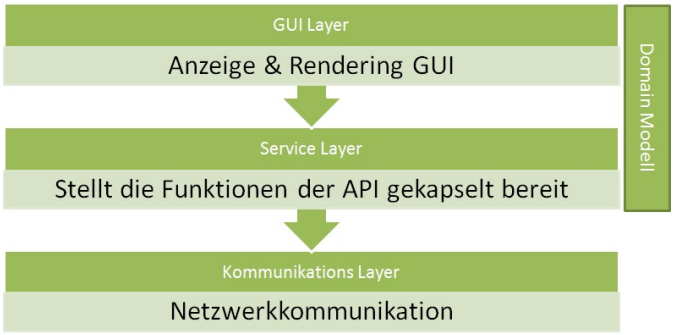

Abbildung 5.6: Android Beispielarchitektur

Der Service Layer kapselt den Zugriff auf die remote API. Er stellt die entsprechenden Methoden zur Verfügung damit die Anwendung die benötigten Daten erhält. Da es sich um ein Remote Service handelt wird noch eine weitere Schicht verwendet. Diese dient zur Kommunikation mit der Remote API und übernimmt die eigentliche Übermittlung der Daten. Im Falle unserer Anwendung bindet sie das entsprechende REST Service ein und mapped das erhaltene JSON zu den equivalenten Objekten. Hier wird der Jackson Mapper[Codb] verwendet. Dieser übernimmt die Aufgabe gleichnamige JSON Attribute zu den jeweiligen Objekt Attributen zu kopieren. Das Datenmodel entspricht dem Domain Modell des Vendors.

5.3 Technische Umsetzung

Im Folgenden sollen die verwendeten Frameworks in der Beispielanwendung kurz vorgestellt werden. Die Frameworks werden nur kurz beschrieben, da eine genauere Beschreibung der umfangreichen Funktionen nicht im Themenbereich dieser Arbeit liegt.

Maven

Maven[**?**][Son] ist ein Software Projekt Management Tool das auf dem Project Object Model (POM) basiert. Durch die Verwendung können Abhängigkeiten von Projekten, Builds, Reporting und die Dokumentation von einer zentralen Stelle aus gesteuert werden. Durch die Verwendung eines bereitgestellten Repositories können durch Maven benötigte Bibliotheken geladen und in das Projekt eingebunden werden. Alle Beispielsubprojekte verwenden Maven um ihre Abhängigkeiten wie Spring, Hibernate und Jackson zu verwalten.

Spring

Das Spring Framework[sprd] ist ein Application Framework für Java. Es ist Open Source verfügbar und würde erstmals 2004 veröffentlicht. Spring verfügt zur Zeit über mehrere Teilprojekte[sprc]. Die aktuelle Version ist 3.0.

Unsere Beispielanwendung verwendet Spring als Inversion of Control Contrainer[spra], also zu verwalten der Abhängigkeiten der Klassen. Des Weiteren verwendet die API Spring MVC[sprb] (Model View Controller) für die Bereitstellung der REST API und Mapping der Requests.

Hibernate

Hibernate[Mag] ist ein ORM (Object-relational mapping) Framework für Java. Die Hauptaufgabe ist es das objektorientierte Domain Modell der Anwendung auf ein Schema für eine rationale Datenbank zu mappen. Hibernate sorgt also für Anbindung an eine Datenbank und verfügt über diverse Mechanismen um Objekte darin zu Verwaltung bzw. zu suchen. Hibernate liegt in der aktuellen Version 3.6.3 vor. Hibernate wird in unserer Beispielanwedung beim Import und bei der API eingesetzt.

MySQL

Als Persistenzschicht wird in unserer Beispielanwendung MySQL[Orac] eingesetzt. MySQL ist ein rationales Datenbank Management System das frei verfügbar ist. Es liegt in der aktuell in der Version 5.5 vor.

SimpleXML

SimpleXML[Sim] wird von unserer Beispielanwendung verwendet um Java Objekte in XML zu serialisieren und aus den serialiserten Daten wieder Java Objekte zu erstellen. SimpleXML hat gegenüber JAXB den Vorteil einer erheblich geringern Grösse der benötigen Bibliotheken und der Verfügbarkeit auf der Android Plattform. JAXB ist nicht auf der Android Plattform verfügbar[Gooa] und kann auch nicht als externeAbhänigkeit eingebunden werden.

Jackson Mapper

Jackson Java JSON-processor[Codb] ist ein Framework zur Konvertierung von Java Objekten in das JSON Format und vice versa. Die aktuelle Version ist 1.7.6. Jackson Mapper wird einerseits durch Spring verwendet um die Responses in ein JSON Format zu konvertieren und durch die Clients unserer Beispielapplikation um die Daten der API wieder in Java Objekte zu konvertieren. Neben den technischen Aspekten spielt auch die organisatorische Zuorndung eine wesentliche Rolle. Diese soll im folgenden Abschnitt beschrieben werden.

5.4 Beispielszenario

Unser Projekt hat folgende organisatorische Zuordnungen:

Vendor

Firma D. liefert die benötigten Daten. Da auch für Firma D. der Export der Daten direkt zu einem Kunden via XML Neuland ist und die Schnittstelle neu entwickelt wurde, muss auch auf dieser Seite mit Fehlern gerechnet werden.

Frau F. ist hier für die Dateneingabe zuständig. Um bestimmte Beispielszenariien für Testfälle aufzubauen muss also für jeden Testfall Frau F. bemüht werden um bestimmte Daten einzugeben. Diese Daten werden per Excel Datei übermittelt. Diese Datei enthält kein Standardformat wodurch es zu Missverständissen in der Interpretation der Daten kommen kann. Des Weiteren ist Frau F. nur für eine Stunde in der Woche für das Projekt verfügbar. Dies führt zu deutlichen Verzögerungen in der Eingabe und daher in der Ausführung der Testfälle.

Herr E. ist für die technischen Belange des Systems zuständig. Sollten zum Beispiel Daten eingegeben wurden sein, allerdings nicht im XML File vorhanden sein, muss Herr E. die entsprechende Fehleranalyse durchführen. Auch Herr E. ist nicht komplett für dieses Projekt abgestellt sondern muss anderen Aufgaben nachkommen. Eine Fehleranalyse kann daher durchaus 2-3 Arbeitstage in Anspruch nehmen.

Die technische Verteilung der Systeme ist in Abbildung 5.7 dargestellt:

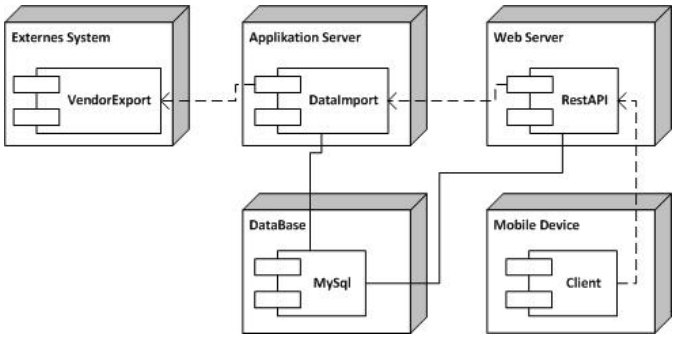

Abbildung 5.7: Verteilungsdiagramm Beispielanwendung

Kernsystem

Da unsere Beispielfirma bereits über eine entsprechende IT-Infrastatuktur mit den dazugehörigen Personal verfügt, stehen die einzelnen Knoten selbst innerhalb des System unter unterschiedlicher personeller Verwantwortung. Herr I. ist für den Knoten zuständig der generell alle Importe von Daten im Unternehmen regelt. Dadurch ist er auch für die Anwendung verantwortlich die unsere Daten vom Vendor in unsere Kernapplikation einspielt. Herr A. ist für die Entwicklung und Verwaltung unserer Kernapplikation zuständig. Er hält die technische Projektleitung inne. Sollten Probleme innerhalb der Anwendug auftreten ist er der entsprechende Ansprechpartner für den Tester. Herr D. ist für die Verwaltung der Datenbankserver zuständig. Sollten Überprüfungen direkt auf der Datenbank notwendig sein bzw. Zugänge etc. für die Tester verlangt werden liegt dies in seinem Zuständigkeitsbereich. Frau W. ist für die Betreuung des WebServers zuständig.

Sie ist Ansprechperson zu Fragen bei Fehlern die in der Schnittstelle zum Endgerät auftreten. Herr G. ist der verantwortliche Entwickler für das Mobile Device.

5.5 Problemstatement

Basierend auf dem oben beschriebenen Szenario ergeben sich folgende Probleme:

Durch die geringe Verfügbarkeit der Mitarbeiter bei Firma D. kann sich die Ausführung von Testfällen über 1-2 Wochen hinziehen. Dies kann folglich zu einer Verschiebung des Abschlusses bzw. des Releases des Projekts führen wenn es sich hierbei um kritische Testfälle handelt. Durch das manuelle übermitteln der Daten kann es zu Fehlern in der Interpretation und der Eingabe der Daten kommen. Die zeitliche Verzögerung der Eingabedaten und der Ausführung der Testfällen kann bei den Testern zu einem Verlust des eigentlichen Kontextes des Tests führen und so die Qualität der Testausführung beeinträchtigen. Durch die Einbindung des Personals entstehen zusätzliche Kosten die abgedeckt werden müssen. Ausfälle durch Urlaub oder Krankheit sind hier besonders kritisch. Dies kann soweit führen, das Testfälle nicht ausgeführt werden können und die Abnahme des Releases nicht durch geführt werden kann. Ebenfalls gilt dies für die Auswertung der Testfälle. Oft wird technische Unterstützung von den Fachtestern benötigt um die Auswertung der Logfiles vornehmen zu können. Die Fachtester müssen oft Auswertungen in der Datenbank oder Log files vornehmen um die Korrektheit zum Beispiel von Batchjobs überprüfen zu können. Da sie allerdings nicht über die nötigen Fertigkeiten verfügen kann die Qualität der Überprüfungen leiden. Durch die zusätzlichen technischen Details um die sich die Fachtester kümmern müssen, können sie sich nicht genügend auf die eigentliche Aufgabe, das Finden von fachlichen Fehlern, konzentieren. Des Weiteren kommt es zu einer demotivierenden Atmospähre für die Fachtester da sich diese ständig im technischen Bereich überfordert fühlen. Durch die diveresen Punkte an denen Fehler auftreten können, gestaltet sich die Fehlersuche oft langwierig und schwierig. Dem Fachtester fehlt oft der Überblick an welchen Stellen im System Fehler auftreten und wie diese zu finden sind.

Im Folgenden wird nun ein Testfall vorgestellt der mittels HEKATE durchgeführt werden soll.

5.6 Verwendung von HEKATE

Der folgende Abschnitt geht näher auf die benötigten Schritte ein die durchlaufen werden müssen um mit HEKATE ein entsprechendes Testsetup zu erstellen. Folgende Punkte sind in der angegebenen Reihenfolge zu beachten:

- Erstellung System
- Identzifzierung kritischer Punkte
- Setup Sensoren
- Registrierung Sensoren

- Erstellung Testfall
- Ausführung und Überwachung

Am Beginn müssen alle Knoten des Systems identifiziert werden. Diese können in HEKATE entsprechend modelliert werden und werden mit Kontak und Standortinformationen hinterlegt. Die Knoten werden nun ihrer Kommunikationsstruktur entsprechend verbunden um die Zusammenhänge klarer grafischen darzustellen. Dies hilft auch im nächsten Schritt kritische Punkte innerhalb des Systems zu finden die für Tests von Mehrwert sein können. Abbildung X zeigt das entsprechend modellierte System für unser Beispielprojekt. Für das System müssen nun die einzelnen zuständigen Personen hinterlegt werden. Diese können später von allein eingesehen werden um so eine rasche Kommunikationsafnahme zu ermöglichen. Die Auffindung von kritschen Punkten sollte gewissen Kriterien genügen um einen Mehrwert zu bieten. Für unsere Beispielanwendung wurden folgende Kriterien festgelegt:

menschliche Interaktion Ein kritischer Punkt ist überall dort vorhanden, wo die Ausführung eines gewissen Aspektes die Interaktion durch eine Person erfordert.

Schnittstellen Ein kritischer Punkt ist überall dort vorhanden, wo Schnittstellen zwischen Komponenten Daten serialisieren und wieder deserialiseren müssen.

Zeitlicher Aspekt Ein kritischer Punkt ist überall dort vorhanden, wo die Ausführung durch eine Komponente verzögert werden kann, da diese zum Beispiel nur zu fixen Zeiten ausgeführt wird.

Persistierung Ein kritischer Punkt ist überall dort vorhanden, wo Daten persistent gespeichert werden. Dies sind gut überprüfbare Stellen.

Für unser Anwendungsbeispiel werden wir zur besseren Übersicht nur folgende kritische Punkte überwachen:

- ExportFile mittels Filesensor
- Import Mittels XML Sensor
- Android mittels Android Sensor

Diese Punkte werden sich als Validierungspunkt in den Testfällen wiederfinden. Für diese Testfälle muss zuerst ein entsprechendes System modelliert werden.

5.7 Systemmodellierung

Die Modellierung des Systems erfolgt im entsprechenden Systemeditor in HEKATE. Abbildung 5.8 zeigt das System bei der Modellierung. Die entsprechenden Knoten können mittels unterschiedlicher Symbole dargestellt werden. Dies dient zur besseren Veranschaulichung und leichteren Zuordnung von Knoten zu ihren tatsächlich existierenden physischen Geräten. Die

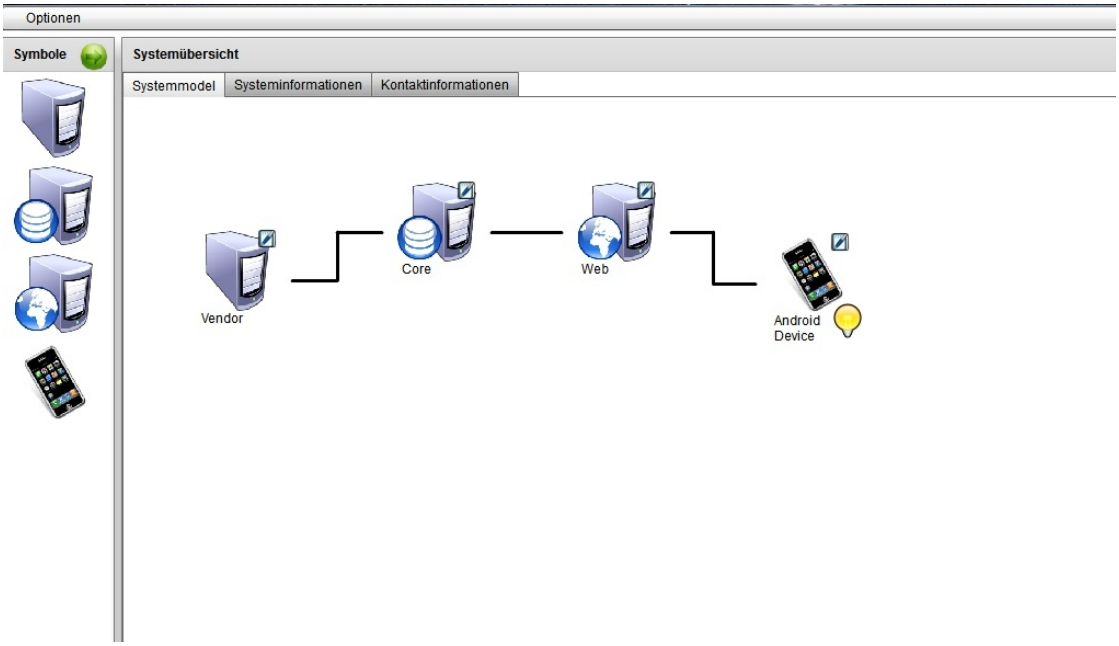

Abbildung 5.8: Systemmodellierung

Verbindungen zwischen den Knoten dienen dazu, um spätere ein bessere Vorstellung vom Datenfluss zu erhalten.

Die Abbildung illustriert die 4 Knoten, der Vendor Knoten, Core, Web und das mobile Endgerät. Abbildung 5.9 zeigt eine Darstellung der Detailinformationen die zu jedem Knoten verfügbar sind bzw. angelegt werden können. In diesem Fall die Kontaktinformationen die den Testern später zur Verfügung stehen.

Testfallübersicht

Das Anlegen und Überwachen der Testfälle wird in Abbildung 5.16 dargestellt. Unter Testfallinformationen kann der Tester entsprechende Attribute wie Ziel, Vorbedingung, etc. angeben. Auch das entsprechende System kann ausgewählt werden. Nach der Auswahl des Systems erhält der Tester die entsprechende Ansicht der Knoten die zuvor modelliert wurden. Die eigentlichen Testinformationen werden schliesslich bei den jeweiligen Knoten hinterlegt. Jeder Knoten verfügt über mehrere Symbole. Abbildung 5.10 enthält die entsprechenden Referenzen zu den einzelnen Icons.

Im Folgenden wird der Einsatz der Sensoren im Speziellen erklärt. Bei den Konfigrationsbeispielen wurde für diesen Zweck nicht relevante Teile des XML Baumes entfernt um eine kompaktere Darstellung zu erreichen.

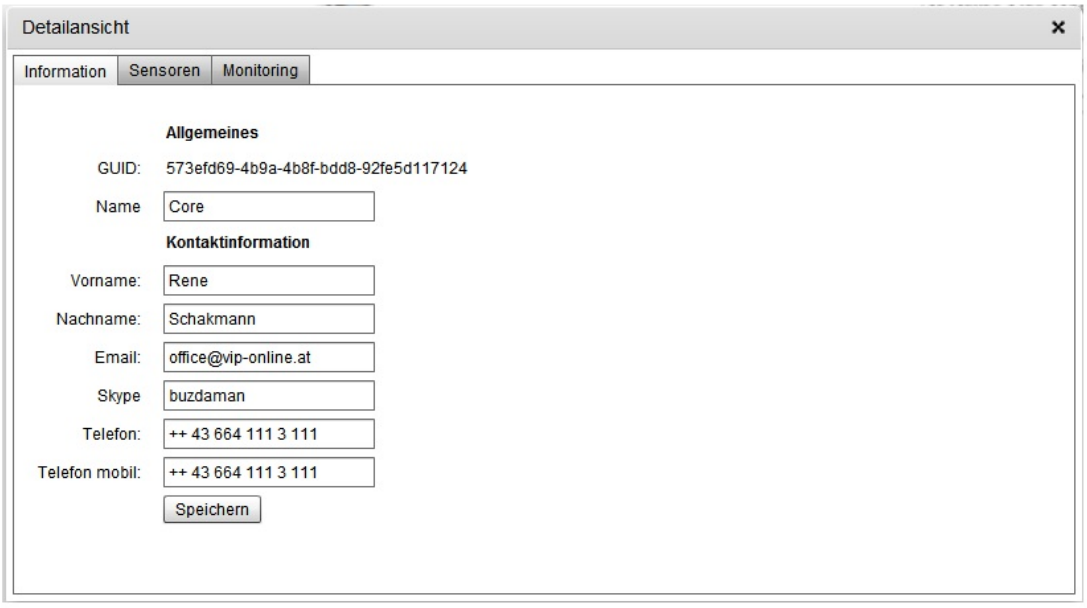

Abbildung 5.9: Detailinformation Knoten

5.8 Anwendungsbeispiel: File-Sensor

Am Knoten des Vendors bietet sich der Einsatz eines File Sensors an. Daher der Sensor stellt fest ob und wann ein Export erfolgte. Da es für den Tester von Vorteil ist diesen Export selber anstossen zu können, bietet der Sensor ein Service für diese Funktion an. Es sind keine Parameter notwendig, da ein kompletter Export durchgeführt wird. Durch diese Funktion kann der Tester unabhänig von anderen Faktoren einen Export starten und so Testfälle die einen Export beinhalten zu jeder beliebigen Zeit durchführen, ohne auf fixe Exportintervalle oder ähnliches Rücksicht nehmen zu müssen. Konfigurationsbeispiel 5.3 zeigt hierfür notwendige HEKATE Konfiguration. Eine Besonderheit bei diesem Sensor ist, dass hier ein Service (TestService) angeboten wird. Dieses stellt Funktionen bereit, dass später vom Tester aus dem HEKATE Frontend aufgerufen werden können.

Abbildung 5.10: HEKATE Icons

```
1
<sensorConfig>
3   <sensors class="java.util.ArrayList">
      <sensor>
5       <sensorModus>LIVE</sensorModus>
        <name>fileSensor1</name>
7       <callableMethods class="java.util.ArrayList">
          <callableMethod>
9           <implementingClass>at.hekate.example.vendor.TestService
            </implementingClass>
11          <serviceName>ImportService</serviceName>
            <methodName>initiateExport</methodName>
13          <methodParameters class="java.util.ArrayList" />
          </callableMethod>
15      </callableMethods>
        <sensorType>FILE</sensorType>
17    </sensor>
    </sensors>
19  <generalConfig>
      <backendEndpoint>http://localhost:8080/HekateBackend/
21    </backendEndpoint>
      <systemGUID>69868850-c2ce-4f1d-aca5-21011557480a</systemGUID>
23    <nodeGUID>400c44a1-d0ea-4c27-a18d-38806b5fab24</nodeGUID>
      <overwriteSensorModus>false</overwriteSensorModus>
```

```
25      <sensorModus>LIVE</sensorModus>
        <commandPoolingIntervalInSeconds>5</commandPoolingIntervalInSeconds>
27    </generalConfig>
      <liveConfig>
29      <intervalInMinutes>10</intervalInMinutes>
      </liveConfig>
31  </sensorConfig>
```

Listing 5.3: Konfigurationsbeispiel Vendor Knoten

Abbildung 5.11 beinhaltet die nötige Konfiguration für den entsprechenden Validierungspunkt. Der Validierungsausdruck der in diesem Fall verwendet wird, enthält eine spezielle für den Filesensor gültige Syntax. Durch den Prefix after wird sicher gestellt, dass die entsprechende Datei über ein Änderungsdatum nach dem angegebenen Datum verfügt. So kann festgestellt werden ob diese Datei exportiert wurde und ob es sich um eine aktuelle Datei handelt oder von einem früheren Export.

Die Matching Attribute ermöglichen des Weiteren eine genauere Zuordnung zu einem Testfall. Es kann zum Beispiel angegeben werden, dass Daten die von einem Sensor registriert werden nur unter bestimmten Umständen zu diesem Testfall gehören. Diese Funktion wird in diesem Beispiel allerdings nicht benötigt.

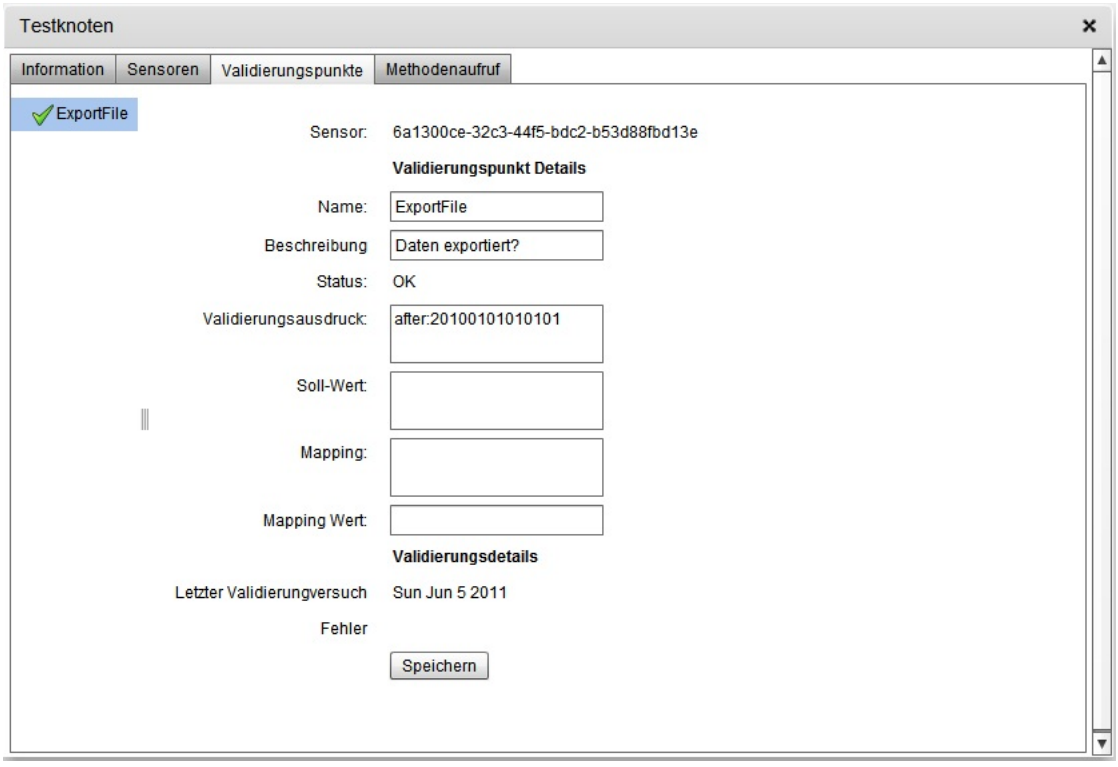

Abbildung 5.11: Validierungspunkt Vendor

Des Weiteren ist ersichtlich, dass der Validierungspunkt mit einem entsprechenden Icon für valide Validierungspunkte gekennzeichnet ist. Dieses Icon ist abhängig vom Status des Punktes. Der Validierungspunkt in der Abbildung wurde bereits erfolgreich validiert und befindeet sich im Status OK. Daher es wurde ein File exportiert das den angegebenen Bedingungen genügt.

5.9 Anwendungsbeispiel: XML-Sensor

Der nächste Sensor befindet sich auf Seite des Cores und überprüft bestimmte Attribute des XML Files das importiert wird. Konfigurationsbeispiel 5.4 zeigt die hierfür nötige Konfiguration. Wie an der Konfiguration ersichtlich enthält der Knoten nicht nur einen XML-Sensor sondern ebenfalls einen Sensor für die Datenbank. Dieser wird im nächsten Abschnitt erklärt. Der XML Sensor liefert XML Daten an das Backend. Diese können nun wie in Abbildung 5.12 entsprechend mittels XPATH validiert werden. Der in der Abbildung dargestellte Validierungspunkt befindet sich im Status PENDING. Dies bedeutet, dass dieser Knoten bisher nicht validiert wurde, daher keine Daten von diesem Sensor für den entsprechenden Testfall vorliegen. Der Validierungspunkt überprüft ob in der entsprechenden XML Datei der XML-Knoten BarCode den entsprechenden Wert 42 enthält. Ist dies nicht der Fall würde der Validierungsknoten dies bei der nächsten Rückmeldung durch den Sensor anzeigen.

```
<sensorConfig>
  <sensors class="java.util.ArrayList">
    <sensor>
      <sensorModus>LIVE</sensorModus>
      <name>xmlSensor1</name>
      <callableMethods class="java.util.ArrayList" />
      <sensorType>XML</sensorType>
    </sensor>
      <sensor>
      <sensorModus>LIVE</sensorModus>
      <name>dbSensor1</name>
      <callableMethods class="java.util.ArrayList" />
      <sensorType>DATABASE</sensorType>
    </sensor>
  <generalConfig>
    <backendEndpoint>http://localhost:8080/HekateBackend/
    </backendEndpoint>
    <systemGUID>69868850-c2ce-4f1d-aca5-21011557480a</systemGUID>
    <nodeGUID>573efd69-4b9a-4b8f-bdd8-92fe5d117124</nodeGUID>
    <overwriteSensorModus>false</overwriteSensorModus>
    <sensorModus>LIVE</sensorModus>
    <commandPoolingIntervalInSeconds>5</commandPoolingIntervalInSeconds>
  </generalConfig>
  <liveConfig>
    <intervalInMinutes>10</intervalInMinutes>
  </liveConfig>
</sensorConfig>
```

Listing 5.4: Konfigurationsbeispiel Core Knoten

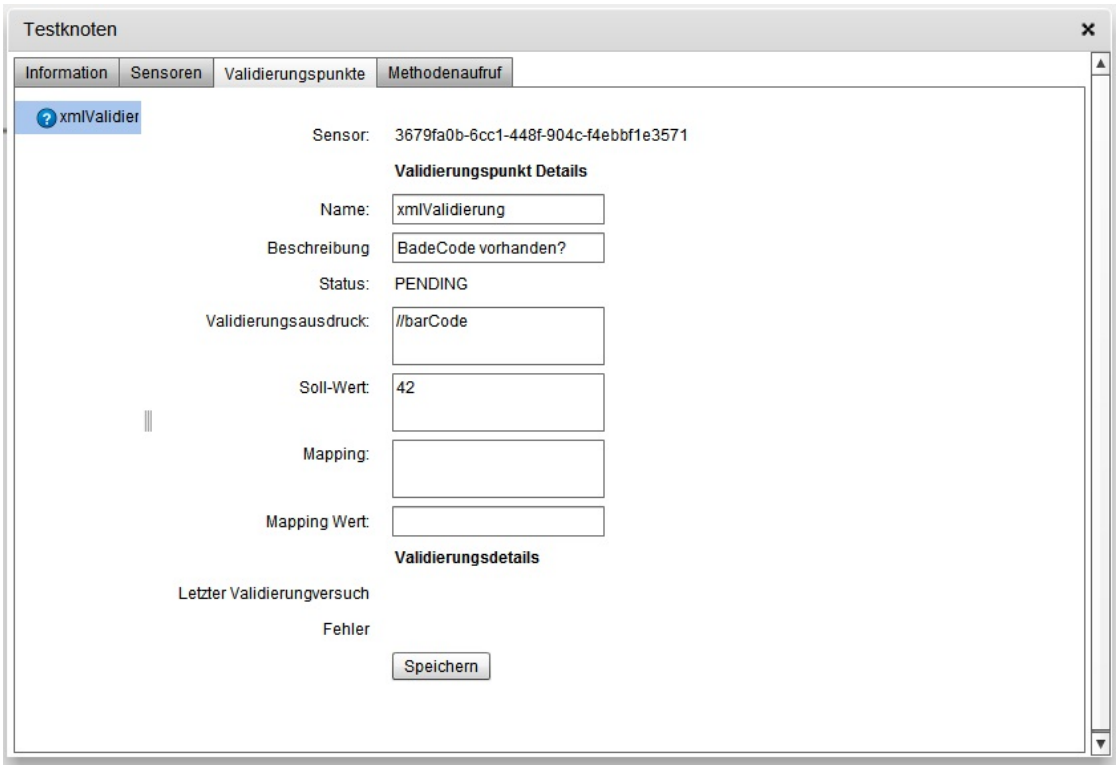

Abbildung 5.12: Validierungspunkt XML Core

5.10 Anwendungsbeispiel: DB-Sensor

Der Core sollte des Weiteren sicherstellen, dass die importierten Daten auch in der Datenbank gespeichert sind. Hierfür stellt HEKATE einen Datenbanksensor bereit. Dieser überprüft ob die entsprechenden Einträge in der Datenbank vorhanden sind. Die Konfiguration für diesen Sensor befindet sich in 5.4. In Abbildung 5.13 sieht man den entsprechenden Ausdruck für den Validierungspunkt. ProducerCount ist ein Wert der vom Sensor übermittelt wurde. Dieser wird mit dem vom Tester eingegebenen Wert verglichen. Wie ersichtlich, erwartet der Tester einen Wert von 2, der Sensor lieferte einen Wert von 1. Dies ist im Feld Fehler abgebildet. Dieses gibt Aufschluss wieso der entsprechende Validierungspunkt nicht valide ist, und sich wie abgebildet im Status FAILED befindet. Dies wird auch durch das entsprechende Icon dargestellt.

5.11 Anwendungsbeispiel: Android-Sensor

Für das mobile Endgerät wird der spezielle Sensor für Android Systeme verwendet. Dieser erlaubt es, Daten direkt aus dem Userinterface abzugreifen und entsprechend zu prüfen. So können die Daten bis zur absolut letzten möglichen Stelle auf Fehler kontrolliert werden. Abbildung 5.14 zeigt eine Darstellung des Interfaces. Für die korrekte Funktionalität ist es vor allem wichtig,

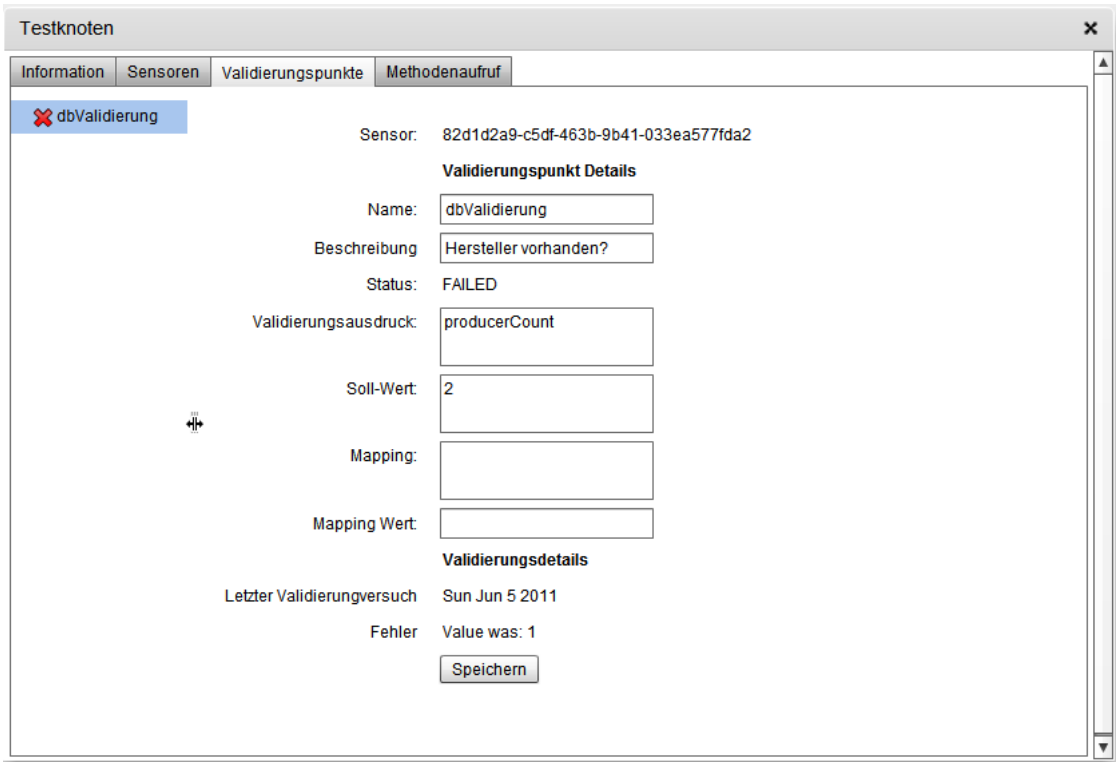

Abbildung 5.13: Validierungspunkt Core Datenbank

dass die in der Liste aufgeführten Labels der Hersteller die richtigen Werte enthalten.

Nachdem Empfang der Daten durch das mobile Endgerät und erst nach der Aufbereitung des Interfaces für den User extrahiert der Sensor die Werte direkt aus der Darstellung und leitet sie an das HEKATE Backend weiter. Durch diese Methode erhält das Backend die selben Daten wie sie auch tatsächlich dargestellt werden, und Fehler in der Aufbereitung der Daten für das User Interface, etc. können ausgeschlossen werden.

Codebeispiel 5.5 stellt die dazugehörige Beispielkonfiguration für den Android Knoten dar. Der Sensor verfügt über keinerlei aufzurufende Methoden und das commandPoolingIntervalInSeconds ist wegen des mobilen Datennetzes und der damit verbundenen langsameren Reaktionszeit und geringern Übertragungsrate erhöht worden.

```
<sensorConfig>
  <sensors>
    <sensor>
      <GUID>6d227e59-4e0e-4660-b985-c35ea6658af3</GUID>
      <sensorModus>TEST</sensorModus>
      <name>androidSensor</name>
      <sensorType>ANDROID</sensorType>
    </sensor>
    <generalConfig>
      <backendEndpoint>localhost</backendEndpoint>
```

```
11          <systemGUID>systemGUID</systemGUID>
            <nodeGUID>nodeGUID</nodeGUID>
13          <commandPoolingIntervalInSeconds>
            15</commandPoolingIntervalInSeconds>
15        </generalConfig>
      </sensors>
17  </sensorConfig>
```

Listing 5.5: Konfigurationsbeispiel Android mobiles Endgerät

Der hier beispielhaft gewählte Testfall soll ein Integrationstestfall sein der die komplette Systemlandschaft abdeckt. Daher wird beginnend vom Vendor, über den Export, Import bis hin zum Endgerät geprüft ob der entsprechende Use Case richtig umgesetzt wurde.

Es soll überprüft werden ob für ein gescannte Produkt die entsprechenden Daten auf dem mobilen Client korrekt ausgegeben werden. Beim Vendor werden die entsprechenden Daten zum Import freigeben und entsprechend exportiert. Nach dem Import werden die Daten in der Datenbank persistiert. Der mobile Client fragt mittels API den Produzenten eines bestimmten Produktes und dessen Labels ab. Abbildung 5.14 zeigt eine Abbildung des User Interfaces des Android Clients. Wie ersichtlich, wird der letzte Datensatz mit dem Attribute Fair Work Conditions mit YES angezeigt.

Abbildung 5.15 zeigt den dazugehörigen erstellen Validierungspunkt. Dieser Validierungspunkt zeigt dem Tester einen entsprechenden Fehler an, da die Darstellung im GUI nicht den geforderten Wert NO enthält. Die Auswertung des GUIs erfolgt automatisch nach dem Start der Anwendung ohne zusätzlichen Aufwand durch den Tester.

Der Validierungsausdruck ist eine eindeutige ID des Feldes im GUI. Dieses muss dem Tester vorher bekannt gegeben werden und kann frei gewählt werden.

Ein weiterer wichtiger Punkt ist das Testfall übergreifende Monitoring von Knoten. Dies wird im folgenden Abschnitt näher beschrieben.

5.12 Monitoring & Reporting

HEKATE stellt dem Bentutzer alle Daten bereit um weitere Auswertungen vorzunehmen. Da Reporting und Monitoring an sich sehr umfangreiche Themengebiete sind die hier nicht abzudecken sind, wurden in HEAKTE nur einige Beispiele implementiert die die grundlegende Funktionsweise darstellen.

Abbildung 5.17 zeigt ein Reporting über die Testfälle. Es werden die Anzahl der jeweiligen Testfälle in jedem bestimmten Status angezeigt. Dies hilft um den Fortschritt des Testvorgangs feststellen zu können.

Abbildung 5.18 illustriert die Fehleranzahl pro Knoten. Diese Metrik wird als Beispiel für einen QoS Parameter herangezogen werden.

In HEKATE erfolgt das Monitoring der Knoten, in für Tests wesentlichen Punkten. Wie bereits beschrieben gibt HEKATE mittels des Offline Icons Feedback über den Status eines Knoten. Das Monitoring der Verfügbarkeit eines Systems ist für Tests essentiell. Abbildung 5.19 stellt eine Ansicht der von einem Knoten erhaltenen Daten dar. So kann zu jedem Zeitpunkt ein Knoten überwacht werden.

Abbildung 5.14: Android GUI

5.13 Vergleich HEKATE

Abbildung 5.20 illustriert eine beispielhafte Durchführung des Testfalles im Vergleich zwischen einer Umgebung mit und ohne HEKATE. Am Beginn wird der Testfall vom Tester erstellt. Dies beginnt am Vormittag eines Arbeitstages und endet um 10 Uhr. Danach werden die benötigten Daten an Firma D. übermittelt. Da die zuständige Bearbeiterin nicht am gleichen Tag Zeit findet werden die Daten erst am nächsten Tag eingegeben. Dies erfolgt am Vormittag. Da der Datenimport allerdings erst um 20 Uhr Abends erfolgt kann der Tester auch an diesem Arbeitstag den Testfall nicht durchführen. Am darauffolgenden Tag kontrolliert der Fachtester die Daten und stellt einen Eingabefehler fest. Dies führt zu einem weiteren Tag Verzögerung. Schlussendlich sind alle Daten korrekt importiert und können per API bereitgestellt werden. Der Tester überprüft die Daten am Endgerät und übersieht auf dem mobilen Endgerät den letzten nicht korrekten Eintrag in der Tabelle. Als Resultat ergibt sich ein falsch validierter Testfall der mehere Tage benötigt hat um durchgeführt zu werden und so zu enventuellen Verzögernungen bei einem

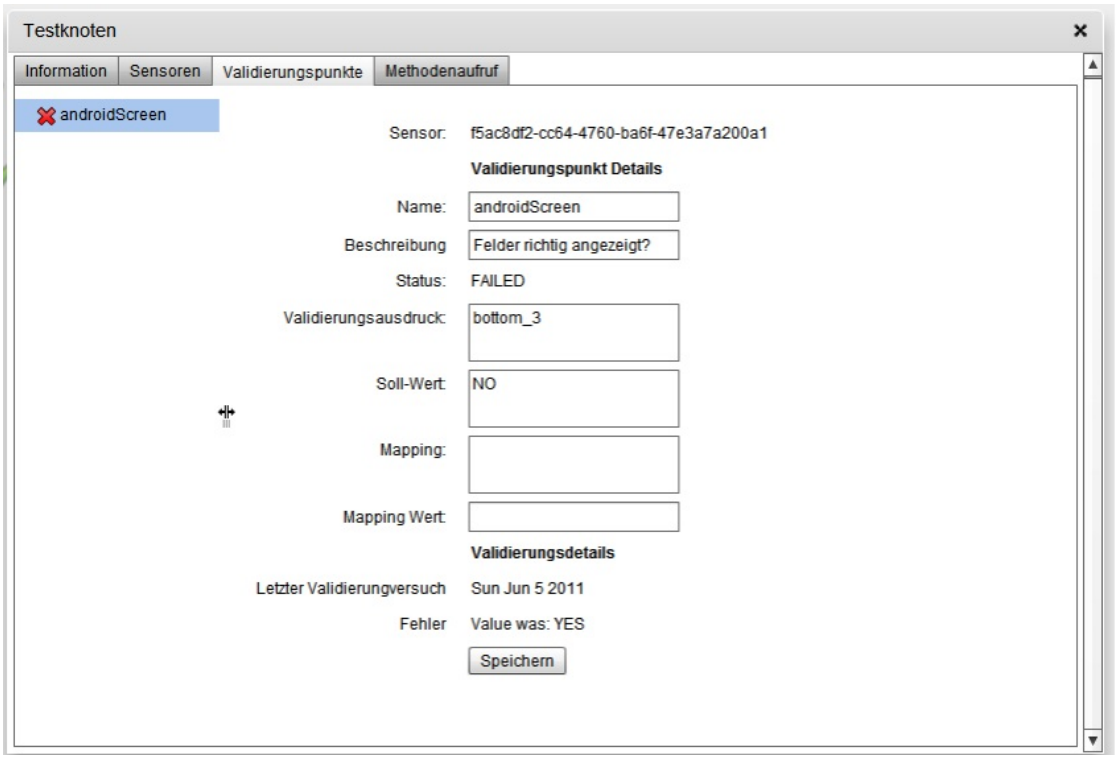

Abbildung 5.15: Validierungspunkt Android

Release führen kann.

Der Ablauf mit HEKATE erfolgt in folgenden Schritten. Der Fachtester erstellt wiederrum den Testfall am Vormittag. Durch HEKATE ist er allerdings in der Lage die benötigten Daten direkt anzufordern und kann so den Testfall sofort ausführen. Durch die maschinelle Verarbeitung wird der Fehler bei der manuellen Eingabe auf Seiten der Firma D. vermieden. HEKATE überprüft die Eingaben auf dem Endgerät mit dem erstellen Testfall vollständig. Daher wird auch der letzte Datensatz der sich zum Beispiel nur durch Scrollen erreichen lassen würde und unkorrekte Daten enthält erkannt und als nicht validiert gekennzeichnet. Der Testfall benötigt bei der Erstellung im Vergleich mehr Zeit da dieser genauer spezifiert wird. Dies hat allerdings den weiteren Vorteil, dass dies nur einmal durchgeführt werden muss. Die automatische Überprüfung im Anschluss erfolgt um ein vielfaches schneller als die Überprüfung aller Parameter durch einen manuellen Test.

5.14 Zusammenfassung der Testergebnisse

HEKATE bringt vor allem bei stark verteilten Systemen mit unterschiedlichen organisatorischen Zuständigkeiten starke Vorteile im Test. Durch die Verwendung des Frameworks können Tests autonomer von Fachtestern durchgeführt und ausgewertet werden. Durch die teils automatisierte Auswertung verschiedener Validierungspunkte kann der meschenliche Tester unetstützt werden

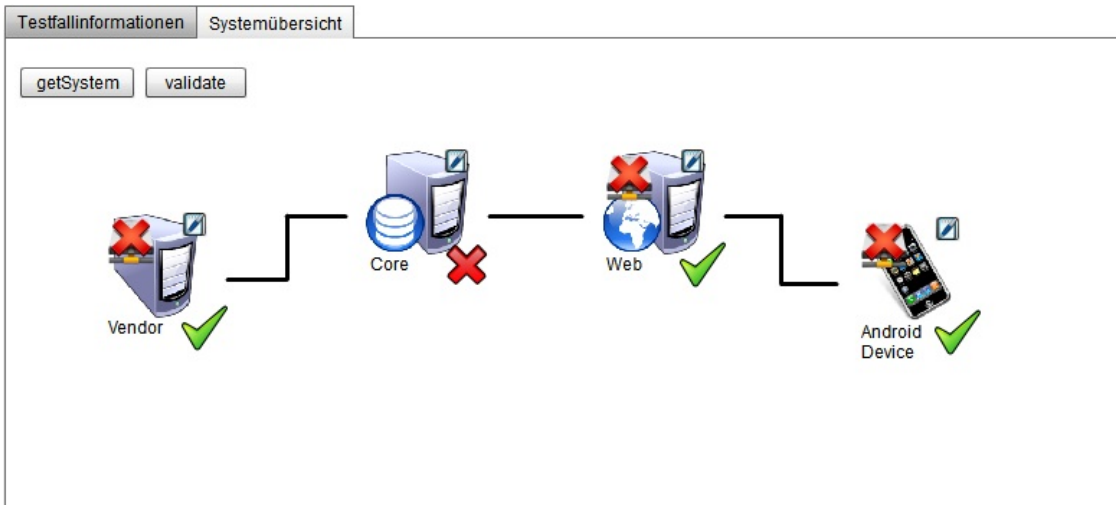

Abbildung 5.16: Übersicht Testfallknoten

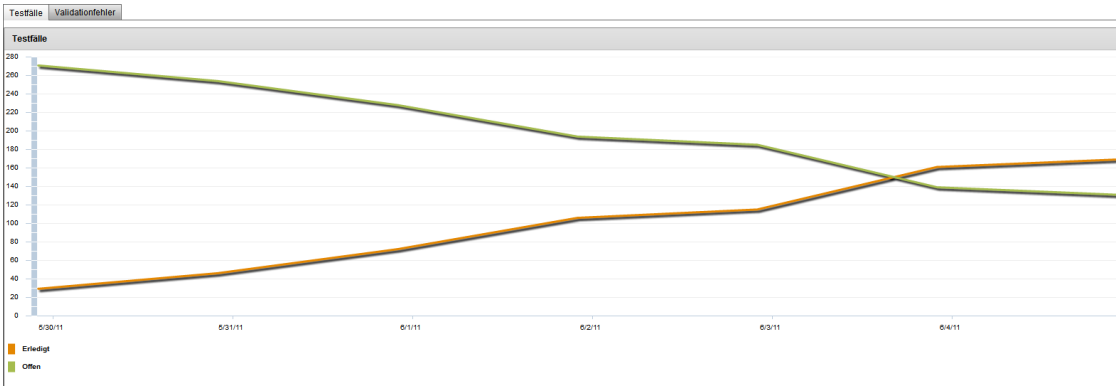

Abbildung 5.17: Reporting Testfälle

und so Fehler vermieden und dadruch die Qualität der Tests verbessert werden. Durch die Möglichkeit der Tester mittels HEKATE Aktionen in fremden Systemen anzustossen, unabhängig von der Verfügbarkeit von Personal, können Tests rascher durchgeführt werden. Der Beispielvergleich zeigt, dass in bestimmten Szenarien die Durchführungszeit von Tests verkürzt werden kann. HEKATE ermöglicht es des Weiteren einen guten Systemüberblick über die in die Test involvierte IT-Infrastruktur zu erhalten. Durch das entsprechende Monitoring können so Ausfälle von Knoten die zu Verzögernungen oder Fehlern in Tests führen erkannt und rasch darauf regaiert werden. Durch die hinterlegten Kontaktdaten ist eine rasche Kontaktaufnahme möglich und die Fehlerursachen können behoben und Tests wieder ausgeführt werden. Sollte ein Knoten entlang der Teststrecke ausgefallen sein, kann durch die Ausführung bestimmter Aktionen am nächst funktionierend Knoten der Test zumindest Teilweise fortgesetzt werden. Durch die grafische Unterstützung erleichtert HEKATE den Testern technische Überprüfungen

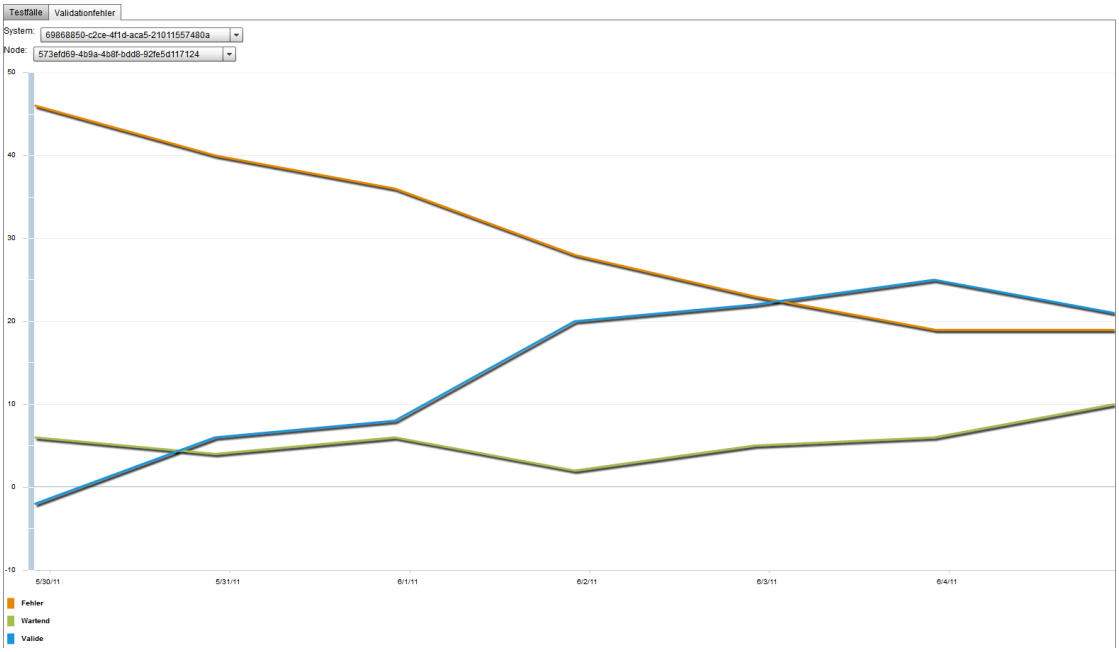

Abbildung 5.18: Reporting Fehler

durchführen zu können. Durch die Eingabe von Validierungen können Tester so festlegen welche Ergebnisse korrekt sind. Sollte ein Test erneut ausgeführt werden muss vom Tester nicht erneut ein Logfile oder ähnliches auf Korrektheit überprüft werden, sondern HEKATE übernimmt diese Auswertung und sichert so eine rasche Validierung der Tests. Optimierungen gilt es vor allem in der Usability der Spezifierung von korrekten Testresultaten vorzunehmen. Um den Tester noch besser zu unterstützen gilt es hier grafische Unterstützungen zu entwerfen die ihm helfen Validierungspunkte einfacher und korrekt zu setzen ohne eine bestimmte Sytanx erlernen zu müssen. Um auch personalisierte Prozesse zu unterstützen gilt es entsprechende Funktionen vorzusehen, zum Beispiel das Festlegen von eigenen Status für Testfälle, hinzufügen von Rollen oder Berechtigungen. Die genauere Modellierung eines Knoten könnte bei komplizierten System ebenfalls Vorteile bringen, so dass zum Beispiel Adapter etc. eigenes modelliert werden können. Durch die zusätzliche benötigte Konfiguration und Erstellung von Adapter und Anbindungspunkten des Frameworks kommt es zu einem Mehraufwand. Dieser rentiert sich allerdings je länger das Projekt läuft und amortisiert sich rasch mit den Zeiteinsprarungen bei der Durchführung der Tests. Des Weiteren kommt es dadurch zu einer Steigerung der Qualität bei der Testdurchführung und mehr Fehler können aufgedeckt werden.

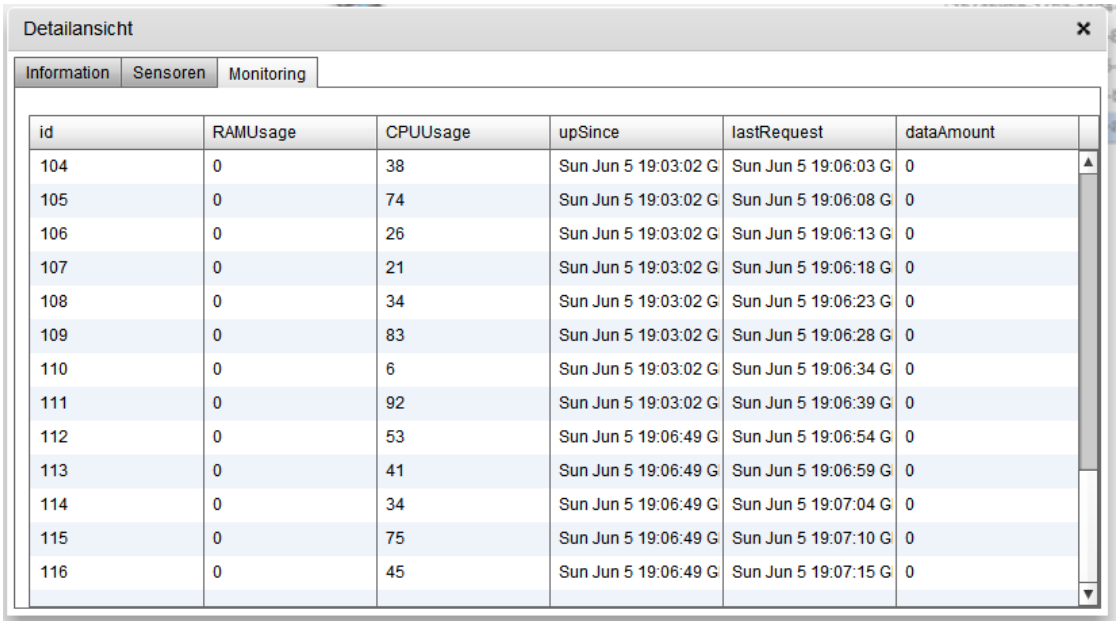

Abbildung 5.19: Monitoring Fehler

	Uhrzeit	Normal	HEKATE
Tag 1	08:00	Erstellung Testfall	Erstellung Testfall
	09:00		
	10:00	Übermittelung Daten	
	11:00	Warten auf Eingabe der Daten	
	12:00		Durchführung Testfall & Überprüfung
	13:00-18:00		
Tag 2	13:00	Eingabe der Daten	
	20:00	Export der Daten	
Tag 3	08:00	Eingabefehler wird gefunden	
	12:00	Neue Eingabe	
	20:00	Erneuter Export	
Tag 4	08:00	Überprüfung Daten auf Endgerät	

Abbildung 5.20: Testfalldurchführung Vergleich

KAPITEL 6

Zusammenfassung

6.1 Zusammenfassung

Softwareanwendungen in der heutigen IT-Landschaft sind meist verteilte Systeme die über örtliche und organisatorische Grenzen hinaus interagieren. Unternehmen bieten ihre Dienste as-A-Service an. Auch Unternehmen intern werden viele Applikationen nicht mehr nur von einer einzigen Stelle aus entwickelt und betrieben. Die vorhandene Infrastruktur muss genauso berücksichtig werden die wie personellen und organisatorischen Aspekete. Die Erstellung von Software deren Teile an unterschiedlichen Orten und Zuständigkeiten läuft, ist vor allem im Test dieser Anwendungen eine Herausforderung für die fachlichen Tester. Diese verfügen oft nur über Wissen innerhalb ihrer Domain und wenig technisches Verständnis und Wissen. Ihnen die Arbeit so einfach wie möglich zu machen muss desshalb ein essentielles Ziel des Testsmanagements seien. Durch die Unterstützung der Fachtester durch HEKATE können sich diese besser auf die eigentlich Sicherstellung der fachlichen Korrektheit und der damit verbundenen Qualität der Software konzentireren. HEKATE stellt hierzu einen Ansatz und ein Framework vor, das in der vorliegenden Version bereits für diverse Szenarien eine gute Unterstützung für die Tester darstellt.

6.2 Ausblick

HEKATE stellt eine gute Basis für Erweiterungen bereit die diese Anwendungsdomain weiter unterstützen. Der Ausbau von HEKATE ist in diversen Punkten vorstellbar. Das Monitoring Service kann durch den modularen Aufbau leicht auf weitere Systemattribute ausweitet werden.

Auch vorstellbar wäre es bestehende Log Frameworks wie log4j in HEKATE einzubinden, um so das gesamte logging an einer zentralen Stelle abwickeln zu können. Diese Daten könnten ebenfalls in das Monitoring und Reporting einfliesßen.

HEKATE bietet des Weiteren nur rudimentären Support zur Unterstüzung verschiedener Entwicklungs- bzw. Testprozesse. Der Ausbau des Frameworks hin zur Definition von Workflows des gesamten Testmanagements inklusive Controlling würde weitere Vorteile bringen. Die Ergänzung durch diverse Statistiken etc. würde für das Testmanagement genauere Erkenntnisse über den aktuellen Testablauf vorbringen. Auch könnten so Stellen identifiziert werden an denen es häufig zu Problemen kommt, und so an diesen Knoten angesetzt und Verbesserungen vorgenommen werden. Durch die gezielte Suche und Elemenierung dieser Engpässe kann der gesamte Testbetrieb beschleunigt und dadurch die Testphase verkürzt werden. Die HEKATE Architektur bietet sich durch die Möglichkeit der Aufrufe von Funktion in externen Systemen auch zur Automatisierung von Tests an. In Zusammenspiel mit den Validierungspunkten könnten so automatisierte Tests erstellt und für Regressiontests verwendet werden. Bisher gibt es nur Adapter für Java für die Sensoren. Durch die Verwendung der REST Schnittstelle zur Kommunikation mit dem Backend ist es allerdings auch einfach mögliche zusätzliche Umgebungen wie C++ oder PHP Frontends anzubinden. Die Erweiterung und Bereitstellung von Adaptern für diese Bereich ist essentiell für den effektiven Einsatz von HEKATE. Jede Stelle an der das Framework eingebunden werden kann vergrößert den Vorteil den Tester daraus ziehen können.

Auch im Bereich der Sensoren können Erweiterungen vorgenommen werden. Die Einbindung der Sensoren mittels aspektorientierter Programmierung könnte die Verwendung erheblich vereinfachen. So könnten breitere Bereiche innerhalb der Software mit geringerm Aufwand abgedeckt

werden. HEKATE hat beispielhaft einige Sensorentypen demonstriert. Durch den modularen Aufbau können ohne größere Aufwände weitere Sensorentypen hinzugefügt werden. Native Sensoren für unterschiedliche Datenbanken (SQL, wie NoSQL, etc.) würden die Fähigkeiten des Frameworks entsprechend erweitern. Die Sensoren sind bisher in Java implementiert. Native Sensoren in C, etc. würden es ermöglichen weitere Systeme in die Tests einzubinden.

Im Bereich Android wäre die Integration mit automatischen GUI Testtools eine sinnvolle Erweiterung. So könnten Tests komplett automatisiert ohne größeren Aufwand ausgeführt werden und die Daten zum HEKATE Backend gesandt und dort ausgewertet werden.

Bei verteilten Systemen darf der Sicherheitsaspekt nicht vernachlässigt werden. Die Übertragung der Daten der Sensoren zum Backend sollte verschiedenste Mechnismen zur Verschlüsselung unterstützen.

Auch die Integration in bestehende Systeme unter der Verwendung von Complex Event Processing[TFDB08] stellt eine Möglichkeit zur Einbindung von HEKATE da.

Auch sind weitere rechtliche Aspekte zu beachten. Nicht immer ist es erlaubt Daten ausserhalb des eigentlichen Systems zu speichern. Für diesen Fall kann HEKATE erweitert werden, dass nur mehr die Validierung mit den Daten erfolgt und diese nicht mehr persistent gehalten werden.

APPENDIX A

Appendix

A.1 Velocity Template

```
{
package domain
{
  [RemoteClass(alias="$classname")]

  public class $simpleName
  {
  #foreach( $pet in $petList )

  [Bindable]
  public var $pet.name:$pet.type;
  public function get${pet.nameFirstUpper}():$pet.type { return $pet.name; }
  public function set${pet.nameFirstUpper}value(value:${pet.type}):void {
      $pet.name = value; }

  #end

  public function $simpleName()
  {
  }

  }
}
```

Listing A.1: Velocity Template

A.2 XSD Schema HEKATE Konfiguration

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="sensorConfig" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="sensorConfig" msdata:IsDataSet="true" msdata:Locale="en-US">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="generalConfig">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="backendEndpoint" type="xs:string" minOccurs="0" />
              <xs:element name="commandPoolingIntervalInSeconds" type="xs:string" minOccurs="0" />
              <xs:element name="nodeGUID" type="xs:string" minOccurs="0" />
              <xs:element name="overwriteSensorModus" type="xs:string" minOccurs="0" />
              <xs:element name="sensorModus" type="xs:string" minOccurs="0" />
              <xs:element name="systemGUID" type="xs:string" minOccurs="0" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="liveConfig">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="intervalInMinutes" type="xs:string" minOccurs="0" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="sensors">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="sensor" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="GUID" type="xs:string" minOccurs="0" />
                    <xs:element name="name" type="xs:string" minOccurs="0" />
                    <xs:element name="sensorModus" type="xs:string" minOccurs="0" />
                    <xs:element name="sensorType" type="xs:string" minOccurs="0" />
                    <xs:element name="callableMethods" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="callableMethod" minOccurs="0" maxOccurs="unbounded">
                            <xs:complexType>
                              <xs:sequence>
```

```
                                <xs:element name="implementingClass" type="xs
                                    :string" minOccurs="0" />
42                              <xs:element name="methodName" type="xs:string
                                    " minOccurs="0" />
                                <xs:element name="serviceName" type="xs:
                                    string" minOccurs="0" />
44                              <xs:element name="methodParameters" minOccurs
                                    ="0" maxOccurs="unbounded">
                                  <xs:complexType>
46                                  <xs:sequence>
                                      <xs:element name="name" type="xs:string
                                          " minOccurs="0" />
48                                    <xs:element name="position" type="xs:
                                          string" minOccurs="0" />
                                      <xs:element name="type" type="xs:string
                                          " minOccurs="0" />
50                                    <xs:element name="value" type="xs:
                                          string" minOccurs="0" />
                                    </xs:sequence>
52                                </xs:complexType>
                                </xs:element>
54                            </xs:sequence>
                            </xs:complexType>
56                        </xs:element>
                        </xs:sequence>
58                    </xs:complexType>
                    </xs:element>
60                </xs:sequence>
                </xs:complexType>
62            </xs:element>
            </xs:sequence>
64        </xs:complexType>
        </xs:element>
66    </xs:choice>
  </xs:complexType>
68 </xs:element>
</xs:schema>
```

Listing A.2: HEKATE XML Sensor Konfiguration

A.3 XSD Schema Anwendungsbeispiel Export

```
1 <?xml version="1.0" encoding="utf-8"?>
  <xs:schema id="productExportObject" xmlns="" xmlns:xs="http://www.w3.org
      /2001/XMLSchema" xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
3   <xs:element name="products">
      <xs:complexType>
5       <xs:sequence>
          <xs:element name="barCode" type="xs:string" minOccurs="0" />
7         <xs:element name="description" type="xs:string" minOccurs="0" />
          <xs:element name="name" type="xs:string" minOccurs="0" />
```

```
9         <xs:element ref="ingredients" minOccurs="0" maxOccurs="unbounded" />
          <xs:element name="productVersion" minOccurs="0" maxOccurs="unbounded"
              >
11          <xs:complexType>
              <xs:sequence>
13              <xs:element name="createDate" type="xs:string" minOccurs="0" />
                <xs:element name="lastUpdate" type="xs:string" minOccurs="0" />
15              <xs:element name="version" type="xs:string" minOccurs="0" />
              </xs:sequence>
17          </xs:complexType>
          </xs:element>
19        <xs:element ref="products" minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
21    </xs:complexType>
  </xs:element>
23  <xs:element name="ingredients">
    <xs:complexType>
25    <xs:sequence>
        <xs:element name="amount" type="xs:string" minOccurs="0" />
27      <xs:element name="substance" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
29          <xs:sequence>
              <xs:element name="description" type="xs:string" minOccurs="0"
                  />
31            <xs:element name="name" type="xs:string" minOccurs="0" />
            </xs:sequence>
33        </xs:complexType>
        </xs:element>
35      <xs:element ref="ingredients" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
37  </xs:complexType>
  </xs:element>
39  <xs:element name="productExportObject" msdata:IsDataSet="true" msdata:
      Locale="en-US">
    <xs:complexType>
41    <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="products" />
43      <xs:element ref="ingredients" />
        <xs:element name="producers">
45        <xs:complexType>
            <xs:sequence>
47            <xs:element name="producer" minOccurs="0" maxOccurs="unbounded"
                  >
                <xs:complexType>
49                <xs:sequence>
                    <xs:element name="id" type="xs:string" minOccurs="0" />
51                  <xs:element name="lastUpdate" type="xs:string" minOccurs=
                        "0" />
                    <xs:element name="name" type="xs:string" minOccurs="0" />
53                  <xs:element name="address" minOccurs="0" maxOccurs="
                        unbounded">
                      <xs:complexType>
55                      <xs:sequence>
```

```
                        <xs:element name="city" type="xs:string" minOccurs=
                            "0" />
57                      <xs:element name="country" type="xs:string"
                            minOccurs="0" />
                        <xs:element name="postal" type="xs:string"
                            minOccurs="0" />
59                      <xs:element name="street" type="xs:string"
                            minOccurs="0" />
                      </xs:sequence>
61                  </xs:complexType>
                  </xs:element>
63                <xs:element name="labels" minOccurs="0" maxOccurs="
                      unbounded">
                    <xs:complexType>
65                    <xs:sequence>
                        <xs:element name="label" minOccurs="0" maxOccurs="
                            unbounded">
67                        <xs:complexType>
                            <xs:sequence>
69                            <xs:element name="labelName" type="xs:string"
                                  minOccurs="0" />
                              <xs:element name="suitesLabel" type="xs:
                                  string" minOccurs="0" />
71                          </xs:sequence>
                          </xs:complexType>
73                      </xs:element>
                      </xs:sequence>
75                  </xs:complexType>
                  </xs:element>
77                <xs:element ref="products" minOccurs="0" maxOccurs="
                      unbounded" />
                </xs:sequence>
79            </xs:complexType>
            </xs:element>
81        </xs:sequence>
        </xs:complexType>
83    </xs:element>
    </xs:choice>
85  </xs:complexType>
</xs:element>
87</xs:schema>
```

Listing A.3: Export XSD

Literaturverzeichnis

[Ado] ADOBE SYSTEMS (Hrsg.): *Adobe Flash Player*. `http://get.adobe.com/de/flashplayer/`, Abruf: 2011-2-1

[AGM08] ABITEBOUL, Serge ; GREENSHPAN, Ohad ; MILO, Tova: Modeling the mashup space. In: *Proceeding of the 10th ACM workshop on Web information and data management*. New York, NY, USA : ACM, 2008 (WIDM '08). – ISBN 978–1–60558–260–3, 87–94

[Apa] APACHE SOFTWARE FOUNDATION (Hrsg.): *Maven - Welcome to Apache Maven*. `http://maven.apache.org/`, Abruf: 2011-04-04

[Api] APIGEE CORP. (Hrsg.): *Apigee | Free and Enterprise API Management Products and Infrastructure*. `http://apigee.com/`, Abruf: 2011-2-1

[AS08] ANNETT, Michelle ; STROULIA, Eleni: Building highly-interactive, data-intensive, REST applications: the Invenio experience. In: *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*. New York, NY, USA : ACM, 2008 (CASCON '08), 15:192–15:206

[AZW09] AL-ZOUBI, Khaldoon ; WAINER, Gabriel: Using REST Web-Services Architecture for Distributed Simulation. In: *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*. Washington, DC, USA : IEEE Computer Society, 2009 (PADS '09). – ISBN 978–0–7695–3713–9, 114–121

[BDS10] BRATANIS, Konstantinos ; DRANIDIS, Dimitris ; SIMONS, Anthony J. H.: An extensible architecture for run-time monitoring of conversational web services. In: *Proceedings of the 3rd International Workshop on Monitoring, Adaptation and Beyond*. New York, NY, USA : ACM, 2010 (MONA '10). – ISBN 978–1–4503–0422–1, 9–16

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

[BGS01] BEYDEDA, S. ; GRUH, V. ; STACHORSKI, M.: A graphical class representation for integrated black- and white-box testing, 2001, S. 706 –715

[Bir97] BIRMAN, Kenneth P.: *Building secure and reliable network applications*. Greenwich, CT, USA : Manning Publications Co., 1997. – ISBN 1–884777–29–5

[Bla67] BLANCHARD, Ben S.: Cost Effectiveness, System Effectiveness, Integrated Logistics Support, and Maintainability. In: *Reliability, IEEE Transactions on* R-16 (1967), dec., Nr. 3, S. 117 –126. http://dx.doi.org/10.1109/TR.1967.5217480. – DOI 10.1109/TR.1967.5217480. – ISSN 0018–9529

[BMN89] BRANTLEY, W. C. ; MCAULIFFE, K. P. ; NGO, T. A.: RP3 performance monitoring hardware. (1989), S. 35–47. http://dx.doi.org/http://doi.acm.org/10.1145/75705.75707. – DOI http://doi.acm.org/10.1145/75705.75707. ISBN 0–201–50390–5

[boo10] GEORGE APPLING AND GIULIO PAPPALARD (Hrsg.). BOOZ&CO: The Rise of Mobile Applications Stores: Gateways to the World of Apps / booz&co. 2010. – Forschungsbericht

[BP09] BERTOLINO, A. ; POLINI, A.: SOA Test Governance: Enabling Service Integration Testing across Organization and Technology Borders, 2009, S. 277 –286

[Cha] CHARLES GOLDFARB (Hrsg.): *SGMLUG SGML*. http://xml.coverpages.org/sgmlhist0.html, Abruf: 2010-10-17

[Coda] CODEHAUS FOUNDATION (Hrsg.): *jetty - Jetty WebServer*. http://jetty.codehaus.org/jetty/, Abruf: 2011-04-04

[Codb] CODEHOUSE (Hrsg.): *Jackson Java JSON-processor*. http://jackson.codehaus.org//, Abruf: 2011-04-04

[CP98] CHEN, T.Y. ; POON, P.L.: Teaching black box testing, 1998, S. 324 –329

[Cro06] CROCKFORD, Douglas: The application/json Media Type for JavaScript Object Notation (JSON) / IETF. 2006 (RFC 4627). – Forschungsbericht

[DG92] DEWITT, David ; GRAY, Jim: Parallel database systems: the future of high performance database systems. In: *Commun. ACM* 35 (1992), Nr. 6, S. 85–98. http://dx.doi.org/http://doi.acm.org/10.1145/129888.129894. – DOI http://doi.acm.org/10.1145/129888.129894. – ISSN 0001–0782

[DGH03] DUSTDAR, Schahram ; GALL, Harald ; HAUSWIRTH, Manfred: *Softwarearchitekturen für verteilte Systeme*. Springer, 2003

[Fie00] FIELDING, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*. Irvine, California, University of California, Irvine, Diss., 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[Flo94] FLORISSI, Patrícia Gomes S.: Quality of service management automation in integrated distributed systems. In: *CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, IBM Press, 1994, S. 18

[Gar] GARTNER (Hrsg.): *Gartner's 2009 Hype Cycle Special Report Evaluates Maturity of 1,650 Technologies*. http://www.gartner.com/it/page.jsp?id=1124212, Abruf: 2010-10-17

[gar96] GARTNER (Hrsg.). SSA RESEARCH NOTE SPA-401-068: Service Oriented' Architectures, Part 1 / SSA Research Note SPA-401-068. 1996. – Forschungsbericht

[GH99] GIGUETTE, Ray ; HASSELL, Johnette: Toward a resourceful method of software fault tolerance. In: *ACM-SE 37: Proceedings of the 37th annual Southeast regional conference (CD-ROM)*. New York, NY, USA : ACM, 1999. – ISBN 1–58113–128–3, S. 1

[Gooa] GOOGLE (Hrsg.): *Issue 314: Conversion to Dalvik format failed with error 2*. http://code.google.com/p/android/issues/detail?id=314, Abruf: 2011-04-04

[Goob] GOOGLE INC. (Hrsg.): *android.com - Android at Google I/O*. http://www.android.com/, Abruf: 2010-10-17

[Gooc] GOOGLE INC. (Hrsg.): *Google Docs*. https://docs.google.com/#all, Abruf: 2010-10-17

[Good] GOOGLE INC. (Hrsg.): *What is Android? | Android Developer Guide*. http://developer.android.com/guide/basics/what-is-android.html, Abruf: 2010-10-17

[Gro94] GROSSPIETSCH, K.E.: Fault Tolerance in Highly Parallel Hardware Systems. In: *IEEE Micro* 14 (1994), S. 60–68. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/40.259902. – DOI http://doi.ieeecomputersociety.org/10.1109/40.259902. – ISSN 0272–1732

[HE91] HEATH, M.T. ; ETHERIDGE, J.A.;: Visualizing the performance of parallel programs. In: *Software, IEEE*, IEEE Computer Society, 1991, S. 29–39

[Hen08] HENNING, Michi: The rise and fall of CORBA. In: *Commun. ACM* 51 (2008), Nr. 8, S. 52–57. – ISSN 0001–0782

[HPB10] HOANG, Dat D. ; PAIK, Hye-young ; BENATALLAH, Boualem: An analysis of spreadsheet-based services mashup. In: *Proceedings of the Twenty-First Australasian Conference on Database Technologies - Volume 104*. Darlinghurst, Australia, Australia : Australian Computer Society, Inc., 2010 (ADC '10). – ISBN 978–1–920682–85–9, 141–150

[IEE08] IEEE: IEEE Standard for Software and System Test Documentation. In: *IEEE Std 829-2008* (2008), jul., S. 1 –118

[Irm02] IRMEN DE JONG: Web Services/SOAP and CORBA. (2002)

[JAX] JAX (Hrsg.): *JAX*. `https://jaxp.dev.java.net//`, Abruf: 2010-10-17

[JDO] JDOM (Hrsg.): *JDOM*. `http://www.jdom.org/`, Abruf: 2010-10-17

[JfSJb+06] JUN-FENG, Yao ; SHI, Ying ; JU-BO, Luo ; DAN, Xie ; XIANG-YANG, Jia: Reflective Architecture Based Software Testing Management Model, 2006, S. 821 –825

[JW98] JOGALEKAR, P. ; WOODSIDE, M.: Evaluating the scalability of distributed systems, 1998, S. 524 –531 vol.7

[KT04] KHARE, Rohit ; TAYLOR, Richard N.: Extending the Representational State Transfer (REST) Architectural Style for Decentralized Systems. In: *Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2004 (ICSE '04). – ISBN 0–7695–2163–0, 428–437

[KZBA10] KONTOGIANNIS, Kostas ; ZOU, Ying ; BREALEY, Chris ; ATHANASOPOULOS, Michael: Issues and challenges leveraging REST architectural style in enterprise service systems. In: *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*. New York, NY, USA : ACM, 2010 (CASCON '10), 368–370

[Lee09a] LEE, Youngkon: SOA Test Framework Based on BPA -Simulated Event Proxy. In: *NISS '09: Proceedings of the 2009 International Conference on New Trends in Information and Service Science*. Washington, DC, USA : IEEE Computer Society, 2009. – ISBN 978–0–7695–3687–3, S. 293–298

[Lee09b] LEE, Youngkon: SOA Test Framework Based on BPA -Simulated Event Proxy, 2009, S. 293 –298

[LLB+09] LUDWIG, Heiko ; LAREDO, Jim ; BHATTACHARYA, Kamal ; PASQUALE, Liliana ; WASSERMANN, Bruno: REST-based management of loosely coupled services. In: *Proceedings of the 18th international conference on World wide web*. New York, NY, USA : ACM, 2009 (WWW '09). – ISBN 978–1–60558–487–4, 931–940

[LLZ+09] LIU, Hehui ; LI, Zhongjie ; ZHU, Jun ; TAN, Huafang ; HUANG, Heyuan: A Unified Test Framework for Continuous Integration Testing of SOA Solutions, 2009, S. 880 –887

[Lon08] LONDESBROUGH, I.: A Test Process for all Lifecycles, 2008, S. 327 –331

[Mag] MAGNOLIA AND SBS AND JBOSS EAP AND RHEL (Hrsg.): *Hibernate - JBoss Community*. `http://www.hibernate.org/`, Abruf: 2011-04-04

[Mas95] MASOUD MANSOURI-SAMANI,: Monitoring of Distributed Systems, Dissertation / University of London. 1995. – Forschungsbericht

[Mil] MILOSLAV NIC AND JIRI JIRAT (Hrsg.): *XPath Tutorial*. `http://zvon.org/xxl/XPathTutorial/General/examples.html`, Abruf: 2010-10-17

[Mül09] MÜLLER, Florian: *Professionelle Rich-Client-Lösungen mit Flex und Java*. Addison-Wesely, 2009

[MMVA02] MALAMOS, A. G. ; MALAMAS, E. N. ; VARVARIGOU, T. A. ; AHUJA, S. R.: A Model for Availability of Quality of Service in Distributed Multimedia Systems. In: *Multimedia Tools Appl.* 16 (2002), Nr. 3, S. 207–230. `http://dx.doi.org/http://dx.doi.org/10.1023/A:1013956019587`. – DOI http://dx.doi.org/10.1023/A:1013956019587. – ISSN 1380–7501

[MNSB06] MARTENS, Wim ; NEVEN, Frank ; SCHWENTICK, Thomas ; BEX, Geert J.: Expressiveness and complexity of XML Schema. In: *ACM Trans. Database Syst.* 31 (2006), Nr. 3, S. 770–813. – ISSN 0362–5915

[MSN06] MARKIEWICZ, R. ; SAKOWICZ, B. ; NAPIERALSKI, A.: Distributed Reporting System Based on Java 2 Micro Edition, 2006, S. 456 –459

[Nel90] NELSON, V.P.: Fault-tolerant computing: fundamental concepts. In: *Computer* 23 (1990), jul., Nr. 7, S. 19 –25. `http://dx.doi.org/10.1109/2.56849`. – DOI 10.1109/2.56849. – ISSN 0018–9162

[NRH10] NASEER, F. ; REHMAN, S. ur ; HUSSAIN, K.: Using meta-data technique for component based black box testing. In: *Emerging Technologies (ICET), 2010 6th International Conference on*, 2010, S. 276 –281

[Oraa] ORACLE (Hrsg.): *Java Message Service (JMS)*. `http://www.oracle.com/technetwork/java/jms/index.html`, Abruf: 2011-2-1

[Orab] ORACLE (Hrsg.): *Java Metadata Interface (JMI)*. `http://java.sun.com/products/jmi/index.jsp`, Abruf: 2010-10-17

[Orac] ORACLE CORPORATION (Hrsg.): *MySQL :: The world's most popular open source database*. `http://www.mysql.com/`, Abruf: 2011-04-04

[PB96] POMBERGER, Gustav ; BLASCHEK, Günther: *Grundlagen des Software Engineering - Prototyping und objektorientierte Software-Entwicklung*. Hanser Fachbuchverlag, 1996

[PWM10] PAUTASSO, Cesare ; WILDE, Erik ; MARINOS, Alexandros: First International Workshop on RESTful Design (WS-REST 2010). In: *Proceedings of the First International Workshop on RESTful Design*. New York, NY, USA : ACM, 2010 (WS-REST '10). – ISBN 978–1–60558–959–6, 1–3

[Ral04] RALSTON, A.: Four editions and eight publishers: a history of the encyclopedia of computer science. In: *Annals of the History of Computing, IEEE* 26 (2004), jan., Nr. 1, S. 42 – 52. http://dx.doi.org/10.1109/MAHC.2004.1278849. – DOI 10.1109/MAHC.2004.1278849. – ISSN 1058–6180

[Ref] REFSNES DATA (Hrsg.): *Introduction to XML Schema*. http://www.w3schools.com/schema/schema_intro.asp, Abruf: 2010-10-17

[RKH88] RAGHAVENDRA, C.S. ; KUMAR, V.K.P. ; HARIRI, S.: Reliability analysis in distributed systems. In: *Computers, IEEE Transactions on* 37 (1988), mar., Nr. 3, S. 352 –358. http://dx.doi.org/10.1109/12.2173. – DOI 10.1109/12.2173. – ISSN 0018–9340

[Rud10] RUDOLF RAMLER AND DIETMAR WINKLER: Vorlesungsfolien: Software Testing: Definitions and Bascis / Vienna University of Technology, Institute of Software Technology and Interactive Systems. 2010. – Forschungsbericht

[SAX] SAX (Hrsg.): *SAX*. http://www.saxproject.org, Abruf: 2010-10-17

[Sch95] SCHROEDER, B.A.: On-line monitoring: a tutorial. In: *Computer*, IEEE Computer Society, 1995. – ISBN 0018–9162, S. 72–78

[SGM07] SETH, Anita ; GUPTA, Hari M. ; MOMAYA, Kirankumar: Quality of service parameters in cellular mobile communication. In: *Int. J. Mob. Commun.* 5 (2007), Nr. 1, S. 68–93. http://dx.doi.org/http://dx.doi.org/10.1504/IJMC.2007.011490. – DOI http://dx.doi.org/10.1504/IJMC.2007.011490. – ISSN 1470–949X

[Sim] SIMPLE XML (Hrsg.): *Simple XML 2.5.2*. http://simple.sourceforge.net/, Abruf: 2011-04-04

[soa07] PHIL BIANCO AND RICK KOTERMANSKI AND PAULO MERSON (Hrsg.). SOFTWARE ENGINEERING INSTITUTE: Evaluating a Service-Oriented Architecture / Software Engineering Institute. 2007. – Forschungsbericht

[Son] SONATYPE, INC. (Hrsg.): *Maven By Example*. http://www.sonatype.com/books/mvnex-book/reference/preface-copyright.html, Abruf: 2011-07-04

[spra] SPRINGSOURCE.ORG (Hrsg.): *About Spring*. http://www.springsource.org/about, Abruf: 2011-04-04

[sprb] SPRINGSOURCE.ORG (Hrsg.): *Developing a Spring Framework MVC application step-by-step*. http://static.springsource.org/docs/Spring-MVC-step-by-step/, Abruf: 2011-04-04

[sprc] SPRINGSOURCE.ORG (Hrsg.): *Spring Projects*. http://www.springsource.org/projects, Abruf: 2011-05-05

[sprd] SPRINGSOURCE.ORG (Hrsg.): *springsource.org*. `http://www.springsource.org`, Abruf: 2011-04-04

[SUN] SUN (Hrsg.): *SOA Benefits Overview*. `http://www.sun.com/products/soa/benefits.jsp`, Abruf: 2010-10-17

[TFDB08] TAHER, Yehia ; FAUVET, Marie-Christine ; DUMAS, Marlon ; BENSLIMANE, Djamal: Using CEP technology to adapt messages exchanged by web services. In: *Proceeding of the 17th international conference on World Wide Web*. New York, NY, USA : ACM, 2008 (WWW '08). – ISBN 978–1–60558–085–2, 1231–1232

[UNE] UNECE (Hrsg.): *United Nations Directories for Electronic Data Interchange for Administration, Commerce and Transport*. `http://www.unece.org/trade/untdid/welcome.htm`, Abruf: 2010-10-17

[VKBG94] VOGEL, Andreas ; KERHERVÉ, Brigitte ; BOCHMANN, Gregor v. ; GECSEI, Jan: Distributed multimedia applications and quality of service: a survey. In: *CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, IBM Press, 1994, S. 71

[VKZ05] VÖLTER, Markus ; KIRCHER, Michael ; ZDUN, Uwe: *Remoting Patterns*. WILEY, 2005

[W3Ca] W3C (Hrsg.): *Document Object Model (DOM) Level 3 Core Specification*. `http://www.w3.org/TR/DOM-Level-3-Core/`, Abruf: 2010-10-17

[W3Cb] W3C (Hrsg.): *The Document Object Model(DOM)*. `http://www.w3.org/2002/07/26-dom-article.html`, Abruf: 2010-10-17

[W3Cc] W3C (Hrsg.): *XHTML 2.0*. `http://www.w3.org/TR/xhtml2/`, Abruf: 2011-03-03

[W3Cd] W3C (Hrsg.): *XML Development History*. `http://www.w3.org/XML/hist2002`, Abruf: 2010-10-17

[W3Ce] W3C (Hrsg.): *XML Path Language (XPath)*. `http://www.w3.org/TR/xpath`, Abruf: 2009-07-10

[W3Cf] W3C CONSORTIUM (Hrsg.): *W3C Input to the United Nations Working Group on Internet Governance*. `http://www.w3.org/2005/04/dd-wsig.html`, Abruf: 2010-10-17

[W3Cg] W3C SCHOOL (Hrsg.): *XPath Tutorial*. `http://www.w3schools.com/xpath/`, Abruf: 2010-10-17

[Wer00] WERNER KURSCHL,: Monitoring von verteilten Systemen / Johannes Kepler Universität. 2000. – Forschungsbericht

[WPG+10] WU, Jiyi ; PING, Lingdi ; GE, Xiaoping ; WANG, Ya ; FU, Jianqing: Cloud Storage as the Infrastructure of Cloud Computing. In: *Intelligent Computing and Cognitive Informatics (ICICCI), 2010 International Conference on*, 2010, S. 380 –383

[Yam98] YAMAURA, T.: How to design practical test cases. In: *Software, IEEE* 15 (1998), nov., Nr. 6, S. 30 –36. http://dx.doi.org/10.1109/52.730835. – DOI 10.1109/52.730835. – ISSN 0740–7459

[Yan08] YAN, N.: Make the Consumable Services via REST. In: *Web Services, 2008. ICWS '08. IEEE International Conference on*, 2008, S. 12

[YJC08] YOON, Hoijin ; JI, Eun M. ; CHOI, Byoungju: Generating Test Requirements for the Service Connections based on the Layers of SOA, 2008, S. 348 –355

[ZfXp09] ZHI-FANG, Liu ; XIAO-PENG, Gao: SOA Based Mobile Device Test, 2009, S. 641 –644

[ZQ06] ZHANG, Jia ; QIU, R.G.: Fault Injection-based Test Case Generation for SOA-oriented Software, 2006, S. 1070 –1078

[ZZS+10] ZHANG, Hongguang ; ZHAO, Zhenzhen ; SIVASOTHY, Shanmugalingam ; HUANG, Cuiting ; CRESPI, Noël: Quality-assured and sociality-enriched multimedia mobile mashup. In: *EURASIP J. Wirel. Commun. Netw.* 2010 (2010), January, 11:1–11:13. http://dx.doi.org/http://dx.doi.org/10.1155/2010/721312. – DOI http://dx.doi.org/10.1155/2010/721312. – ISSN 1687–1472