Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Daniel Löffelholz

Selbst-adaptive Mechanismen für die autonome Rekonfiguration und Optimierung mobiler Anwendungen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Daniel Löffelholz

Selbst-adaptive Mechanismen für die autonome Rekonfiguration und Optimierung mobiler Anwendungen

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Master Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Abgegeben am 28. Juni 2011

Daniel Löffelholz

Thema der Masterarbeit

Selbst-adaptive Mechanismen für die autonome Rekonfiguration und Optimierung mobiler Anwendungen

Stichworte

selbst-adaptive System selbst-optimierend selbst-konfigurierend mobil Android OSGi

Kurzzusammenfassung

Moderne Anwendungen auf ultraportablen Geräten als ständige Begleiter, sehen sich den unterschiedlichsten Voraussetzungen und Gegebenheiten der Umgebung und des eigenen Geräts ausgesetzt. So muss ein Entwickler einer mobilen Anwendung beispielsweise von unregelmäßigen und oft auftretenden Änderungen von Stromversorgung oder Internetanbindung ausgehen. Anwendungen, die ihr Verhalten nicht anpassen können, sondern in jeder Umgebung auf jedem Gerät gleich agieren, können die Erwartungen an moderne mobile Anwendungen deshalb nicht erfüllen. Die dynamische Anpassungsfähigkeit beziehungsweise Adaptivität ist demnach eine neue Herausforderung, hervorgehend aus der universalen Nutzung von mobilen Geräten, die es zu meistern gilt. Die vorliegende Arbeit versucht aus der Vielzahl adaptiver Mechanismen eine geeignete Zusammenstellung zu identifizieren, die den Charakter einer reflexiven mobilen Anwendung optimal unterstützt. Mit Hilfe dieser Mechanismen wird im Rahmen dieser Arbeit ein Konzept entwickelt werden, welches nicht nur die Rekonfiguration ermöglicht, sondern auch die autonome Rekonfigurationsentscheidung trifft. Dies geschieht auf Basis von Meta-Daten die die Qualitätseigenschaften der eingesetzten Softwarekomponenten beschreiben, und Erfassung des Umgebungs- und Gerätezustandes, sowie unter Einbeziehung der Kosten und Häufigkeit der Rekonfigurationen. Die sich je nach Kontext ändernden Benutzeranforderungen werden ebenso mit einbezogen. Dieses Konzept wird am Ende der Arbeit in einem Software-Entwurf umgesetzt, welcher auf der Android-Plattform unter Verwendung eines OSGi-Frameworks die definierten Anforderungen erfüllen soll. Der Autor demonstriert damit die Umsetzbarkeit des Konzepts, und dass sich die ausgewählten Technologien für diesen Zweck gut eignen. Zudem wird mit Hilfe einer Simulationsumgebung systematisch geprüft, ob die Anwendung die gestellten Anforderungen erfüllen kann. Ein Fazit und der Ausblick auf mögliche weitere Schritte und Ansätze bilden den Schluss dieser Arbeit.

Title of the paper

Self adaptive mechanisms for the autonomic reconfiguration and optimization of mobile applications

Keywords

self adaptive system self-optimizing self-configuring mobile Android OSGi

Abstract

Modern applications, running on ultra portable devices as permanent accompanists, see themselves constantly exposed to many different conditions of their environment and their own device. For example, a developer of a mobile application must assume that the power supply or internet connection often changes at runtime. Applications, which can not adapt their behavior, but act identical in every environment and on every device, can not fulfil the expectations on modern mobile applications. The ability to dynamically adapt is a new challenge, resulting from the universal use of mobile devices, which shall be mastered. These thesis aims to identify a appropriate combination of adaptive mechanism and techniques, which support the character of an reflexive mobile application. In the scope of this thesis, a concept is developed that combines those mechanisms for reconfiguration of the applications components, and a decision-mechanism if the reconfiguration should be done at all. This decision is made based on meta data, which describe the quality properties of the components, and on the capturing of the state of the environment and device. Additionally, the changin user requirements, and the costs and frequency of the reconfiguration get also comprised to that decision. The concept goes from theory to practice with an first application draft, running on the android plattform with the usage of an OSGi-framework. The draft is the proof of concept, and demonstrates how the concept can be implemented into a running application. It also demonstrates that the chosen technology is well applicable for that kind of challenge. Using a simulation environment, it will be shown that the application can fulfill the definied requirements. A conclusion and the outlook on further work form the closing of this thesis.

Inhaltsverzeichnis

1 Einleitung **10**
- 1.1 Motivation und Problemstellung . . . 10
- 1.2 Ziele . . . 11
- 1.3 Abgrenzung . . . 11
- 1.4 Gliederung . . . 12

2 Fachliche Grundlagen **13**
- 2.1 Adaption . . . 13
  - 2.1.1 Selbst-adaptive Systeme . . . 14
  - 2.1.2 Self-*-Eigenschaften . . . 15
  - 2.1.3 Autonomes reflexives Verhalten . . . 16
  - 2.1.4 Kontext . . . 17
  - 2.1.5 Klassifizierung . . . 19
    - 2.1.5.1 Adaptionsebene . . . 20
    - 2.1.5.2 Adaptionsentscheidung . . . 20
    - 2.1.5.3 Auswirkungsvorhersage . . . 21
    - 2.1.5.4 Adaptionszeitpunkt . . . 22
- 2.2 Komponentenbasierte Entwicklung . . . 22
  - 2.2.1 Komponente . . . 23
  - 2.2.2 Komponentenmodell . . . 24
- 2.3 Quality of Service für Komponenten . . . 25
  - 2.3.1 Definitionen und Terminologie . . . 26
  - 2.3.2 Zusammenhang Qualität und selbst-adaptive Systeme . . . 27
  - 2.3.3 Benutzeranforderungen an und Eigenschaften von mobilen Anwendungen . . . 27
  - 2.3.4 Metriken und Messbarkeit . . . 28

3 Technologische Grundlagen **29**
- 3.1 Auswahl des Komponentenmodells und der mobilen Plattform . . . 29
- 3.2 OSGi . . . 30

3.2.1 Bundle . . . 31
3.2.2 Bundle Lifecycle . . . 32
3.3 Android . . . 33
3.3.1 Android-Komponenten . . . 34
3.3.1.1 Activity . . . 35
3.3.1.2 Service . . . 35
3.3.1.3 Content Provider . . . 35
3.3.1.4 Broadcast Receiver . . . 35
3.3.2 Activity Stack . . . 36

4 Verwandte Arbeiten **37**
4.1 Selbst-adaptive Systeme . . . 37
4.2 Quality of Service und Komponenten . . . 40
4.3 Einordnung dieser Arbeit . . . 41

5 Konzept **45**
5.1 Vorgehen . . . 45
5.1.1 Szenario . . . 45
5.1.2 Anwendungsfälle . . . 47
5.1.3 Anwendungstyp . . . 50
5.1.4 Optionale Anforderungen . . . 51
5.2 Architektur . . . 52
5.2.1 Self-*-Eigenschaften und Qualitätseigenschaften . . . 52
5.2.2 Technologisches Profil und Architekturentscheidungen . . . 52
5.2.3 Szenariomodell . . . 54
5.2.3.1 Umgebungskontext . . . 55
5.2.3.2 Ressoucenkontext . . . 55
5.2.3.3 Benutzerkontext . . . 56
5.2.3.4 Qualitätseigenschaften der Komponenten . . . 58
5.2.4 Kontrollfluss . . . 59
5.2.4.1 *collect* . . . 60
5.2.4.2 *analyze* . . . 61
5.2.4.3 *decide* . . . 61
5.2.4.4 *act* . . . 65
5.3 Zusammenfassung Konzept . . . 66

6 Entwurf **69**
6.1 Architektur . . . 69
6.2 Realisierung . . . 71
6.2.1 Mobile Plattform . . . 71

6.2.2 Komponentenframework . . . 72
6.2.3 Qualitätsattribute und Modelle . . . 73
6.2.4 Ablauf . . . 74
6.2.4.1 Inbetriebnahme . . . 74
6.2.4.2 Zyklus . . . 75
6.3 Fazit . . . 76

7 Simulation und Verifikation **78**
7.1 Aufbau der Simulationsumgebung . . . 78
7.2 Versuche . . . 79
7.2.1 Erster Versuchsaufbau: Schnelle Rekonfiguration . . . 80
7.2.2 Zweiter Versuchsaufbau: Langsame Rekonfiguration . . . 83
7.2.3 Dritter Versuchsaufbau: Unvorhergesehene Szenarien . . . 84
7.3 Auswertung . . . 86
7.3.1 Erster Versuchsaufbau . . . 86
7.3.2 Zweiter Versuchsaufbau . . . 86
7.3.3 Dritter Versuchsaufbau . . . 87
7.4 Fazit . . . 87

8 Schluss **89**
8.1 Zusammenfassung . . . 89
8.2 Fazit . . . 90
8.3 Ausblick . . . 91

9 Anhang **92**
A Deployment-Befehle . . . 92
B Klassendiagramm ROSt . . . 93
C Berechnung der Kontextabweichungen . . . 94
D Berechnung ob die Rekonfiguration lohnenswert ist . . . 96
E Inhalt der CD-ROM . . . 99

Tabellenverzeichnis

4.1 Zusammenfassung: verwandte Arbeiten - Teil 1 42
4.2 Zusammenfassung: verwandte Arbeiten - Teil 2 43

5.1 Anwendungfall: Fachliches Szenario 1 „Werft“ 47
5.2 Anwendungfall: Fachliches Szenario 2 „Hotel“ 48
5.3 Anwendungfall: Fachliches Szenario 3 „Büro“ 48
5.4 Anwendungfall: Allgemeines Szenario A „Geringer Batteriestand“ 48
5.5 Anwendungfall: Allgemeines Szenario B „Keine Internetverbindung“ 49
5.6 Anwendungfall: Allgemeines Szenario C „System ausgelastet“ 49
5.7 Nicht-funktionale Anforderungen an das Konzept 51
5.8 $Kontext_{Umgebung}(Umgebung_{Inet})$ 55
5.9 $Kontext_{Ressourcen}(Ressourcen_{Strom}, Ressourcen_{Last})$ 56
5.10 $Kontext_{Benutzer}(Benutzer_{Zeit}, Benutzer_{Verb}, Benutzer_{Richtigk})$ 58
5.11 $Qual_{Komponenten}(...)$ 59
5.12 Resultierende adaptive Mechanismen 67
5.13 Szenariomodell und Kontexte 68

7.1 $Kontext_{Benutzer}(Benutzer_{Zeit}, Benutzer_{Verb}, Benutzer_{Richtigk})$ 79

Abbildungsverzeichnis

2.1 Hierachie der Self-*-Eigenschaften - Salehie und Tahvildari (2009) 16
2.2 Autonomic Control Loop - Dobson u. a. (2006) 17
2.3 Hierachische Klassifizierung nach den 5 Ws 19
2.4 ISO 9126 ISO/IEC (2001) 27

3.1 OSGi Schichten OSGi-Alliance (2011) 31
3.2 Bundle-Registrierung (3.2,Wütherich u. a. (2008)) 32
3.3 Bundle Lifecycle (Wütherich u. a. (2008)) 33
3.4 Android-Architektur Becker u. a. (2010) 34

5.1 Herleitung der Architektur 46
5.2 Konzept - Regelkreis 60
5.3 Konzept - *collect* 60
5.4 Konzept - *analyze* 61
5.5 Konzept - *decide* 62
5.6 Konzept - *act* 65

6.1 Komponenten der Anwendung 70
6.2 `RostBundle` Komponenten 71
6.3 `QualityBundle` Klassendiagramm 73
6.4 Sequenzdiagramm ROSt 77

7.1 Aufbau Simulationsumgebung 79

B.1 ROSt-Bundle Klassendiagramm 93

Kapitel 1

Einleitung

Dieses Kapitel widmet sich zunächst der dieser Masterarbeit zugrunde liegenden Motivation und Problemstellung. Dadurch soll der Leser einen Eindruck über die Notwendigkeit und Herausforderungen der dynamischen Anpassung von mobilen Anwendungen gewinnen. Anschließend werden die Ziele aufgeführt und erläutert, die in dieser Arbeit erreicht werden sollen. Um den Fokus zu verdeutlichen wird die Arbeit in Absatz 1.3 von anderen Problemstellungen und Anforderungen abgegrenzt. Abschließend folgt der Ausblick auf die kommenden Kapitel.

1.1 Motivation und Problemstellung

Anwendungen befanden sich traditionell auf im Betrieb stationären Computern. Der Datenaustausch fand über fest installierte Leitungen statt, und die einzige Ressource war letztlich die Leistungsfähigkeit der technologischen Ausstattung. Diese grundlegenden Eigenschaften haben sich in den letzten Jahren durch den technologischen Fortschritt bei elektronischen Kleingeräten wie Mobiltelefonen verändert. Der Entwickler kann nicht mehr von konstanten Voraussetzungen ausgehen, auf die man sich im gesamten Lebenszyklus der Anwendung verlassen kann. Moderne Anwendungen auf ultraportablen Geräten als ständige Begleiter, sehen sich den unterschiedlichsten Voraussetzungen und Gegebenheiten bezüglich Geräte- und Umgebungsressourcen ausgesetzt. So muss ein Entwickler von unregelmäßigen und oft auftretenden Änderungen von Stromversorgung oder Internetanbindung ausgehen. Mobiltelefone unterstützen mit ihren Anwendungen schon heute eine Vielzahl von tagtäglichen Aufgaben. Dabei erwartet der Benutzer, trotz vieler laufender Applikationen, in jeder Situation eine angebrachte und zügige Funktionsweise. Natürlich ist beispielsweise damit zu rechnen, dass in der U-Bahn aufgrund des schlechten Empfangs Kompromisse bei der Internetqualität einzugehen sind. Dennoch sollte das Gerät angemessen bedienbar bleiben. Die Anwendungen müssen also jederzeit eine adäquate Dienstgüte bieten können. Anwendungen, die ihr Verhalten nicht anpassen können, sondern in jeder Umgebung auf je-

dem Gerät gleich agieren, können die Erwartungen an moderne mobile Anwendungen nicht erfüllen. Die dynamische Anpassungsfähigkeit beziehungsweise Adaptivität ist demnach eine neue Herausforderung, hervorgehend aus der universalen Nutzung von mobilen Geräten, die es zu meistern gilt.

1.2 Ziele

Ziel dieser Masterarbeit ist die Fragestellung beantworten zu können, wie und welche adaptiven Mechanismen verwendet werden können, um das Verhalten einer mobilen Anwendung an die aktuelle Umgebung und die vorhandenen Ressourcen anzupassen. Die sich durch die Umstände gegebenenfalls geänderten Benutzeranforderungen müssen dabei mit berücksichtigt werden. Um die geeigneten Mechanismen auswählen zu können, müssen diese zunächst identifiziert werden. So hat der erste Teil dieser Arbeit die Aufgabe, dem Leser einen Überblick über die unterschiedlichen Arten adaptiver Mechanismen und deren Eigenschaften zu bieten, wie auch über alle weiteren technologischen und fachlichen Themen die im Laufe der Arbeit für das Verständnis benötigt werden. Ziel des Konzepts ist es, eine Auswahl von Mechanismen und Technologien zusammenzustellen, die geeignet ist die Problemstellung zu bewältigen. Anschließend gilt zu demonstrieren, wie diese Mechanismen in ein Anwendungskonzept integriert werden können, welches für ein exemplarisches Szenario, die sich aus der Motivation abzuleitenden Anforderungen erfüllt. Ein Entwurf dieses Szenarios auf einer konkreten mobilen Plattform soll die Umsetzbarkeit demonstrieren. Zuletzt soll validiert werden, ob und wie gut man mit dem Konzept beziehungsweise dem Entwurf die Herausforderungen lösen kann.

1.3 Abgrenzung

Dieses Thema tangiert viele technologische und fachliche Fragen und Bereiche, die nicht alle im Rahmen dieser Masterarbeit beantwortet oder bearbeitet werden können. Zentral ist im wesentlichen die Frage nach der Auswahl und Umsetzung der Mechanismen, und ob der Einsatz eines solchen Konzepts sinnvoll ist. Die Art und Weise der technologischen Umsetzung in den Entwurf steht dabei nicht im Vordergrund und hat auch nicht den Anspruch eine in der Praxis einsetzbare Anwendung zu sein, sondern dient lediglich als Erweis der Umsetzbarkeit. Um das Konzept tatsächlich in der Realität einsetzen zu können, werden an vielen Stellen noch Optimierungen und Erweiterungen nötig sein, die jedoch die eigentliche Idee und Machbarkeit nicht tangieren.

1.4 Gliederung

Folgend eine Übersicht über die einzelnen Kapitel dieser Arbeit.

Kapitel 2 und 3 schildern die fachlichen Grundlagen aus den Bereichen adaptive Software, komponentenbasierte Entwicklung und Qualitätanforderungen, sowie die Grundlagen zu den im Entwurf verwendeten Technologien.

In Kapitel 4 wird dem Leser ein Überblick über verwandte Arbeiten aus unterschiedlichen Themengebieten geboten, und wie die vorliegende Arbeit eingeordnet werden kann.

Kapitel 5 stellt ein Konzept vor, welches die oben geschilderten Anforderungen, die Anpassbarkeit von mobilen Anwendungen an Umgebung, Ressourcen, und Benutzeranforderungen, erfüllen soll.

Kapitel 6 demonstriert anhand eines Anwendungsentwurfs die Umsetzbarkeit des in Kapitel 5 vorgestellten Konzepts. Zudem dient der Entwurf als erster Anhaltspunkt um den tatsächlichen Nutzen des Konzepts bewerten zu können.

Um die definierten Anforderungen systematisch überprüfen zu können, wird die Anwendung in Kapitel 7 in eine Simulationsumgebung eingebettet, und die gewonnen Ergebnisse ausgewertet.

In Kapitel 8 fasst der Autor die Arbeit mit ihren Herausforderungen und Ergebnissen kurz zusammen. Anschließend werden die gewonnen Erkenntnisse diskutiert und bewertet, so wie ein Ausblick auf die weitere Arbeit gegeben.

Kapitel 2

Fachliche Grundlagen

Eine Definition der in der Arbeit verwendeten Fachbegriffe, Methoden, und Technologien ist notwendig, um die Verständlichkeit der Ausführungen zu gewährleisten. Somit steht die Erläuterung der Grundlagen am Anfang. In der vorangegangenen Motivation geht es um eine Anwendung für mobile Geräte, die ihr Verhalten an sich ändernde Zustände anpassen kann. Insbesondere an Zustände unter denen die erwartete Dienstgüte nicht mehr gewährleistet werden kann. Durch die Anpassung soll in jedem Kontext ein für den Benutzer optimales Anwendungsverhalten erreicht werden. Aufgrund dessen gilt der erste Abschnitt der Grundlagen den Möglichkeiten der Anpassung von Software zur Laufzeit.

2.1 Adaption

Die Anpassung oder auch Adaption [1] bedeutet in der Informatik, wie auch in den meisten anderen Domänen eine Reaktion auf eine veränderte Umgebung. Dies kann die Fähigkeit von Personen oder Gesellschaften sein, je nach Umständen alternativ zu reagieren, oder auch in der Biologie die Reaktion von Zellen auf veränderte Umweltbedingungen. Ziel dabei ist, entsprechend den aktuellen Umständen (dem sogenannten *context*), ein angemesseneres Verhalten zu erreichen. Somit teilen sich adaptive Systeme auch in der Informatik zunächst nur zwei Gemeinsamkeiten: (1) eine Anpassung zur Laufzeit des Systems (2) aufgrund geänderter Umstände. Frühe Beispiele adaptiver Systeme entstammen vor allem technischen Domänen, beispielsweise in Form der adaptiven Regelung in der Regelungstechnik. Dabei wird ein adaptiver Regler eingesetzt, um eine bessere Anpassung an ein sich veränderndes Streckenverhalten zu erreichen. (vgl. Sastry und Bodson (1989))

Eine der frühesten wissenschaftlichen Erwähnungen im Softwarebereich findet sich in

[1] lateinisch: adaptare = anpassen

Gouda und Herman (1991). Neben der grundlegenden Definition[2] stellen sie fest, dass einer der häufigsten Gründe für die Änderung des Programmverhaltens das Performanzkriterium ist: die Leistungsfähigkeit einer Verhaltensweise in einer bestimmten Umgebung ist denen anderer Verhaltensweisen überlegen. Jedoch ist auch logische Korrektheit ein mögliches Kriterium: nur eine Verhaltensweise ist logisch korrekt in einer bestimmten Umgebung. Eine Untergruppe der adaptiven Systeme bildet die Gruppe der selbst-adaptiven Systeme.

2.1.1 Selbst-adaptive Systeme

Der Präfix „Selbst“ weist bei selbst-adaptiven Systemen darauf hin, dass das System autonom entscheiden kann, wie es sich umorganisieren oder verändern soll, um Veränderungen des Kontextes oder der Umgebung ausgleichen zu können. Dabei bezieht es auch den eigenen Zustand (genannt *self*) mit ein (Brun u. a. (2009)). Selbst-adaptive Systeme unterscheiden sich somit vom Rest der adaptiven Systeme durch ihre Reflexivität und autonomes Handeln. Cheng u. a. (2009) definieren hier als typische Gemeinsamkeiten, dass (1) klassische Entwurfsentscheidungen erst zur Laufzeit getroffen werden, und (2) dass das System den eigenen Zustand und die Umgebung berücksichtigt, welches oft als reflexives Verhalten bezeichnet wird. Dies bedeutet nicht, dass selbst-adaptive Systeme per se versuchen ein angebrachtes Verhalten zu entwickeln. Es gibt zwar Systeme, die völlig ohne externe Einwirkung auskommen und eigenständig Anpassungswege entwickeln, jedoch sind in den meisten Systemen Richtlinien in Form von Zielen oder Regeln hinterlegt. Diese Ziele und Regeln liegen bereits zur Entwicklungszeit der Anwendung oder in nachgereichten Teilen der Anwendung vor. Somit sind etwaige Veränderungen die eine Adaption auslösen vorher bekannt und über den Quellcode abgedeckt. Dieses *Adaptionsverhalten* nennt man auch *antizipierte Adaption*, sprich alle potenziellen Varianten sind a priori bekannt. Ist dies nicht gegeben, sprich man von der *nicht-antizipierten Adaption*. (vgl. Geihs (2007)). Trotz des Unterschieds werden die Begriffe des selbst-adaptiven Systems mit dem des adaptiven Systems manchmal synonym verwendet. Selbst-adaptive Systeme sind stark mit anderen Arten von Systemen, wie autonomen und selbst-verwaltenden Systemen (Kephart und Chess (2003a)), verwandt. Aufgrund dessen benutzen einige Forscher auch die Begriffe selbst-adaptiv, autonom und selbst-verwaltend synonym. (vgl. Huebscher und McCann (2008)). Das Entscheidende bei selbst-adaptiven Systemen ist, dass der Lebenszyklus nicht nach der Entwicklung und dem initialen Starten gestoppt wird. Der Lebenszyklus wird in Form eines geschlossenen Kreises (*closed loop*) zur Laufzeit fortgesetzt, um auf Veränderungen angemessen reagieren zu können. Dieser geschlossene Kreis wird in Abschnitt 2.1.3 genauer vorgestellt.

[2] An adaptive program is one that changes its behavior according to its environment Gouda und Herman (1991)

2.1.2 Self-*-Eigenschaften

Die adaptiven Eigenschaften eines Systems werden oft als *Self-**-Eigenschaften bezeichnet. Die ursprünglich von IBM veröffentlichte Liste beinhaltet acht unterschiedliche Eigenschaften, welche folgend vorgestellt werden sollen (IBM-Research (2011)). Salehie und Tahvildari (2009) ordnen diese dabei in einer hierarchischen Form an (siehe auch Abbildung 2.1).

General Level
Hierunter fallen alle globalen Eigenschaften selbst-adaptiver Software. Zusätzlich umfasst diese Kategorie Untergruppierungen wie *self-managing*, *self-governing*, *self-maintenance* (Kephart und Chess (2003b)) oder *self-control* (Kokar u. a. (1999))

Major Level
IBM definiert vier Eigenschaften für dieses Level (Kephart und Chess (2003a)). Dabei wurde die Klassifizierung auf Basis von biologischen selbst-adaptiven Mechanismen erstellt, da sich beispielsweise der menschliche Körper an die unterschiedlichsten Kontexte (*context*) (z.B. Temperaturwechsel der Umgebung) oder sich selbst betreffende Ereignisse (*self*) (z.B. eine Verletzung) einstellen kann. (vgl. Kephart und Chess (2003b))

- *self-configuring* ist Fähigkeit eines Systems sich als Anwort auf *self*- oder *context*-Änderungen automatisch selbst zu rekonfigurieren.
- *self-healing* ist die Fähigkeit Störungen zu erkennen, zu diagnostizieren, und darauf zu reagieren. Ein solches System muss demnach auch *self-diagnosing*- und *self-reparing*-Eigenschaften (Robertson und Williams (2006)) haben.
- *self-optimizing* beschreibt die Fähigkeit Ressourcen und Performanz zu optimieren, um den Anforderungen des Benutzers gerecht zu werden.
- *self-protecting* beschreibt die Fähigkeit Sicherheitslücken oder -brüche zu erkennen und angemessen zu reagieren. Auch der vorherige Schutz des Systems vor schädlichen Angriffen zählt dazu.

Primitive Level
Den oben genannten Eigenschaften liegen letztlich zwei Eigenschaften zugrunde:

- *self-awareness* bedeutet, dass das System seinen *self*-Zustand kennt, sowie Informationen über sein Verhalten hat. Deshalb genügt es auch den Ansprüchen des *self-monitoring*, welches nötig ist, um Informationen über den eigenen Zustand zu erlangen.

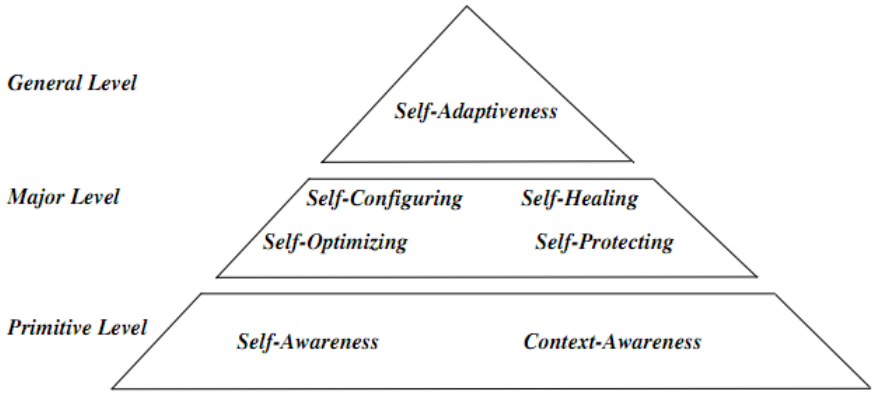

Abbildung 2.1: Hierachie der Self-*-Eigenschaften - Salehie und Tahvildari (2009)

- *context-awareness* bedeutet, dass das System Informationen über den Umgebungszustand, seinen *context*, hat.

Es gibt unterschiedliche Möglichkeiten, wie die Adaption programmatisch realisiert werden kann, welche in folgendem Abschnitt vorgestellt werden.

2.1.3 Autonomes reflexives Verhalten

Autonomes reflexives Verhalten beinhaltet mehrere Fähigkeiten, die auch in einem selbstadaptiven System vorhanden sein müssen. Diese Fähigkeiten verküpft Dobson u. a. (2006) zu einem *Closed Control Loop*[3]. Somit entsteht ein sogenannter *Autonomic Control Loop* (vgl. Dobson u. a. (2006)). Das in Abbildung 2.2 dargestellte Modell ist ein Weiterentwicklung des von der AI-Gemeinschaft vorgeschlagenen Sense-Act-Plan-Ansatz der frühen 1980er Jahre um autonome Roboter zu kontrollieren. (Brun u. a. (2009)).

Der Zyklus startet mit dem Sammeln (*collect*) von Daten, die die äußere Umgebung oder den Zustand des Systems betreffen. Auch Daten über die Änderungsrate der Zustände können gesammelt werden. Zusammen mit hinterlegten Benutzeranforderungen sind diese eine wichtige Entscheidungsgrundlage. Eine Übersicht über wichtige Größen für die Entscheindung findet man in Kapitel 2.1.4. Anschließend analysiert (*analyze*) das System die gesammelten Daten. Hierbei kommen bereits einige Fragen auf: Wie kann man auf Basis der gesammelten Daten den Systemzustand ableiten? Oder wieviele der Daten alter Zustände braucht man möglicherweise für Entscheidungen in der Zukunft? Anschließend wird eine

[3] Der Closed Control Loop ist ursprünglich ein Teil eines Regelkreises, welcher Feedback und Selbstkorrektur in Abhängigkeit von Eingang und Ausgangssignal ermöglicht.

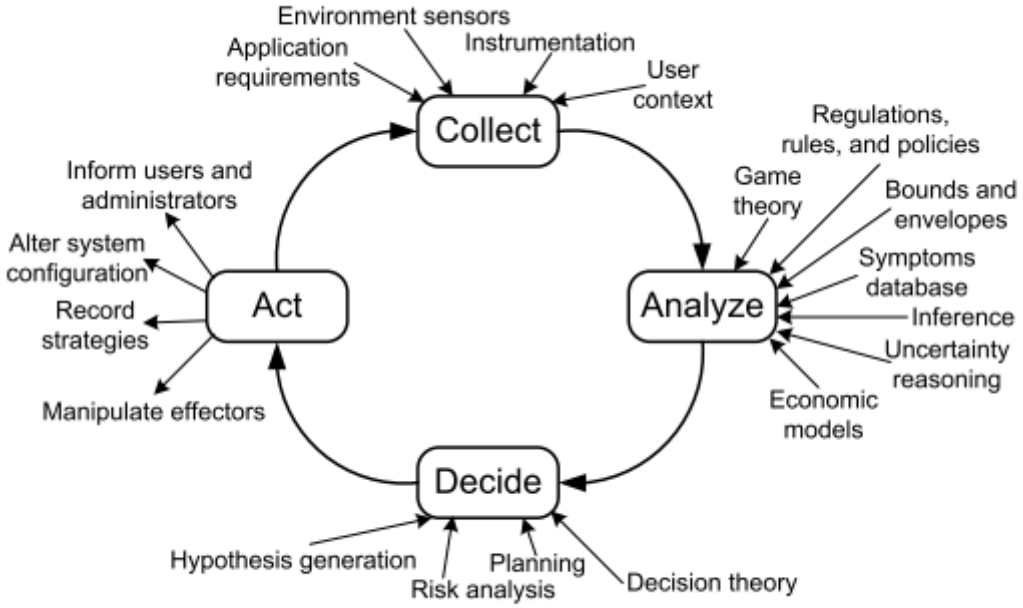

Abbildung 2.2: Autonomic Control Loop - Dobson u. a. (2006)

Entscheidung getroffen ob und wie das System adaptiert wird, um den optimalen Zustand zu erreichen (*decide*). Neben einer möglichen Risikoanalyse gibt es vielfältige Ansätze die eine Entscheidungsfindung unterstützen. Beispiele wären hierbei eine Abschätzung der Auswirkung einer Adaption auf das Systemverhalten, die Rekonfigurationskosten, oder auch eine nutzenorientierte Berechnung des zu erwarteten Mehrwerts. Eine genauere Erläuterung findet sich in Abschnitt 2.1.5.2. Zuletzt muss das System die Adaption durchführen (*act*), falls die Adaptionentscheidung positiv ausgefallen ist.

2.1.4 Kontext

Jede Adaptionsentscheidung eines selbst-adaptiven Systems verwendet per Definition eine oder mehrere Größen des inneren (*self*) oder äußeren(*context*) Zustands, oder Größen die von inneren oder äußeren Zuständen beeinflusst werden. Aus diesem Grund nennt man sie auch kontextsensitiv (*context-aware*)(Geihs (2007), siehe auch Abschnitt 2.1.2). Die Anforderungen des Benutzers spielen dabei zusätzlich eine wichtige Rolle. Somit muss die Anwen-

dung Daten aus unterschiedlichen Kontexten berücksichtigen. Schilit u. a. (1994) klassifiziert die Kontextinformationen in Ressourcen-, Umgebungs- und Benutzerkontext.

Benutzerkontext
Zu dem Benutzerkontext zählen die Anforderungen des Benutzers an die Anwendung. Beispielsweise stellen Benutzerprofile, wie sie in vielen Betriebssystemen für tragbare Computer vorhanden sind, bereits eine einfache Art der Adaption dar. Jedoch können Benutzeranforderungen auch für selbst-adaptive Systeme verwendet werden, müssen jedoch passend hinterlegt sein. Zudem lassen sich auch indirekte Informationen wie Daten aus einem Terminkalender entnehmen und möglicherweise verwerten. Das automatische Erfassen stellt sich jedoch als ungleich schwieriger als die Auswertung statischer Informationen heraus. (Geihs (2007))

Umgebungskontext
Auch der *Umgebungskontext* bzw. *Umgebungszustand* (manchmal auch *Domain Characteristics*) ist ein wichtiges Kriterium der Adaption. Als Beispiel sei hierbei sei das *Transmission Control Protocol* (TCP) herangezogen, welches abhängig von der Netzlast und anderen Informationen über das Netzwerk, in welchem sich der Sender befindet eine parametrische Adaption durchführt. (vgl. Kurose und Ross (2008)). Weitere Beispiele wären hier sämtliche Umgebungssensoren wie GPS-Sensor, Thermometer, oder Schallpegelmesser. Aber auch Größen wie beispielsweise die Anzahl der Knoten in verteilten Systemen(vgl. Gross u. a. (1999)) fallen in den Umgebungskontext und spielen bei der Adaptionsentscheidung eine wichtige Rolle.

Ressourcenkontext
Den inneren Zustand der Anwendung oder des Systems bezeichnet man auch als den *Ressourcenkontext*, welcher Größen wie CPU-Last und Akkustand umfasst. Meist werden diese Daten vom Betriebssystem über Schnittstellen der Software zur Verfügung gestellt.

Die konkreten Ausprägungen der Kontexte werden innerhalb des Regelkreises (siehe 2.1.3) in der *collect*-Phase gesammelt. Oft korrelieren die unterschiedlichen Kontexte oder beeinflussen sich gegenseitig. Ein Beispiel ist das Videostreaming, bei dem je nach vorhandener Bandbreite die Videoqualität angepasst werden kann. Da die Benutzeranforderung (*Benutzerkontext*) in diesem Fall beispielsweise eine Wiedergabe ohne Aussetzer sein könnte, ist die Erfüllung der Anforderung also direkt abhängig von dem *Ressourcenkontext*.

Die *Zeit* ist abseits der oben angeführten Klassifizierung eines der wichtigsten Kriterien einer Adaptionsentscheidung (vgl. Gouda und Herman (1991)). Die Zeit wie lange eine Funktion dauern darf, spielt in den meisten Benutzeranforderungen eine Rolle. Zudem wird

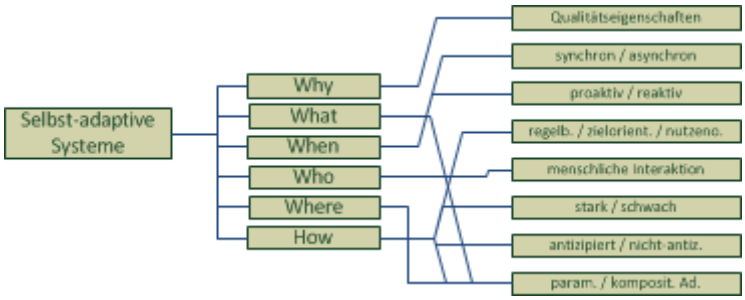

Abbildung 2.3: Hierachische Klassifizierung nach den 5 Ws

die Ausführungsgeschwindigkeit meistens vom *Ressourcenkontext* oder *Umgebungskontext* beeinflusst. Ein Beispiel für die Abhängigkeit vom Umgebungszustand ist das Finden der schnellsten Route durch ein Netzwerk für eine Punkt-zu-Punkt-Verbindung. Somit sollte die benötigte Zeit der Adaption beim Treffen einer Adaptionsentscheidung eine Rolle spielen, um ein genaues Ergebnis zu erlangen (siehe auch Abschnitt 2.1.5.3).

2.1.5 Klassifizierung

Es gibt viele unterschiedliche Möglichkeiten ein adaptives Verhalten umzusetzen, abhängig von den Anforderungen an das System. Salehie und Tahvildari (2009) verwenden die *5 Ws*[4] um passende Anforderungen aufzustellen. Die *5 Ws* sind Fragekategorien, die vor und während der Entwicklung eines solchen Systems gestellt und beantwortet werden sollten. Das *When* adressiert temporale Charakteristiken des Systems. Dazu gehören auch Rekonfigurationsshäufigkeit über einen gewissen Zeitraum oder Zeitpunkt der Rekonfiguration. Die Ebene der Adaption bzw. deren Granularität beschreibt das *Where*. Das *How* steht für Fragen, die die Adaptionskosten und Auswirkungen betreffen, und ob diese in der Adaptionsentscheidung berücksichtigt werden. *Who* bewertet den Grad der Automation. Beispielsweise am Falle einer einfachen Adaption, wie den oben erwähnten Benutzerprofilen, hat der Benutzer aktiven Einfluss auf die Adaption. Viele selbst-adaptive System können sich jedoch auch ohne Eingriff von Außen verändern.Das *What* definiert welche Attribute oder Artefakte mit der Adaption verändert werden können, beispielsweise Parameter oder Methoden bis hinzu ganzen Komponenten, und welche Ressourcen davon betroffen sein können. Zuletzt behandelt *Why* die Ziele der angewendeten Adaption, beispielsweise Angemessenheit oder Performanz, die oft aus den *Self-**-Eigenschaften abgeleitet werden können. (Salehie und Tahvildari (2009)) Jegliche Klassifizierungsansätze von adaptiven Systemen tangieren ein oder mehrere dieser Fragekategorien. Folgend werden einige Klassifizierungen vorgestellt, die in Abbildung 2.1.5 hierarchisch angeordnet sind.

[4]Sechs Fragen (What,Who,When,Why,Where,How) entstammend einem Gedicht aus Kipling (1902)

2.1.5.1 Adaptionsebene

Bei der Realisierung eines adaptiven Verhaltens wird in Geihs (2007) und McKinley u. a. (2004) zwischen zwei Arten unterschieden, der *parametrischen Adaption*, und der *kompositionellen Adaption*. Dabei werden das *How*, das *What* und das *Where* tangiert. *Parametrische Adaption* bedeutet, dass der Zustand der Umgebung interne Parameter der Anwendung beeinflusst, und sich damit eine Funktionalität auf vordefinierte Weise ändern kann. Also verändert diese Adaptionsart nicht die Struktur einer Anwendung, sondern lediglich einzelne Stellgrößen. Die Adaptionsmöglichkeiten der Funktionalität werden vom Entwickler bereits zur Entwurfszeit festgelegt. So kann nachträglich keine Funktionalität angepasst oder erweitert werden. (vgl. Geihs (2007)). Ein Beispiel dafür ist in dem *Transmission Control Protocol* (TCP) zu finden. Beim TCP stellt jeder Sender die Rate, mit der er Verkehr in seine Verbindung überträgt, als Funktion der wahrgenommenen Überlast ein. (vgl. Kurose und Ross (2008))

Kompositionelle Adaption ermöglicht den kontextabhängigen Austausch von Komponenten zur Laufzeit und findet somit auf Architekturebene statt. Dadurch kann sich die ganze Anwendungsstruktur des Softwaresystems verändern, um sich ihrer aktuellen Umgebung anzupassen. Man bezeichnet diese Form der Adaption deshalb auch als *architekturbasierte Adaption* (vgl. wen Cheng u. a. (2004)). Im Vergleich zur rein parametrischen Adaption bietet diese Form der Adaption eine weitaus höhere Flexiblität und Mächtigkeit. Jedoch stellt Sie auch höhere architektonische Anforderungen. So werden nach McKinley et al., neben einem durchgängig komponentenbasiertem Design, noch Technologien zur Reflexion des Systems und für die *Separation of Concerns*[5] benötigt. ((vgl. McKinley u. a. (2004), Abbildung 2.1) Somit eignen sich insbesondere komponentenbasierte Systeme gut, um dieses adaptive Verhalten zu implementieren (siehe Kapitel 2.2).

2.1.5.2 Adaptionsentscheidung

In den vorangegangenen Abschnitten wurden bereits eine Klassifizierung von Adaptionsarten, sowie die wichtigsten Größen welche eine Adaptionsentscheidung beeinflussen vorgestellt. Diese Daten werden zunächst, falls nötig für eine Entscheidung analysiert (Regelkreis *analyze* 2.1.3) und auf Basis der vorangegangenen Analyse anschließend entschieden (Regelkreis *decide*). Laut McKinley u. a. (2004) ist die Entscheidungsfindung eine der wichtigsten Herausforderungen bei der Entwicklung eines selbst-adaptiven Systems. Es gibt unterschiedliche Ansätze wie man eine Entscheidung treffen kann. Die Entscheidungsfindung gehört zum *How*.

[5] *Separation of Concerns* ist ein Softwareparadigma, nachdem separate Funktionalitäten in unabhängigen unterschiedlichen Programmteilen realisiert werden sollten.

Regelbasierte Entscheidung
In regelbasierten Systemen steuert eine Menge vorgegebener Regeln das Anpassungsverhalten (Li u. a. (2000)). Beispielsweise könnte eine solche Regel wie folgt lauten: *FALLS (Bandbreite<100) DANN (verringere Bit-Rate)*, um beispielsweise ein ruckelfreie Wiedergabe zu ermöglichen. Ein Problem der Situations-Aktions-Regeln ist, dass sie sich nur bedingt für dynamische Adaption eigenen. Einerseits steigt bei größeren Adaptionsmöglichkeiten die Komplexität immens. Andererseits müssen alle Möglichkeiten zur Entwurfszeit feststehen, wodurch sich keine komplett dynamische Adaption zur Laufzeit realisieren lässt (vgl. Geihs (2007)).

Zielorientierte Entscheidung
Hinter der zielorientierten Entscheidung steckt die Idee, ein zu erreichendes Ziel vorzugeben und die betroffenen Komponenten selbst entscheiden zu lassen was getan werden muss um diese Ziel zu erreichen. Das oben aufgeführte Beispiel mit der Bandbreite würde sich in einem zielorientierten System durch die Angabe der Anforderung nach einer Wiedergabe ohne Aussetzer formulieren lassen. Dem System bleibt es selbst überlassen das vorgegebene Ziel zu erreichen. Die Entscheidungsfindung wird dadurch zum Planungsproblem. Gut abbilden lässt sich ein solche Entscheidungsfindung mit Agententechnologie. In einigen Multi-Agenten-Systemen werden die Agenten als autonom, ziel-orientiert beschrieben, die die Fähigkeit haben, mit ihrer Umwelt und untereinander zu kommunizieren. Durch die Kooperation der individuellen Agenten wird ein Gesamtziel verfolgt (Tesauro u. a. (2004)).

Nutzenorientierte Entscheidung
Ein probates Mittel ist die Abbildung der Nützlichkeit mit einer Nutzenfunktion, die den Nutzen einzelner Realisierungsvarianten numerisch auf einer Skala abbildet. Mit geeignet gewählten Nutzenfunktionen lassen sich ganz unterschiedliche Aspekte der Adaption abbilden. Beispielsweise können Aussagen über Kosten einer Kommunikationsverbindung oder über den Ressourcenverbrauch unterschiedlicher Komponenten getroffen werden. Die Adaptionsentscheidung basiert daher auf einer vom Entwickler vorgegebenen Nutzenfunktion, mit deren Hilfe diejenige Konfiguration des Systems oder Anwendung ausgewählt werden kann, welche für den aktuellen Kontext den größten Nutzen hat (vgl. Geihs (2007)). Die Adaptionsentscheidung wird zum Optimierungsproblem - bzw. zur Suche nach der idealen Nutzenfunktion.

2.1.5.3 Auswirkungsvorhersage

Die Auswirkungsvorhersage ist eine wichtige Größe in der Entscheidungsfindung. Dabei geht es um die Auswirkungen einer Rekonfiguration auf das Systemverhalten, sowie um anfallende Kosten wie die Ausführungszeit oder für die Adaption benötigte Ressourcen. Basierend

auf den Auswirkungen und den Kosten können Adaptionen in zwei Kategorien unterteilt werden: *starke* und *schwache* Adaptionen. *Starke* Adaptionen können Subsysteme oder Module der Anwendung verändern, hinzufügen, entfernen, oder ersetzen, und finden typischerweise auf Architekturebene statt. Somit ist die komponentenbasierte Adaption eine starke Adaption. *Schwache Adaptionen*, zu denen auch die parametrische Adaption zählt (siehe 2.1.5.1), verändert nur einzelne Stellgrößen (z. B. die Bitrate beim Streaming) oder benutzen vorgefertigte statische Methoden (z.B. Datenkompression, Caching) mit geringen, meist lokalen, Auswirkungen. Da es durchaus adaptive Systeme mit starken und schwachen Adaptionsarten gibt, ist dieser Aspekt durch den möglicherweise höheren Aufwand bei der Adaptionsentscheidung zu beachten. Die Auswirkungsvorhersage gehört ebenso zum *How*.(vgl. Salehie und Tahvildari (2009))

2.1.5.4 Adaptionszeitpunkt

Zuletzt kann unterschieden werden, wodurch der Prozess bzw. der Regelkreis ausgelöst wird. Da diese Einteilung das *When* betrifft, nennt man sie auch temporal. Der Auslösezeitpunkt kann wie folgt unterschieden werden: Bei *synchroner* Adaption findet die Veränderung *synchron* mit dem Aufruf, respektive mit der Anwendungslogik statt und ist weder reflexiv, sprich ist nicht abhängig von dem internen Zustand des Systems, noch abhängig von äußeren Umständen. Eine *asynchrone* Adaption (oder auch Änderungsverhalten) bedeutet, dass der Zeitpunkt der Adaption abhängig von den äußeren (*context*) und internen (*self*) Umständen und nicht von einem aufruf-behaftetem Ereignis ist. Man nennt dieses Verhalten auch kontext-basiert. (Geihs (2007)). Beispiele hierfür sind der Wechsel von Kabel auf Akkubetrieb eines mobilen Endgeräts, oder die Veränderung der zur Verfügung stehenden Netzwerkbandbreite.

Eine dazu orthogonale Klassifizierung die auch den Zeitpunkt der Adaptionsentscheidung betrifft, ist die nach *proaktivem* und *reaktivem* Adaptionszeitpunkt. Der *proaktive* Ansatz versucht vorauszusehen wann eine Veränderung des *self* oder des *Context* passieren wird. Der *reaktive* Ansatz reagiert nur auf tatsächliche Veränderungen der beiden Zustände. (Parashar und Hariri (2005))

2.2 Komponentenbasierte Entwicklung

In Abschnitt 2.1.5 wurden einige Adaptionsarten vorgestellt. Dabei wurde bereits erwähnt dass sich für die architekturielle Adaption der Einsatz von Komponenten bzw eines Komponentenmodells besonders eignet. Die Anwendung wird dann im Rahmen einer komponentenbasierten Entwicklung erstellt. Dabei handelt es sich um ein Paradigma, bei dem sich der Großteil einer Anwendung aus klar abgegrenzten Komponenten zusammensetzt.

Komponentenbasierte Architekturen zeichnen sich somit durch eine hohe Kohäsion und lose Kopplung unter den Komponenten aus. Die Verantwortlichkeiten sind klar auf die Komponenten aufgeteilt (*Separation of Concerns*). Damit erreicht man eine gute Wiederverwendbarkeit und Reduktion der Gesamtkomplexität einer Anwendung. Durch die wenigen Wechselwirkungen mit anderen Komponenten bleiben diese lokal änder- oder austauschbar. (vgl. Andresen (2004))

Die Idee, Software aus vorgefertigten Einzelteilen herzustellen fand unter dem Namen *„Mass-produced Software Components“* 1968 erste Erwähnung auf der *NATO-Conference on software engineering* (siehe McIlroy (1968)). Das Hinzufügen von Pipes[6] und Filter in das Betriebssystem Unix war anschließend die erste Implementierung dieser Idee.

2.2.1 Komponente

Der Begriff der Komponente wird jedoch in der Literatur unterschiedlich definiert. Eine vielfach zitierte Version ist das Ergebnis der Workshop on Componont-Oriented Programming bei der 10. European Conference on Object Oriented Programming (ECOOP 1996): *„A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.“ (Szyperski (2002))*. Heineman et al. (Heineman und Councill (2001)) definiert eine Komponente weniger abstrakt und auf die komponentenbasierte Entwicklung bezogen: *„Eine Software-Komponente ist ein Software-Element, das konform zu einem Komponentenmodell ist und gemäß einem Composition-Standard ohne Änderungen mit anderen Komponenten verknüpft und ausgeführt werden kann.“*. Laut Siedersleben (vgl. Siedersleben (2004)) definiert sie noch zusätzlich ihre angebotenen Dienste inklusive der genauen Semantik der Schnittstellen. Letztlich kann eine Komponente zusammenfassend als Element der komponentenbasierten Entwicklung betrachtet werden, die eine Schnittstelle zur Kommunikation mit anderen Komponenten besitzt, und in der Regel eine hohe Kohäsion und lose Kopplung unterstützt. Durch diese Eigenschaften, geringe Abhängigkeit und feste Schnittstelle, wird die Komponente austauschbar und somit gut geeignet für die Realisierung eines adaptiven Verhaltens, in Form der bereits erwähnten komponentenbasierten Adaptivität. (vgl. 2.1.5.1,Geihs (2007)). Dass Komponenten eine identische Schnittstelle anbieten können, macht sie jeweils bezüglich der Erfüllung nichtfunktionaler Anforderungen vergleichbar. Falls zwei Komponenten beispielsweise die selbe Schnittstelle erfüllen, die auch semantisch die gleiche Funktionalität bietet, werden Metriken wie Durchlaufzeit unter spezifischen Umständen oder Robustheit vergleichbar. Jedoch ist die technologisch-konkrete Form einer Komponente vor allem vom jeweilig verwendeten Komponentenmodell (siehe 2.2.2) abhängig. Zusammengesetzt kann sie mit anderen Kom-

[6] Eine Pipe ist in Unix eine Menge von Prozessen, die mit Hilfe von Streams aneinander gereiht werden.

ponenten zu größeren Software-Konstrukten verbunden, und somit als Softwarebaustein angesehen werden (vgl. Atkinson u. a. (2003)). Eine Komponente kann auch eine Menge von Modulen oder Subsystemen[7] sein. Ein Modul somit auch eine minimale Komponente (Hug (2001)).

Komponenten vereinfachen die unternehmensweite und unternehmensübergreifende Nutzung von Software grundlegend. Verbesserte oder veränderte Versionen der Komponente lassen sich aufgrund deren Eigenschaften leicht austauschen ohne die Schnittstelle grundsätzlich ändern zu müssen. Dadurch kann die Komponente problemlos kontinuierlich qualitativ verbessert werden. Durch diese Konzentration auf die Qualität der Software reduzieren sich die Gesamtkosten der Entwicklung. Zudem haben Komponenten als Experten-Module meist weniger Fehler und weisen ein besseres Laufzeitverhalten auf, als ständig neu zu entwickelnde geschlossene Systeme. (vgl. Andresen (2004)). All diese Eigenschaften sind auch im Bereich der adaptiven Software von Nutzen, falls die zu verändernden Funktionalitäten als Komponenten abgebildet werden können.

2.2.2 Komponentenmodell

Ein Komponentenmodell ist eine konkrete Ausprägung der komponentenbasierten Entwicklung. Laut Grun und Thiel (Gruhn und Thiel (2000)) legt es einen Rahmen für die Entwicklung und Ausführung der Komponenten fest. Dieser Rahmen stellt strukturelle Anforderungen hinsichtlich Verknüpfungs- und Kompositionsmöglichkeiten sowie verhaltensorientierte Anforderungen bezüglich der Kollaboration an die Komponenten. Das Modell legt nicht nur fest in welcher Form und mit welchen Eigenschaften die Komponenten vorliegen müssen, sondern auch wie die Komponenten miteinander interagieren und verbunden werden können. Als weit verbreitete Vertreter seien hier CORBA Component Model[8], Distributed Component Object Model (DCOM)[9], und OSGi (siehe auch Abschnitt 3.2) genannt. Systeme, welche auf einem Komponentenmodell basieren, ermöglichen oft eine dynamische Rekonfiguration zur Laufzeit der Anwendung, welches für eine komponentenbasierte Adaption (siehe 2.1.5.1 ein wichtiges Kriterium ist. Komponententechnologie verspricht zudem Anwendungsentwicklung durch Service-Spezifikation zu ermöglichen, beispielsweise durch die Auswahl und Verbindung verschiedenerer Komponenten. Der Anwendungsentwickler muss dabei den physikalischen Ort an dem sich die Komponente befindet nicht kennen, da der Zugriff auf entfernte Komponenten von der darunterliegenden Middleware kaschiert wird. Dadurch erreicht man eine höhere Wiederverwendung, geringere Entwicklungskosten und höhere Stabilität. Aus diesen Gründen sind komponentenbasierte Middleware-Plattformen in der

[7] Ein Subsystem ist eine Menge von Modulen, Dokumenten und Ressourcen Hug (2001)

[8] Die Common Object Request Broker Architecture ist eine Spezifikation für eine objektorientierte Middleware

[9] DCOM ist ein objektorientiertes RPC-System

Industrie weit verbreitet: in der Bank- oder Finanzindustrie wie auch in kleineren Multimedia-Anwendungen, wie man sie typischerweise auf mobilen Endgeräten findet (vgl. Amundsen u. a. (2004)).

Effiziente Komponenten-Architekturen müssen diversen Anforderungen genügen. Laut Andresen (2004) müssen Änderungen, Verbesserungen, und Erweiterungen auf einfache und flexible Weise und eine einfache Wiederverwendung von Komponenten ermöglicht werden. Zudem muss das System die Anforderungen in Bezug auf Robustheit, Zuverlässigkeit, Performanz, Sicherheit, und Skalierbarkeit erfüllen.

2.3 Quality of Service für Komponenten

Im Rahmen dieser Arbeit soll erarbeitet werden, wie eine mobile Anwendung sich an sich ändernde äußere und innere Zustände und Ressourcen anpassen kann. Um entscheiden zu können ob eine Änderung des Verhaltens, beispielsweise durch Austausch einer Komponente, eine relevante Verbesserung bewirkt, müssen neben den Benutzeranforderungen auch die nicht-funktionalen Eigenschaften der Komponenten definiert sein, da die funktionalen Eigenschaften nicht beeinträchtigt werden sollen. Wenn es um die Erfüllung von nicht-funktionalen Eigenschaften geht, spricht man auch von *Quality of Service* oder Dienstgüte. Diese ist eine Menge von Qualitätsanforderungen die beschreiben wie stark die Güte eines Dienstes den gestellten Anforderungen entspricht. *Quality of Service*-Technologien werden in der Regel mit Netzwerk- oder Multimediatechnologien assoziiert, da diese dort inzwischen unverzichtbar geworden sind (vgl. Abbas und Kure (2010)). In diesem Kontext kann eine QoS-Anforderung beispielsweise der Datendurchsatz sein der benötigt wird um eine Videodatei ohne Aussetzer übertragen zu können. Jedoch spielen Performanz, Ressourcenverbrauch bzw. die nicht-funktionalen Anforderungen auch bei der Entwicklung und Verbreitung von Softwarekomponenten eine Rolle. Benutzer von Anwendungen auf mobilen Geräten stellen ebenfalls Anforderungen an die Verhaltensweise einer Applikation. Dazu zählen zum Beispiel Ausführgeschwindigkeit von Funktionen, möglichst geringer Energieverbrauch im Akkubetrieb, oder eine akzeptable Geschwindigkeit bei der Datenübertragung über die verfügbaren Datenkanäle. Natürlich hat eine Anwendung auf ihre aktuellen äußeren Umstände und die technologischen Limitierungen des Engeräts keinerlei Einfluss, kann jedoch abhängig vom äußeren und inneren Zustand seine Arbeitsweise anpassen. Anwendungen und Architekturen die sich an ihren inneren und an die äußeren Zustände anpassen können nennt man, wie in dem obigen Kapitel beschrieben, selbst-adaptiv.

Komponenten können untereinander eine ähnliche Funktionalität bieten, jedoch unterscheidet sich möglicherweise deren Anforderung an eine Internetanbindung oder vorhandenen Speicher, sodass sich deren jeweilige situationsbedingte Eignung deutlich unterscheiden

kann. Dennoch erscheinen QoS-Konzepte nur selten in Komponentenstandards und in der komponentenbasierten Entwicklung. (vgl. Brahnmath u. a. (2002)). Dies mag darin begründet liegen, dass es deutlich schwerer ist geeignete Qualitätsmerkmale für die unterschiedlichen Anwendungsgebiete von Software zu finden. Um zunächst einen einheitlichen Qualitätsbegriff zu finden werden nun einige Aspekte und Definitionen von Software- und Komponentenqualität vorgestellt.

2.3.1 Definitionen und Terminologie

Im Softwareengineering beschreibt der Begriff Softwarequalität zunächst zwei unterschiedliche Eigenschaften von Software. Zum einen wie gut eine Software aufgebaut ist (*quality of design*) und zum anderen in wie weit die Software diesem Design entspricht (*quality of conformance*)(vgl. Pressman (2005), Kan (1994)). Dennoch existiert eine Vielzahl weiterer Definitionen die insbesondere die Benutzeranforderungen oft mit einbeziehen. Weinberg (1992) unterstreicht dies mit seiner Definition *„Quality is value to some person“*. Dabei spricht man auch oft von der *Gebrauchstauglichkeit* bzw. *fitness for purpose*) (oder auch *fitness for use* in Juran (1988)). Balzert stellt fest, dass der Begriff der Softwarequalität nicht operabel und deshalb in der Praxis nicht direkt anwendbar ist. Deshalb benötigt man Qualitätsmodelle um das Konzept der Softwarequalität zu operationalisieren (vgl. Balzert (1997)). Ein Beispiel ist dabei das Qualitätsmodell nach ISO 9126. Nach diesem versteht man unter Softwarequalität die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, die sich auf dessen Eignung beziehen festgelegte oder vorausgesetzte Erfordernisse erfüllen zu können (ISO/IEC (2001), Balzert (1997)). Dabei ist die Softwarequalität als ein 6-Tupel beschrieben (siehe Abbildung 2.3.1). Dieses Modell beschreibt sechs Hauptcharakteristiken und jeweils drei bis sechs Untercharakteristiken an Qualitätseigenschaften.

Das Qualitätsmodell ähnelt dem bereits 1977 entwickelten Modell vom McCall (McCall (1977)), welches ursprünglich aus 55 sogenannten Faktoren bestand, die letztlich aus Gründen der Vereinfachung auf 11 beschränkt wurden. Weitere Ähnlichkeiten lassen sich zudem zu dem 1978 entwickelten Modell vom Boehm (Boehm (1978)) finden. Qualitätsfaktoren sind nicht unabhängig voneinander und beeinflussen sich gegenseitig (vgl. Perry (1987)).

McConnell (2004) unterscheidet bei Qualitätseigenschaften generell nach zwei Kategorien: nach *internen* und *externen* Qualitätscharakteristiken. *Externe* Qualitätscharakteristiken betreffen die Teile der Anwendung, die von dem Benutzer wahrgenommen werden, beispielsweise die Robustheit. *Interne* Qualitätscharakteristiken betreffen für den Anwender verborgene Teile, z.B. die Codequalität.

Abbildung 2.4: ISO 9126 ISO/IEC (2001)

2.3.2 Zusammenhang Qualität und selbst-adaptive Systeme

In Abschnitt 2.1.2 wurden die *Self-**-Eigenschaften adaptiver Systeme vorgestellt. Zwischen den Qualitätseigenschaften einer Anwendung und diesen Eigenschaften lässt sich der Bezug herstellen, welche Qualitätseigenschaften betroffen sind und wie diese von Self-*-Eigenschaften beeinflusst werden. Die Eigenschaft *Self-configuring* beeinflusst beispielsweise die Qualitätseigenschaften Wartbarkeit, Funktionalität, Übertragbarkeit, Benutzbarkeit, und gegebenenfalls auch Zuverlässigkeit (Salehie und Tahvildari (2009)). Für Systeme mit der *Self-healing*-Eigenschaft besteht das Hauptziel die Qualitätseigenschaften Wartbarkeit und Zuverlässigkeit zu maximieren (Ganek und Corbi (2003)). *Self-optimizing* hängt wiederum stark mit der Effizienz zusammen. Wie in Abschnitt 2.3.1 erläutert, sind die Benutzeranforderungen ausschlaggebend für die resultierende Qualität und somit auch für selbst-adaptive Systeme.

2.3.3 Benutzeranforderungen an und Eigenschaften von mobilen Anwendungen

Nachdem der Qualitätsbegriff in der Software erläutert wurde und das Qualitätsmodell der ISO 9126 vorgestellt wurde, muss für die Bestimmung der Qualitätseigenschaften der Komponenten ermittelt werden, welche qualitativen Eigenschaften im Rahmen einer erwarteten Dienstgüte für mobile Anwendungen relevant sind. Wie in Abschnitt 2.3.1 erläutert hängt die wahrgenommene Qualität vom Benutzer und dessen Anforderungen ab. Für die Entwicklung einer mobilen Applikation, die durch selbst-adaptives Verhalten eine stets optimale Gebrauchstauglichkeit erhält, ist es somit wichtig zunächst die Benutzeranforderungen an, sowie die grundsätzlichen Eigenschaften von mobilen Anwendungen zu betrachten. Neben

dieser Aufgabe besteht anschließend die Herausforderung diese Anforderungen in zur Laufzeit von der adaptiven Software zu erreichende Ziele umzuformulieren (Kramer und Magee (2007)). Salehie und Tahvildari (2009) nennen hier goal-oriented Requirements-Engineering[10] als probates Mittel.

Gebauer u. a. (2007) untersuchten mittels einer Umfragen, welche Anforderungen für Benutzer von mobiler Technologie besonders wichtig waren. Neben technischen Eigenschaften des mobilen Geräts ging es dabei auch um Anforderungen im Bereich Performanz, bezüglich der Ausführungsgeschwindigkeit und der Akkulaufzeit. Es stellte sich heraus dass diese für die Benutzer als am wichtigsten empfunden wurden. Dadurch dass sich diese Qualitätseigenschaften jedoch gegenseitig beeinflussen können, gilt es hier eine optimales Verhältnis zu finden. Da die Motivation dieser Arbeit unter anderem in dem Optimieren der Gebrauchstauglichkeit für den Anwender besteht, liegt der Fokus zunächst auf den nichtfunktionalen Qualitätscharakteristiken. Welche für das Konzept verwendet werden, wird in Abschnitt 5.2.3 des Kapitels 5 erläutert.

2.3.4 Metriken und Messbarkeit

Um die Qualitätscharakteristiken der Komponenten in einer Nützlichkeitsfunktion (siehe 2.1.5.2) verwenden zu können, müssen sie zunächst in einer Form bzw. Skalienniveau vorliegen, die sie untereinander vergleichbar macht. Das Skalienniveau bestimmt letztlich welche mathematischen Operationen mit der entsprechend skalierten Variable zulässig sind, und welche Transformationen durchgeführt werden können ohne Informationen zu verlieren (vgl. Fahrmeir und Hamerle (1984)). Höhere Skalenniveaus führen dabei auch zu einem höheren Informationsgehalt. Die *Nominalskala* ist das niedrigste Skalenniveau. Für verschiedene Objekte kann nur zwischen Gleichheit und Ungleichheit unterschieden werden, beispielsweise „blau ist nicht grün“. *Ordinalskalierte* Merkmale können mit Hilfe von Operatoren der Art „größer“ bzw. „kleiner“ angeordnet werden. Über etwaige Abstände kann jedoch keine Aussage getroffen werden. Mit Hilfe von *intervallskalierten* Daten können Aussagen über eine Differenz zwischen zwei geordneten Merkmalen getroffen werden. Da hier jedoch kein absoluter Nullpunkt exisiert stellt die Multiplikation der Merkmalesausprägungen keine sinnvolle Operation dar. Jedoch kann die Ungleichheit durch Differenzbildung quantifiziert werden. Falls ein absoluter Nullpunkt vorhanden ist (z.B. CPU-Last=0 Prozent), spricht man von einer Verhältnisskala, bei der neben den anderen Operationen auch Multiplikation und Division erlaubt sind, und somit Verhältnisse gebildet werden dürfen. (vgl. Backhaus u. a. (2008))

[10] ist eine Methode die sog. Goals benutzt um Requirements zu analysieren, zu dokumentieren, zu strukturieren und modifizieren. Ein Goal beschreibt hierbei die unterschiedlichen Aufgaben die das System lösen soll.Van Lamsweerde (2001)

Kapitel 3

Technologische Grundlagen

Im zweiten Teil der Grundlagen werden die technologischen Grundlagen vorgestellt, welche für das Verständnis der Implementierung des Konzepts von Nöten sind. Hierbei wird neben den Eigenschaften auch die Entscheidungsfindung, die zur Auswahl der jeweiligen Technologien geführt hat, erläutert. Anschließend wird das Komponentenmodell OSGi vorgestellt, welches im Rahmen dieser Arbeit auf dem mobilen Betriebssystem Android Einsatz findet.

3.1 Auswahl des Komponentenmodells und der mobilen Plattform

Zur Unterstützung eines adaptiven Verhaltens muss das Komponentenmodell einigen Anforderungen genügen, die in der Regel jedoch bereits für die meisten effizienten Komponentenmodelle gelten. Andresen (2004) stellt die Anforderungen an die Architektur eines komponentenbasierten Systems, dass Änderungen, Verbesserungen, und Erweiterungen auf einfache und flexible Weise und eine einfache Wiederverwendung von Komponenten ermöglicht werden müssen. Weitere persönliche Anforderungen waren die Unterstützung der Programmiersprache Java, sowie eine Herstellerunabhängigkeit. Die Implementierungen müssen zudem kostenfrei verfügbar sein. Weitere Kriterien sind der Verbreitungsgrad und insbesondere die Tauglichkeit für mobile Plattformen.

Die Plattform OSGi bietet Unterstützung durch viele Großunternehmen und auch durch kleinere Unternehmen aus dem Open-Source-Bereich. Es existiert außerdem eine Vielzahl von Implementierungen, die inzwischen auf allen verbreiteten mobilen Plattformen[11] lauffähig sind. Es existiert eine große Menge an bereits entwickeltem Quellcode, der mit Hilfe des OSGi-Frameworks auf den mobilen Plattformen lauffähig ist. Zudem sind die OSGi-Bundles bis zu einem gewissen Grad zwischen den Implementierungen austauschbar.

[11] Symbian, Windows Mobile, Android, iOS

Die Auswahl der konkreten OSGi-Implementierung richtet sich auch nach Kriterien der mobilen Plattform, des Verbreitungsgrads und der Einfachheit der Verwendung. Zunächst musste demnach die Plattform evaluiert werden. Für Android lassen sich problemlos Java-Programme in Dalvik-VM-Code umwandeln. Zudem ist die Entwicklung für die Plattform durch Emulatoren und kostenloses SDK komfortabel. Zusammen mit der hohen Verbreitung gab dies den Ausschlag für Android. Bei den OSGi-Implementierungen fiel die Wahl auf Apache Felix, welche als einzige Implementierung ohne größere Anpassungen auf der Android-Plattform läuft. Folgend wird das OSGi-Framework unabhängig von der Implementierung in seinen wichtigsten Funktionalitäten kurz erläutert.

3.2 OSGi

Das OSGi-Framework spezifiziert eine Service Plattform, die sich durch ihr komplettes und dynamisches Komponentenmodell auszeichnet. Applikationen oder deren Teile (sog „Bundles“) können zur Laufzeit gestartet, gestoppt, erneuert, installiert oder deinstalliert werden. Die Bundles können dabei lokal oder aus der Ferne verwaltet werden. Es bietet ein Entwicklungsmodell, durch welches Anwendungen (dynamisch) aus vielen verschiedenen (wiederverwendbaren) Komponenten zusammengesetzt werden können. Dabei kommunizieren die einzelnen Bundles über fest definierte Schnittstellen und verbergen so ihre Implementierung (vgl. 3.2).Ursprüngliches Anwendungsszenario war der Einsatz in sog. „Service Gateways“ [12]. Inzwischen wird die Technologie jedoch vielfältig eingesetzt, von Desktop-Anwendungen, über Automobil-Software, bis hin zu mobilen Anwendungen auf PDAs, Tablet PCs, oder Mobiltelefonen. Neben der Bundlearchitektur spezifiziert die OSGi-Plattform einige Standard-Services, die in den Implementierungen der unterschiedlichen Hersteller vorhanden sein müssen. (siehe Alliance (2011b)) OSGi liegt einem Schichtenmodell, bestehend aus sechs Schichten, zugrunde. (siehe Abbildung 3.2)

- Bundles: Bundles sind die OSGi-Komponenten
- Services: Die Serviceschicht verknüpft dynamisch die einzelnen Bundles indem ein Publish-find-bind-Modell unterstützt wird
- Life-Cycle: Die Zugriffsschicht um Bundles zu starten, zu stoppen, zu aktualisieren, und zu deinstallieren.

[12] Im Falle von OSGi bezeichnet man mit „Service Gateway“ beispielsweise einen Application-Server, welcher zwischen dem Internet und einem lokalen Netzwerk vermittelt.

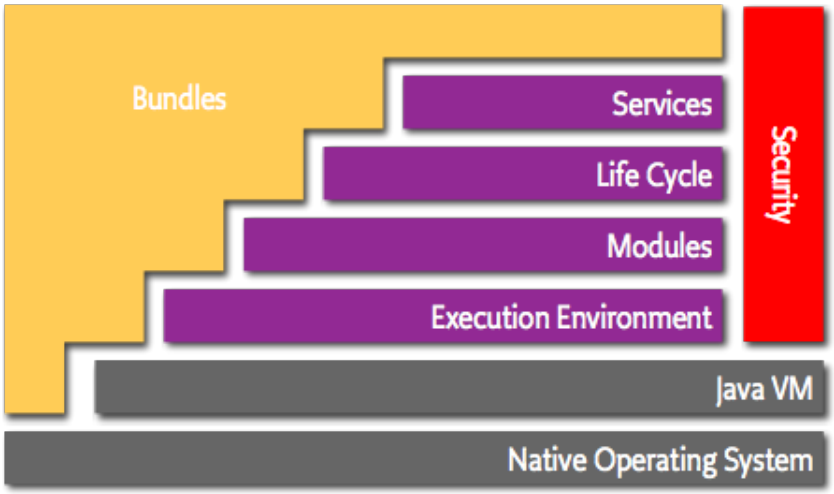

Abbildung 3.1: OSGi Schichten OSGi-Alliance (2011)

- Modules: Die Schicht, die festlegt, wie ein Bundle Code exportieren und importieren kann.
- Security: Die Schicht, die sich um Sicherheitsaspekte kümmert.
- Execution Environment: Definiert welche Methoden und Klassen auf den jeweiligen Plattformen verfügbar sind.

Die Funktionsweisen einiger im Rahmen dieser Arbeit wichtiger Schichten werden folgend genauer erläutert.

3.2.1 Bundle

Ein *Bundle* ist eine fachlich und technisch zusammenhängende Einheit von Klassen und Ressourcen, die als geschlossene Einheit einer OSGi-Plattform hinzugefügt wird. Es besteht aus *Class*-Dateien sowie einem *Bundle Manifest*, in welchem einige Informationen über das Bundle deskriptiv festgehalten werden. Diese Informationen, zu denen beispielsweise etwaige Abhängigkeiten oder die Versionsnummer gehören, können dann vom OSGi-Framework ausgelesen werden. Wie in Kapitel 3.2 angemerkt, können Bundles zur Laufzeit entfernt und hinzugefügt werden. Dies geschiet durch sogenannte *Management Agents* (siehe Abschnitt 3.2.2). *Bundles* können miteinander kommunizieren, in dem man entweder die jeweiligen Klassen explizit mit Hilfe des *Bundle Manifests* exportiert und auf der konsumierenden Seite importiert, oder indem man freizugebende Objekte an einer öffentlichen Registratur (*Service Repository*) anmeldet. Das Objekt kann dabei unter einem oder mehreren Interface-

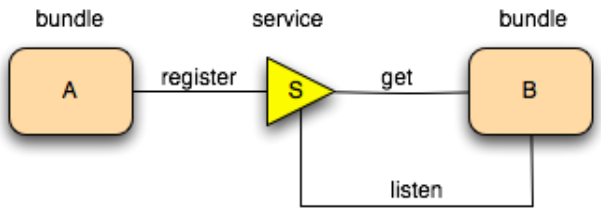

Abbildung 3.2: Bundle-Registrierung (3.2,Wütherich u. a. (2008))

oder Klassennamen angemeldet werden. Über diese Namen können dann andere *Bundles* auf das Objekt zugreifen. Ein *Bundle* kann somit einen *Service* registrieren, anfragen oder warten bis ein *Service* auftaucht beziehungsweise verschwindet. Jede beliebige Anzahl von *Bundles* können den selben *Service* registrieren und auf den Service zugreifen (siehe Abbildung 3.2.1). (siehe Alliance (2011b),Wütherich u. a. (2008))

3.2.2 Bundle Lifecycle

Der *Lifecycle* eines jeden *Bundles* innerhalb einer OSGi-Plattform ist unabhängig von der Anmeldung an der *Service Registry*. Innerhalb dieses *Lifecycles* kann ein *Bundle* sechs verschiedene Zustände annehmen (siehe Abbildung 3.3, vgl. Wütherich u. a. (2008)).

- *Installed*
- *Resolved*
- *Starting*
- *Active*
- *Stopping*
- *Uninstalled*

Nach der Installation des *Bundles* befindet es sich im Zustand *„Installed“*. Falls alle Paketabhängigkeiten erfolgreich aufgelöst werden konnten, wechselt es in den Zustand *„Resolved“*. Der Zustand *„Starting“* beginnt mit dem Aufruf der start()-Methode. Nach erfolgreichen Ausführen der Methode, unter Beachtung der *Activation Policies* befindet sich das *Bundle* im Zustand *„Active“*. Um diesen Zustandsübergang zu beeinflussen kann die sogenannte Lazy-Activation der *Activation Policies* im Manifest aktiviert werden. Danach wird die start()-Methode bis zum ersten Zugriff auf die Klasse angehalten und geht erst anschließend in den Zustand *„Active“* über. (siehe Alliance (2011b),Wütherich u. a. (2008))

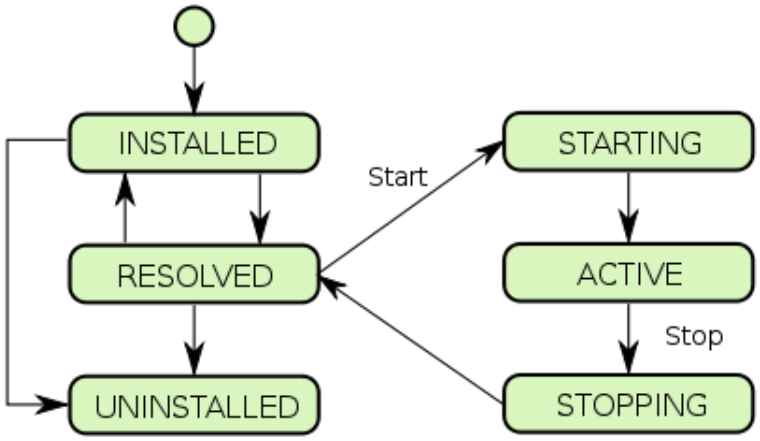

Abbildung 3.3: Bundle Lifecycle (Wütherich u. a. (2008))

3.3 Android

Android ist ein ursprünglich von Google auf Linux-Basis entwickeltes Open-Source-Betriebssystem, welches inzwischen von der *Open Handset Alliance* [13] weiterentwickelt wird. Es ist aufgrund der geringen Hardwareanforderungen sowie den vielfältigen Kommunikationsschnittstellen insbesondere für mobile wie auch Geräte mit beschränkten Ressourcen geeignet. Auf Android laufende Applikationen werden üblicherweise in Java entwickelt. Jedoch muss der resultierende Bytecode cross-kompiliert werden, damit die speziell für Android entwickelte, Java-ähnliche Laufzeitumgebung (Dalvik Virtual Machine) ihn ausführen kann. Die erforderlichen Gerätetreiber sind bereits in den Linux-Kernel integriert. Die Hardware wird durch einige Bibliotheken abstrahiert, welche zum Großteil auf Standard-Linux-Bibliotheken basieren. Die Dalvik Virtual Machine (DVM) ist besonders für den performanten Multitaskingbetrieb auf Geräten mit wenig Speicher und langsamen Ressourcen geeignet. Jede Anwendung läuft dabei in einer eigenen DVM-Instanz, welche sich jedoch einen gemeinsamen Heap teilen. Im weiteren Verlauf des Kapitels soll dem Leser ein Überblick über die wichtigsten Komponenten einer Android-Anwendung verschafft

[13] Die Open Handset Alliance ist ein Industriekonsortium aus mittlerweise 65 Firmen wie Acer, Dell, HTC, LG, Samsung, und Motorola.

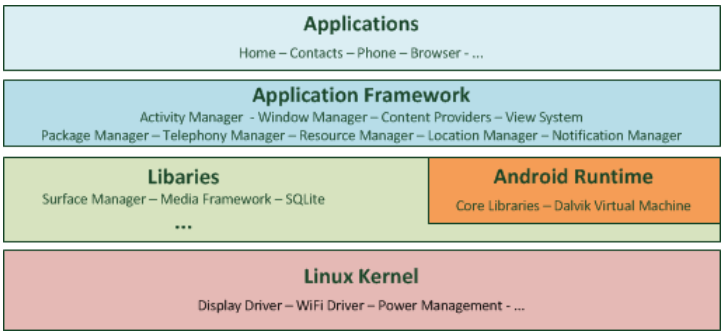

Abbildung 3.4: Android-Architektur Becker u. a. (2010)

werden. (Alliance (2011a)). Android kann auch als Softwarestack bezeichnet werden, der aus Betriebssystem, Middleware, und einigen wichtigen Anwendungen besteht. Diagramm 3.3 zeigt dessen wichtigste Bestandteile. (Becker u. a. (2010))

- *Anwendungsschicht*: Installierte Anwendungen wie beispielsweise der Kalender oder ein Email-Client.
- *Anwendungsrahmen*: Bildet das Bindeglied zwischen für die in Java geschriebenen Programme und der Hardware.
- *Bibliotheken*: Stellen Schnittstellen für den Hardwarezugriff vom Anwendungsrahmen bereit.
- *Android-Laufzeitumgebung*: Beinhaltet einige Laufzeitbibliotheken sowie die virtuelle Maschine
- *Linux-Kernel*: Beinhaltet Gerätetreiber und einige hardwarenahe Verwaltungsprogramme wie z. B. das Memorymanagement.

3.3.1 Android-Komponenten

Üblicherweise werden vier zentrale Android-Klassen als Komponenten bezeichnet, welche im Laufe der Arbeit als „Android-Komponenten“ bezeichnet werden, um nicht mit dem Komponentenbegriff aus Abschnitt 2.2.1 verwechselt zu werden. (siehe 2.2.1). Eine Android-Anwendung - Teil der Anwendungsschicht (siehe 3.3) - besteht typischerweise im Kern aus diesen vier Android-Komponenten: (Becker u. a. (2010))

- *Activity* zur Verwaltung der Benutzeroberfläche
- *Service* zur Ausführung von asynchronen Aufgaben

- *Content Provider* für die Bereitstellung von Schnittstellen an andere Anwendungen
- *Broadcast Receiver* für das Empfangen von Systemnachrichten

Folgend werden die genannten Komponenten kurz etwas genauer erläutert.

3.3.1.1 Activity

In Android besteht die Benutzeroberfläche einer Anwendung in der Regel aus einer Java-Klasse, der sogenannten *Activity* sowie einer XML-Datei. Der Aufbau gleicht somit einer MVVC-Architektur. Die *Activity* ist hierbei der aktive Part, der die Benutzereingaben verwaltet und Informationen zur Verfügung stellt. *Activities* gehören jedoch in Android zu den Komponenten, da sie als abgeschlossene Einheit mit ihrer Business-Logik in andere Anwendungen eingebunden werden können, selbst wenn die umschließende, eigene Anwendungen nicht läuft. Beispielsweise könnte eine Telefonbuch-Anwendung die „Wählscheiben"-Activity anzeigen und die ausgewählte Nummer anrufen lassen, sofern entsprechende Berechtigungen vorhanden sind. (Alliance (2011a),Becker u. a. (2010))

3.3.1.2 Service

Ein Service ist eine Androidkomponente, die selbst nach dem Schließen der zugehörigen Benutzeroberfläche weiter asynchrone Aufgaben ausführen kann.

3.3.1.3 Content Provider

Der *Content Provider* bildet die Persistenzschicht einer Android-Anwendung. Er kann jedoch die persistierten Daten nicht nur seiner eigenen, sondern auch definierten anderen Anwendungen zur Verfügung stellen. Falls eine Anfrage einer Applikation an den *Content Provider* kommt, wird dieser automatisch von der Android-Plattform gestartet.

3.3.1.4 Broadcast Receiver

Broadcast Receiver empfangen systemweite Nachrichten, die oft dazu genutzt werden den Anwendungen Änderungen im Systemzustand zukommen zu lassen. Ein Beispiel hierfür sind Informationen über den Ladestand des Akkus beziehungsweise ob ein Stromkabel angeschlossen ist. Jeder *Broadcast Receiver* wird automatisch beim Ankommen einer Systemnachricht gestartet und bei Bedarf wieder gestoppt.

3.3.2 Activity Stack

Während die Lebenszyklen der Komponenten *Broadcast Receiver* und *Content Provider* von der Android-Plattform selbst verwaltet, sprich gestartet und gestoppt werden, werden die Android-Komponenten *Activity* und *Service* von der zugehörigen Anwendung gestartet. Im Falle knapper Systemressourcen beendet jedoch die Android-Plattform gegebenenfalls diese (Android-)Komponenten. Dabei werden jedoch nur Komponenten beendet, die länger nicht angezeigt oder offenbar beschäftigungslos sind. Grundsätzlich landen alle gestarteten *Activities* auf einem *Activity Stack*. Beim Beenden einer *Activity* wird nun statt des Objekts selbst eine Referenz auf den Stack gelegt. Navigiert ein Benutzer nun auf eine vom System beendete *Activity*, wird diese Anhand der Referenz auf dem Stack wieder geladen. In der *Activity* eingegebene Formulardaten werden vor dem Beenden gespeichert und bei der Wiederherstellung geladen. (Becker u. a. (2010))

Kapitel 4

Verwandte Arbeiten

In diesem Kapitel sollen einige ausgewählte Arbeiten, die sich in einem ähnlichen Kontext bewegen, oder sich mit Teilproblem der vorgestellten Anforderungen der Arbeit beschäftigen, vorgestellt werden. Die Arbeiten sind dabei unterteilt in Arbeiten die sich mit QoS für Komponenten, und solchen, die sich mit selbst-adaptiven respektive reflexiven komponentenbasierten Systemen beschäftigen.

4.1 Selbst-adaptive Systeme

Selbst-adaptive Systeme sind in der Regel interdisziplinär, wobei die Kombination der Disziplinen stark vom Designansatz des selbst-adaptiven Systems abhängt. Die von Laddaga aufgeführten Designmetaphern (*control system*, *planning system*, *self-aware system*, vgl. Laddaga (1999),Laddaga (2000)) enthalten bereits drei Disziplinen: Entscheidungstheorie[14], Kontrolltheorie[15] und - übergeordnet - künstliche Intelligenz. Dazu kann man noch Software Engineering sowie Netzwerk- und verteile Systeme nennen. Dies unterstreicht in welch vielfältigen Disziplinen selbst-adaptive Systeme eingesetzt werden. Dementsprechend ist auch die damit verbundene Forschungslandschaft ein weites Feld. Folgend werden einige ausgewählte Richtungen und damit verbundene Arbeiten und Projekte vorgestellt.

Software Engineering
Software Engineering beinhaltet diverse Forschungsrichtungen, in denen selbst-adaptive Systeme eine Rolle spielen. Wie in Kapitel 2.3.2 erläutert gibt es einen starken Bezug zwischen den *Self-**-Eigenschaften und der Softwarequalität. Somit sind schon mal die Gebiete *Softwarequalität* und bei Qualität als nicht-funktionale Anforderung *Requirements Enginee-*

[14] Die Entscheidungstheorie behandelt die Evaluation der Konsequenzen von Entscheidungen.

[15] Die Kontrolltheorie betrachtet dynamische Systeme, deren Verhalten durch Eingangsgrößen von außen beeinflusst werden kann.

ring zu nennen. Kompositionelle Adaption basiert in der Regel auf komponentenbasierten Systemen, welche mittels *komponentenbasiertem Software-Engineering* entwickelt werden. *Aspektorientierte Programmierung* (AOP) wird oft verwendet um adaptives Verhalten zu realisieren, insbesondere in Verbindung mit *domänenspezifischen Sprachen* (DSL). Truyen und Joosen (2008) betrachten beispielsweise die Anwendbarkeit von AOP um festzustellen ob die AOP-Technologie eine Alternative für die Implementierung selbst-adaptiver Frameworks darstellt. Sie stellen fest, dass ein selbst-adaptives System typischerweise vier Phasen implementiert: Überwachen, Analysieren, Planen und Ausführen. Informelle Grundlage ist das sogenannte systemspezifische Adaptionswissen welches dabei hilft zu entscheiden, wann, wo, und wie das System verändert werden soll. AOP-Technologie kann auf unterschiedliche Weise die identifizierten Phasen unterstützen. Beispielsweise können *Pointcuts*[16] dazu verwendet werden eine Abfolge von Events zu erkennen und darauf zu reagieren. Nach ihren Ausführungen ist der größte Vorteil von AOP-Technologien jedoch, dass sie eine einheitliche Plattform zum Spezifizieren von Adaptionswissen mit domänenspezifischen Sprachen bereitstellen. Architektur-basierte Adaption ist laut Wu u. a. (2010) generell als vielversprechender Ansatz für selbst-adaptive Systeme akzeptiert. Sie schlagen dabei AOP für *cross-cutting* bzw. schichtenübergreifende Adaption vor. Zudem führen sie eine *Case Study* über die Implementierung einer selbst-adaptiven Software mit Hilfe eines reflexiven Komponentenmodells (*Fractal*) und dynamischem AOP innerhalb eines web-basierten Systems durch. Einige Projekte verwenden *Service-Orientierte Architekturen* (SOA) um ein adaptives Verhalten zu realisieren. Durch die lose Kopplung können Services leicht ausgetauscht werden. Rouvoy u. a. (2008) stellen eine Erweiterung zu dem bereits bestehenden Komponenten-Frameworks MADAM vor, welche insbesondere für eine service-orientierte Architektur im mobilen Umfeld geeignet sein soll. Services sind laut ihrer Aussage besonders gut für diese Domäne geeignet, da sie wiederverwendbare und zusammensetzbare Einheiten sind, die dynamisch genutzt werden können um den QoS-Ansprüchen unter sich ändernden Bedingungen gerecht zu werden. Die Middleware unterstützt dabei die Planung der Umkonfiguration, indem neu verfügbare Services untersucht werden, ob diese in einer alternativen Konfiguration der Anwendung eingesetzt werden kann, sprich die benötigten Funktionalitäten bietet. Diese Entscheidung ob ein Service geeignet ist basiert direkt auf den Service-Level-Agreements des Service Providers. Anhand eines Szenarios eines Vertreters auf Kundenbesuch demonstrieren sie, wie für eine beschränkte Anzahl von Services (z.B. Routenplanung) eine Umkonfiguration vorgenommen werden kann. Auslöser, für den Servicewechsel ist hier im wesentlichen die Änderung der Internetverbindung, beispielsweise von WLAN auf 3G, welches zudem als einziges reflexives Wissen dient. Zu dem weiteren Adaptionswissen gehören die SLA-Parameter der Services. Dabei werden insgesamt fünf numerische QoS-Parameter definiert: Die (monetären) Kosten, die Genauigkeit, das Detail beispielsweise in

[16] Im AOP ist ein *Pointcut* eine Menge von *Join-Points*. Wann immer ein Programmablauf einen der *Join-Points* erreicht, wird der zugewiesene Code (genannt *Advice*) ausgeführt.

Bezug auf einen Kartenservice, Neuheit der Information, und Batterieeinheiten die der Service verbraucht. Giner u. a. (2009) stellen den Einsatz selbst-adaptiver Software für Services im Bereich *Ambient Assistent Living (AAL)* vor. Sie schlagen eine Methode vor mit der man selbst-adaptive AAL-Services definieren kann, die sich ohne menschliches Zutun an die Benutzeranforderungen anpassen. Dabei setzen sie eine OSGi-basierte Infrastruktur ein um zu überprüfen, inwieweit die Idee im AAL-Kontext in der Realität eingesetzt werden kann. Neema und Ledeczi (2001) präsentieren eine Herangehensweise an selbst-adaptive Systeme, welche explizite Modelle des sog. *design-space* der Anwendung einsetzt. Dabei dürfen für jede Komponente auf jeder Ebene in der Modellhierarchie alternative Spezifizierungen vorliegen. Nicht-funktionale Anforderungen werden hierbei in Form von OCL-Constraints parametrisiert mit operationalen, zur Laufzeit gemessenen, Parametern hinterlegt. Beispiele für die Parameter sind Latenz, Fehlerrate, oder Genauigkeit. Immer dann, wenn eine neue Konfiguration alle Constraints erfüllt, wird eine Rekonfiguration vorgenommen. ReMMoC (Grace u. a. (2003)) ist, ähnlich wie MADAM eine dynamische Middleware die Interoperabilität zwischen mobilen Clients und Services unterstützt. Sie unterstützt die eine Rekonfiguration zur Laufzeit - jedoch nur um mit allen entdeckten heterogenen Services interoperieren zu können. Die Entwicklung basiert dabei auf der Idee der *Abstract Web Services*.

Künstliche Intelligenz

Die Entscheidungsfindung bzw. Entscheidungstheorie ist, wie erwähnt, ein wichtiger Bereich von adaptiven Systemen sowie eine Unterdisziplin der künstlichen Intelligenz. Norvig (1998) diskutiert die Rolle der Entscheidungstheorie in selbst-adaptiven Agenten in *„Decision Theory: The Language of Adaptive Agent Software“*. Neben der Entscheidungsfindung zählt er beispielsweise auch die Kontrolltheorie zu den Kerntechnologien adaptiver Software. Technologisch werden Softwareagenten oft in Multi-Agenten-Systemen eingesetzt um ein adaptives Verhalten zu realisieren. Wie beispielsweise Tesauro u. a. (2004), die eine dezentralisierte Architektur für autonomes Computing namens '"Unity“ vorstellen. Sie basiert dabei auf einer Multi-Agenten-Architektur, welche ermöglicht autonomes Verhalten bezüglich der *self-**-Eigenschaften *self-assembly*, *self-healing*, und *self-optimization* zu erreichen. Dabei präsentieren sie eine prototypische Implementierung wie sich eine Menge von *„Unity“*-Elementen in einer dynamischen, verteilten Anwendungsumgebung nach Anwendung diverser Fehlerklassen wieder erholt.

Netzwerk- und verteilte Systeme

Die Verhaltensänderung aufgrund von QoS-Parametern spielt bei Netzwerksystemen eine große Rolle und gilt als adaptives Verhalten. QoS-Parameter können jedoch auch auf Softwarekomponentenebene definiert sein und eine Rolle bei einem adaptiven System spielen. QoS-Parameter sind nicht-funktionale Anforderungen und können somit wieder auf die *Self-**-Eigenschaften bezogen werden. Auch abseits von QoS-Parametern spielt adaptives

Verhalten bei Netzwerktechnologien eine Rolle, wie beispielsweise in Streichert (2007), wo einige Methoden und Konzepte entwickelt werden, adaptives Verhalten auf Hardware- und Software-Ebene für Netzwerke zu realisieren. Capra u. a. (2003) beschreiben in ihrer Veröffentlichung *CARISMA*, eine Middleware für mobiles Computing, welche Reflexion verwendet, um mobile, kontextbewusste und adaptive Anwendungen zu ermöglichen. Die Middleware stellt Entwicklern mit Hilfe von *Policies* eine einfache Möglichkeit zur Verfügung zu beschreiben, wie auf Änderungen des externen Zustandes reagiert werden soll. Dabei klassifizieren sie unterschiedliche Typen von Konflikten, die beim Umgang mit diesen *Policies* beim mobilen Computing auftreten können.

4.2 Quality of Service und Komponenten

Im Rahmen dieser Arbeit ist *Quality of Service* in Verbindung mit Komponenten und selbstadaptiven Systemen interessant. Inbesondere wie man die QoS-Eigenschaften definiert, und welche für den Kontext in Frage kommen. Dem weitreichenden Forschungsfeld welches mit *QoS* verbunden ist, werden im folgenden nur Arbeiten vorgestellt, die sich zumindest anteilig mit diesen Herausforderungen beschäftigen. Zhou u. a. (2005) präsentieren in ihrer Veröffentlichung eine QoS-Anforderungskomponente, mit deren Hilfe Systeme mit unterschiedlichen QoS-Anforderungen gebaut werden können. Sie unterstreichen zudem, dass diese Technik auch verwendet werden kann, um dynamisch änderbare adaptive Systeme entwickeln zu können. Zu dieser Technik entwerfen sie ein Werkzeug, welches semi-automatisch den Entwickler bei der Komponentenparametrisierung unterstützen kann. Sie stellen fest, dass weitere Forschungsrichtungen aus ihrem Projekt hervorgehen können, wie beispielsweise die Entwicklung einer umfassenderen Regelbasis für den Automatismus der Komponentenparametrisierung, oder wie QoS-Daten sinnvoll mit der verbundenen Komponente gespeichert und diese eigentlich QoS-Daten genau definiert werden können. Gopalakrishna (2004) stellt einen Ansatz für die Spezifikation von QoS-Eigenschaften vor, der dem Komponentenentwickler ermöglicht, während des Designs der Komponenten QoS-Eigenschaften zu spezifizieren. Dabei wurden einfache Transformationsregeln entwickelt mit deren Hilfe QoS-Code generiert werden kann. Es wurde zudem ein *Proof of Concept* für das *UniGGen*-Framework entwickelt, mit dessen Hilfe Entwickler abhängig vom Einsatzszenario Parameter wählen können, aus denen anschließend passende Vorlagen für den benötigten QoS-Code generiert werden. Aagedal (2001) stellt in seiner Dissertation über QoS in verteilten Systemen die lexikalische Modellierungssprache CQML vor. Damit können QoS-Eigenschaften auf verschiedenen Abstraktionsstufen modelliert, und mit Hilfe eines UML-Profils in die Modellierung der Software, verwendet werden. Dabei können sowohl statische, als auch dynamische Aspekte modelliert werden. Die Modellierung findet hierbei jedoch ausschließlich auf Modellebene statt. Nahrstedt u. a. (2001) benennen in ihrer Veröffentlichung vier Kernaspekte einer QoS-bewussten Middleware: *QoS-Spezifikation* um eine Beschreibung des Anwendungsver-

haltens und der QoS-Parametern zu ermöglichen, *QoS-Übersetzung* und Kompilieren um spezifisches Anwendungsverhalten in Anwendungskonfigurationen für unterschiedliche Ressourcenzustände zu übersetzen, *QoS-Setup* um eine bestimmte Konfiguration auszuwählen und zu instantiieren. Sowie zuletzt *QoS-Adaption* um die Anwendung an Ressourcenfluktuation anpassen zu können. Sie präsentieren anschließend eine eigene QoS-Middleware Architektur, welche jedoch externe Parameter benötigt, die zur Laufzeit miteinbezogen werden. Während der Laufzeit wird dann das QoS-Setup und die Adaption durchgeführt. Das Setup findet dabei kurz vor der Ausführung der Anwendung statt, während die Adaption zur Laufzeit von Ressourcenfluktuation, Benutzermobilität, und dem Ändern von Benutzerpräferenzen angestossen wird. Die Spezifikationen von QoS-Parametern haben laut ihrer Aussage folgende Eigenschaften gemeinsam: sie sind anwendungs- und domänenspezifisch und benötigen eine Übersetzung von den ursprünglichen Anwendungsebenen-Notationen zu *system-level*-QoS-Parametern und Repräsentationen. Grassi u. a. (2007) untersuchen die Analyse von komponentenbasierten Systemen hinsichtlich nicht funktionaler Anforderungen. In diesem Rahmen empfehlen sie den Einsatz von sogenannten Performability-Modellen, welche insbesondere die Anforderungen von und Wechselwirkungen zwischen Leistungsfähigkeit (Performance) und Verlässlichkeit (Reliabilty) abbilden können. Dazu bauen sie auf der bereits bestehende Beschreibungssprache *KLAPER* (Kernel Language for Performance and Reliabilty) auf. Sie untersuchen somit die modellbasierte Analyse für das frühe Zuweisen von QoS-Attributen zu einem komponentenbasierten System noch vor der eigentlichen Entwicklung. Sie schlagen vor, nach der Implementierung *model-based reasoning* für die Vorhersage der Auswirkungen einer Rekonfiguration einzusetzen. QoS-bewusste Systeme sind Systeme, die wissen welchen QoS-Grad sie von anderen Diensten benötigen und gleichzeitig welchen sie zur Verfügung stellen können. Laut Frolund und Koistinen (1998) sind die wichtigsten Herausforderungen bei der Entwicklung eines *QoS*-bewusstem System die Frage, wie man die QoS-Anforderungen und Eigenschaften ausdrücken kann, und wie man diese Kommunizieren kann. Sie stellen fest, dass in realistischen System diese Anforderungen schnell sehr komplex werden können. Um dieser Komplexität mit geeignetem Werkzeug zu begegnen, entwickelten sie eine Spezifikationssprache mit begleitender Laufzeit-Repräsentation für QoS-Ausdrücke. Als Beispiel zeigen sie wie man dynamisch neue Ausdrücke erstellen kann und wie diese untereinander verglichen werden können.

4.3 Einordnung dieser Arbeit

Folgend werden die obig genannten Arbeiten tabellarisch aufgeführt um die Übersicht zu gewährleisten und anschließend die vorliegende Arbeit eingeordnet.

Autoren	Überblick
Truyen und Joosen (2008)	AOP für die Implementierung selbst-adaptiver Frameworks.
Wu u. a. (2010)	Schichtenübergreifende Adaption mit einem Komponentenmodell und dynamischem AOP.
Rouvoy u. a. (2008)	Erweiterung des Komponenten-Frameworks MADAM um Adaption durch Servicekomposition.
Giner u. a. (2009)	Selbst-adaptive Software für AAL-Umgebungen. Definition von autonomen, selbst-adaptiven AAL-Services. Entwurf mit OSGi.
Neema und Ledeczi (2001)	Adaptive Systeme mit auf jeder Ebene der Modellhierarchie abgelegten alternativen Spezifikationen der Komponenten. Nicht-funktionale Anforderungen werden hierbei in Form von OCL-Constraints parametrisiert, und mit operationalen, zur Laufzeit gemessenen, Parametern hinterlegt.
Grace u. a. (2003)	Stellen eine dynamische Middleware (ReMMoC) vor, die dynamische Rekonfiguration einsetzt, um mit allen entdeckten heterogenen Services interoperieren zu können. Basiert auf der Idee der *abstract web services*.
Norvig (1998)	Entscheidungstheorie in selbst-adaptiven Agenten.
Tesauro u. a. (2004)	Dezentralisierte Architektur für autonomes Computing (‘”Unity“). Sie basiert dabei auf einer Multi-Agenten-Architektur, welche ermöglicht autonomes Verhalten, insbesondere bzgl. *self-healing* zu erreichen.
Streichert (2007)	Methoden und Konzepte für die Realisierung adaptiven Verhaltens auf Hardware- und Software-Ebene für Netzwerke.

Tabelle 4.1: Zusammenfassung: verwandte Arbeiten - Teil 1

Capra u. a. (2003)	Middleware für mobiles Computing um kontextbewusste Anwendungen zu ermöglichen. Entwickler können *Policies* hinterlegen, wie (regelbasiert) auf Umgebungsänderungen reagiert werden soll.
Zhou u. a. (2005)	Präsentieren eine QoS-Anforderungskomponente, mit deren Hilfe Systeme mit unterschiedlichen QoS-Anforderungen gebaut werden können. Ausblick auf den Einsatz dieser Technik in adaptiven Systemen.
Gopalakrishna (2004)	Ansatz für die Spezifikation von QoS-Eigenschaften, der dem Komponentenentwickler ermöglicht, während des Designs der Komponenten QoS-Eigenschaften zu spezifizieren. Dabei wurden zudem einfache Transformationsregeln entwickelt mit deren Hilfe QoS-Code generiert werden kann.
Nahrstedt u. a. (2001)	Richtlinien für eine QoS-bewusste Middleware mit vier Kernaspekten: *QoS-Spezifikation*, *QoS-Übersetzung*, *QoS-Setup* und *QoS-Adaption*.
Grassi u. a. (2007)	Untersuchen die Analyse von komponentenbasierten Systemen hinsichtlich nicht funktionaler Anforderungen. Empfehlen Performability-Modelle, welche insbesondere die Anforderungen von und Wechselwirkungen zwischen Leistungsfähigkeit (Performance) und Verlässlichkeit (Reliabilty) abbilden können.
Frolund und Koistinen (1998)	Schlagen eine allgemeine QoS-Spezifikations-Sprache vor, welche QML genannt wird. Zeigen, wie QML benutzt werden kann um QoS-Eigenschaften als Teil des Designs festgehalten werden können.

Tabelle 4.2: Zusammenfassung: verwandte Arbeiten - Teil 2

Die verwandten Arbeiten haben bereits eine Reihe von vielversprechenden Resultaten geliefert. Dabei wurde die Fähigkeit der Selbst-Adaption in unterschiedlichen Richtungen, wie Requirements-Engineering, Software-Architektur, Middleware, und komponentenbasierter Entwicklung, untersucht. Es wurde jedoch deutlich, dass diese Gebiete meist nur isoliert betrachtet wurden, und die verwandten Arbeiten meist nur einen Teilbereich der vorliegenden Arbeit tangieren. Zudem unterscheidet sich die Herangehensweise dieser Arbeit von der der vorgestellten Arbeiten: In dieser Arbeit sollen, basierend aus der mit der Motivation verbundenen Problemstellung, passende adaptive Mechanismen ausgewählt werden, die zur Bewältigung dieser Problemstellung beitragen. Im Vorfeld ist dabei nicht klar, welche Mechanismen dies sein werden. Die in diesem Kapitel betrachteten verwandten Arbeiten gehen meist den Weg, zunächst einen oder mehrere Mechanismen auszuwählen, (wie AOP oder Policies) um dann deren Eignung in einem Kontext zu überprüfen. Diese Arbeit schlägt zudem die Brücke zwischen den Gebieten der Qualitätseigenschaften für Komponenten und der Optimierung mobiler Systeme durch Adaption. Lediglich zwei Arbeiten (ReMMoC und die Erweiterung von MADAM) weisen ähnliche Ziele auf wie die vorliegende. ReMMoC unterscheidet sich dabei jedoch insbesondere durch das Ziel per Rekonfiguration heterogene Schnittstellen unterschiedlicher Serviceanbieter ansprechen zu können. QoS-Eigenschaften werden dabei nur auf Netzwerkebene betrachtet - kontextbasierte Benutzeranforderungen spielen keine Rolle. Die Erweiterung von MADAM, konzentriert sich dabei auf die Komposition lokaler und entfernter Services auf Basis von SLA-Agreements. Der Ressourcenkontext wird bei der Adaptionsentscheidung nicht berücksichtigt - auch der Benutzerkontext bleibt konstant. Im Unterschied zur vorliegenden Arbeit unterscheiden sich die Szenarios dabei vor allem in der Art der angebotenen Services, welche unterschiedliche SLAs garantieren, und nicht auf Basis der durch das Sammeln von Kontextdaten des mobilen Geräts. Zunächst soll die Adaption in dieser Arbeit aber nur mit lokal stets verfügbaren Hilfsmitteln geschehen. MADAM widmet sich somit eher der Servicekomposition, anstatt der Rekonfiguration mit Hilfe eines dynamischen Komponentenframeworks.

Kapitel 5

Konzept

In diesem Kapitel wird das entwickelte Konzept zur adaptiven, dynamischen Rekonfiguration einer mobilen Anwendung vorgestellt. Auf Basis der Grundlagen werden Entwurfsentscheidungen erläutert und begründet. Das angewandte Vorgehen findet der Leser im folgenden Abschnitt.

5.1 Vorgehen

Zunächst wird im nächsten Abschnitt 5.1.1 ein realitätsnahes Szenario vorgestellt, welches die Motivation verdeutlicht und zugleich als Basis der Modelle des Konzepts dient. Anschließend werden die Anforderungen erhoben, die der Benutzer in diesem Szenario an das mobile System stellt, auf deren Basis die *Self-**-Eigenschaften des adaptiven Systems ausgewählt werden. Durch Auswahl der passenden *Self-**-Eigenschaften, sollen die dazu korrelierenden Qualitätseigenschaften die Anforderungen des Benutzers erfüllen. Auf Basis der in den Grundlagen vorgestellten Klassifizierungen wird erörtert, welche technologischen und adaptiven Eigenschaften geeignet sind diese umzusetzen und welche Auswirkungen dies auf das Verhalten der Anwendung hat. Dabei wird insbesondere der mobile Kontext der Anwendung berücksichtigt. Zu guter Letzt wird eine der technologischen Klassifizierung und den QoS-Modellen entsprechende Architektur für den darauf folgenden Entwurf vorgestellt.

5.1.1 Szenario

Wie in der Einleitung des Kapitels beschrieben, soll ein Konzept für die Entwicklung eines sich an wechselnde Umstände anpassenden Systems entwickelt werden. Dabei ist es nötig ein beispielhaftes Szenario zu definieren, was einerseits als Motivation, und andererseits als *informelle* Grundlage der Anforderungen dient. Das Szenario sollte dabei möglichst viele Aspekte einer mobilen Anwendung abdecken, damit im Konzept genügend Freiraum für das Demonstrieren der Möglichkeiten bleibt.

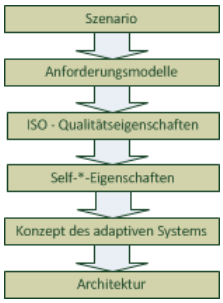

Abbildung 5.1: Herleitung der Architektur

„Ein Unternehmen für Schiffszertifizierungen beschäftigt eine größere Anzahl Gutachter, die einen Großteil der Woche auf Werften diverse Schiffe auf Einhalten einer bestimmten Norm prüft. Über die Anwendung auf dem Tablet-PC wird dabei ein Prüfbericht angefertigt und eine Reihe von Fotos aufgenommen, welche die einzelnen Punkte dokumentieren. Nach der Begehung übermittelt der Gutachter den Prüfbericht inklusive der aufgenommenen Fotos an den Hauptsitz des Unternehmens. Die relevanten technischen Ressourcen seines mobilen Endgeräts sind hierbei der Akku, die Qualität der Internetverbindung, die Rechenstärke, sowie der verfügbare Hauptspeicher.Vor Ort läuft das Gerät bei der Schiffsbegehung im Akkubetrieb. Falls eine Internetverbindung besteht verfügt diese typischerweise nur über geringe Bandbreite. Nach Abschluss der Prüfung will der Gutachter eine erste Version des Prüfberichts übermitteln. Dabei erwartet er, dass dies in einer angemessenen Geschwindigkeit von statten geht und zudem der Akku seines Endgerätes nicht zu sehr belastet wird. Da der Gutachter bei einer Schiffsbegehung manchmal keinen Empfang hat, will er gegebenenfalls abends vom Hotel aus seinen Bericht versenden. Die Netzanbindung ist immer noch nicht optimal, jedoch wurde das Endgerät an das Stromnetz angeschlossen, sodass eine aufwändigere Komprimierung durchgeführt werden kann. Seine Zeitanforderung bleibt jedoch bestehen. Wieder in der Firmenzentrale angekommen, will der Gutachter seine endgültige Version des Berichts versenden und legt dabei besonders Wert auf eine gute Bildqualität. Durch eine Breitband-Internetverbindung und den Netzbetrieb seines Endgeräts ist zudem eine optimale Bandbreite vorhanden und der Akkustand nicht relevant. Im Falle eines geringen Akkustandes bei einer Schiffsbegehung wird zunächst nur der Prüfbericht ohne Bilder versendet um den Akku nicht noch zusätzlich für eine größere Datenübertragung zu belasten. Sofern bei der Übermittlung des Prüfberichts kein Internet vorhanden ist, werden die Bilder mit einer durchschnittlichen Kompression komprimiert und gespeichert um eine Übermittlung schnellstmöglich durchführen zu können. Falls durch andere Anwendungen die Leistungsfähigkeit des Endgeräts sehr ausgelastet ist, sodass eine rechenintensive Operation ein sehr schlechtes Zeitverhalten aufweisen würde, sollen aufwändige Berechnungen vermieden werden.“

5.1.2 Anwendungsfälle

Die textuelle Beschreibung des Szenarios dient als Grundlage der folgenden Anwendungsfälle. Es muss darauf hingewiesen werden, dass die Anwendungsfälle nicht auf die Erfüllung des funktionalen Ziels, der Übermittlung des Prüfberichts abzielen, sondern auf die Rekonfiguration auf Basis der Kontexte. Dem Szenario werden also die Ziele, der Standardablauf, sowie möglicherweise eintretende Sonderfälle (hier: allgemeine Szenarien), die das Erreichen des Ziels verhindern oder abändern können, entnommen. Auf Basis dieser systematisch aus dem Szenario abgeleiteten Informationen soll anschließend ein Anforderungsmodell für die Implementierung der Funktionalitäten und passenden QoS-Parameter erstellt werden. Zudem kann auf Basis dieser Analyse bestimmt werden, welche Qualitäts- und welche *self-**--Eigenschaften das System benötigt. Daraus kann dann bestimmt werden welche Designentscheidungen bezüglich des adaptiven Systems zu treffen sind. Anschließend werden weitere nicht-funktionale Anforderungen an das Konzept definiert.

Name	Fachliches Szenario 1 „Werft“
Beschreibung	Das System erkennt dass sich der Benutzer auf der Werft befindet und passt sich entsprechend an
Ziel	Das System soll für den Kontext Werft optimal konfiguriert sein
Vorbedingungen	Schlechte Internetverbindung, Akkubetrieb
Standardablauf	1. Das System erkennt den Kontext „Werft“ 2. Das System sucht aus seinen Funktionsvarianten die passendste aus 3. Das System entscheidet, ob sich die Rekonfiguration lohnt 4. Das System rekonfiguriert sich gegebenenfalls
Sonderfälle	Das System erkennt bei Schritt 2, dass bereits die optimale Funktionalität eingestellt ist / die Anwendungsfälle „Geringer Batteriestand“/“Kein Internet“/“System ausgelastet“ sind eingetreten

Tabelle 5.1: Anwendungfall: Fachliches Szenario 1 „Werft“

Name	Fachliches Szenario 2 „Hotel“
Beschreibung	Das System erkennt dass sich der Benutzer im Hotel befindet und passt sich entsprechend an
Ziel	Das System soll für den Kontext „Hotel“ optimal konfiguriert sein
Vorbedingungen	Mittelmäßige Internetverbindung, Anschluss ans Stromnetz
Standardablauf	1. Das System erkennt den Kontext „Hotel“ 2. Das System sucht aus seinen Funktionsvarianten die passendste aus 3. Das System entscheidet, ob sich die Rekonfiguration lohnt 4. Das System rekonfiguriert sich gegebenenfalls
Sonderfälle	Das System erkennt bei Schritt 2, dass bereits die optimale Funktionalität eingestellt ist / die Anwendungsfälle “Kein Internet“/“System ausgelastet“ sind eingetreten

Tabelle 5.2: Anwendungfall: Fachliches Szenario 2 „Hotel“

Name	Fachliches Szenario 3 „Büro“
Beschreibung	Das System erkennt dass sich der Benutzer im Büro befindet und passt sich entsprechend an
Ziel	Das System soll für den Kontext „Büro“ optimal konfiguriert sein
Vorbedingungen	Gute Internetverbindung, Anschluss ans Stromnetz
Standardablauf	1. Das System erkennt den Kontext „Büro“ 2. Das System sucht aus seinen Funktionsvarianten die passendste aus 3. Das System entscheidet, ob sich die Rekonfiguration lohnt 4. Das System rekonfiguriert sich gegebenenfalls
Sonderfälle	Das System erkennt bei Schritt 2, dass bereits die optimale Funktionalität eingestellt ist / die Anwendungsfall Systemlast ist eingetreten

Tabelle 5.3: Anwendungfall: Fachliches Szenario 3 „Büro“

Name	Allgemeines Szenario A „Geringer Batteriestand“
Beschreibung	Der Batteriestand des mobilen Endgeräts ist gering
Ziel	Das System soll möglichst ressourcensparend agieren
Vorbedingungen	Geringer Batteriestand (Akkustand <15 Prozent)
Standardablauf	1. Das System erkennt den geringen Batteriestand 2. Das System sucht aus seinen Funktionsvarianten die ressourcen-sparendste aus 3. Das System entscheidet, ob sich die Rekonfiguration lohnt 4. Das System rekonfiguriert sich gegebenenfalls
Sonderfälle	Das System erkennt bei Schritt 2, dass bereits die optimale Funktionalität eingestellt ist / der Anwendungsfall Systemlast ist eingetreten

Tabelle 5.4: Anwendungfall: Allgemeines Szenario A „Geringer Batteriestand“

Name	Allgemeines Szenario B „Keine Internetverbindung“
Beschreibung	Es ist keine Internetverbindung vorhanden
Ziel	Das System soll angemessen reagieren
Vorbedingungen	Keine Internetverbindung
Standardablauf	1. Das System erkennt dass keine Verbindung vorhanden ist 2. Das System sucht aus seinen Funktionsvarianten eine aus, die nicht auf eine Internetverbindung angewiesen ist 3. Das System rekonfiguriert sich
Sonderfälle	-

Tabelle 5.5: Anwendungfall: Allgemeines Szenario B „Keine Internetverbindung“

Name	Allgemeines Szenario C „System ausgelastet“
Beschreibung	Das System ist ausgelastet
Ziel	Das System soll dem Benutzer eine angemessene Geschwindigkeit bieten können. Daher soll es versuchen durch ressourcensparende Funktionsweise die Auslastung zu verringern
Vorbedingungen	System ausgelastet (Last >95 Prozent)
Standardablauf	1. Das System erkennt die hohe Auslastung 2. Das System sucht aus seinen Funktionsvarianten eine aus, die besonders ressourcenschonend bezüglich der CPU-Last ist 3. Das System entscheidet, ob sich die Rekonfiguration lohnt 4. Das System rekonfiguriert sich gegebenenfalls
Sonderfälle	Anwendungsfall „Geringer Akkustand“ oder „Keine Internetverbindung ist eingetreten“

Tabelle 5.6: Anwendungfall: Allgemeines Szenario C „System ausgelastet“

Diese Anwendungsfälle stellen Situationen dar, die sich durch bestimmte Eigenschaften der Umgebung oder aus den Zuständen des Endgeräts definieren. Dabei lassen sich diese Eigenschaften als *Umgebungs-* und *Ressourcenkontext* bezeichnen. Der Benutzer hat zu jeder Kontextkombination andere Anforderungen an das funktionale und nicht-funktionale Verhalten der Anwendung. Dabei spricht man dann von dem *Benutzerkontext*.

Die *fachlichen Szenarien* eint, dass nur ein Szenario gleichzeitig auftreten kann. Die Anwendung befindet sich aufgrund der Eigenschaften der Umgebung beziehungsweise des Geräts immer in Szenario A, B, oder C. Die Sonderfälle (hier: *allgemeine Szenarien*) können jedoch zusätzlich zu einem anderen Szenario auftreten, da sie von vielen anderen spontanen Faktoren abhängen, die nicht zu berücksichtigen sind. Beispielsweise ergibt sich die Systemlast aus anderen laufenden Anwendungen und Leistungsfähigkeit der Hardware. Da das Eintreffen der allgemeinen Szenarien in der Regel die Gebrauchstauglichkeit stark belastet oder potentiell bedroht (mit leerer Batterie oder ausgelastetem System gar keine Funktion), können diese die Anwendungsfälle der *fachlichen Szenarien* überlagern und haben somit eine höhere Priorität.

5.1.3 Anwendungstyp

Im vorigen Kapitel wurde ein Szenario vorgestellt, für das das Anwendungskonzept besonders geeignet ist. Natürlich lässt sich die Aussage über Art von Anwendung, auf die das Konzept abzielt, weiter präzisieren. Die nachfolgende Liste beschreibt die allgemeinen Eigenschaften des Anwendungstypen, für welchen das Konzept geeignet ist.

- Mobilität: Die Anwendung sollte in unterschiedlichen, wechselnden Umgebungen, sprich nicht nur im Büro oder nur zu Hause, ausgeführt werden. Die inneren und äußeren Ressourcen verändert sich dabei, so dass es einen Anlass für das adaptive Verhalten gibt.

- Skalierbarkeit: Die Funktionsgüte, beispielsweise die Präzision des Ergebnisses einer Berechnung sowie der Ressourcenverbrauch, sollen sich durch die Implementierung beeinflussen lassen. Beispielsweise wenn man durch den Einsatz von mehr Ressourcen die Präzision des Ergebnisses oder die Geschwindigkeit der Funktion verbessern kann. Je mehr es möglich ist Teile der Anwendung in Bezug auf Performanz, Qualität, oder Verbrauchsverhalten zu optimieren, desto größer ist der Nutzen des Konzepts.

- Wechselnde Benutzeranforderungen: Innerhalb dieses Konzepts wird voraus gesetzt, dass der Benutzer in unterschiedlichen Situationen und Umgebungen unterschiedliche Anforderungen an die Anwendung hat. Dies wird in die Entscheidung ob eine Rekonfiguration durchgeführt werden soll mit einbezogen. Prinzipiell ist es zwar möglich das

Konzept mit gleichbleibenden Benutzeranforderungen einzusetzen, jedoch sinkt dadurch der Nutzen.

5.1.4 Optionale Anforderungen

Das Erfüllen der Anwendungsfälle ist die wichtigste Anforderung. Falls es nicht möglich ist die Ziele mit Hilfe des Konzepts zu erreichen, ist das Konzept nicht für diese Art von Anwendung geeignet. Abgesehen von den Anwendungsfällen existieren jedoch noch weitere, optionale, aber nicht unwichtige Anforderungen an eine solche Anwendung. Diese haben eher nicht-funktionalen Charakter, und sind für die Frage nach der Praxistauglichkeit besonders wichtig.

Angemessenheit	Eine Rekonfiguration ist eine Aktion, deren Durchführung Ressourcen verbraucht. Es ist zu beachten, dass die Kosten der Rekonfiguration nicht den erwarteten Nutzen übersteigen. Dazu gehört auch, dass die Häufigkeit der Kontextwechsel beachtet wird, da sich dadurch die Rekonfigurationskosten summieren.
Robustheit	Darunter ist die Robustheit gegenüber nicht vordefinierten Szenarien zu verstehen. Die Anwendung muss auf jedes eintretende Szenario angemessen reagieren können. Beispielsweise wenn im Szenario „Hotel“, unerwartet doch eine gute Internetverbindung vorhanden ist, sollte die Anwendung eine für diese Variante passende Funktionalität auswählen können.
Erweiterbarkeit	Es sollte möglich sein, die Anwendung auch nach Auslieferung um Komponenten zu erweitern, die die benötigte Funktionalität sowie die Qualitätseigenschaften der Komponente bereit stellen. Wenn beispielsweise die Möglichkeiten der Rekonfiguration zur Optimierung des Verbrauchsverhaltens und zur Optimierung der Funktionsgüte vorhanden sind, sollte man die Anwendung noch um die Möglichkeit zur Optimierung der Performanz erweitern können.
Transparenz	Die Rekonfiguration durch die adaptiven Mechanismen sollte autonom, sprich ohne Eingreifen des Benutzers geschehen. Der Mechanismus der Rekonfigurationsentscheidung und der Rekonfiguration soll dabei transparent sein, und vom Benutzer nicht bewusst wahrgenommen werden.

Tabelle 5.7: Nicht-funktionale Anforderungen an das Konzept

5.2 Architektur

Nachdem nun die Szenarien und Anforderungen definiert sind, soll ein Konzept entwickelt werden, welches die geänderten Umgebungs- und Gerätezustände (beziehungsweise die Kontexte) erkennt und passend auf die geänderten Anforderungen reagiert. Es muss definiert werden, welche grundsätzlichen adaptiven Eigenschaften die Anwendung haben muss. Daraus resultiert ein technologisches Profil und eine Architektur. Zudem müssen Modelle für die Umgebungs- und Gerätezustände sowie für die Anforderungen des Benutzers für den jeweiligen Kontext erstellt werden. Anhand der Anforderungen wird jedoch zunächst bestimmt, welcher Typ von adaptivem System diesen am ehesten entspricht.

5.2.1 Self-*-Eigenschaften und Qualitätseigenschaften

Auf Basis der Anforderungsanalyse wurde bestimmt, welche Qualitätseigenschaften die funktionalen und nicht-funktionalen Anforderungen des Benutzers (siehe 5.2.3.3) erfüllen. Im wesentlichen wurden die drei Eigenschaften *Zeitverhalten*, *Verbrauchsverhalten*, und *Effizienzkonformität* aus dem Bereich Effizienz und *Angemessenheit* sowie *Richtigkeit* aus dem Bereich Funktionalität identifiziert. Zeitverhalten und Verbrauchsverhalten, bedeutet eine situationsabhängige Steigerung der Leistungsfähigkeit und Senken des Verbrauchs, was für die Eigenschaft *self-optimizing* steht. Das Anpassen der Funktionalität an aktuelle Gegebenheiten passt hingegen zur Eigenschaft *self-configuring*. Somit ergeben sich die folgenden Eigenschaften, welche mit Hilfe des Konzepts umsetzbar sein müssen. (siehe 2.1.2)

- *self-optimizing*
- *self-configuring*
- *self-awareness/self-monitoring* (resultiertend aus den obigen Eigenschaften)

Auf Basis dieser Eigenschaften können nun technologische und architektonische Entscheidungen getroffen werden, um die resultierenden adaptiven Mechanismen umsetzen zu können.

5.2.2 Technologisches Profil und Architekturentscheidungen

Nicht jeder adaptive Mechanismus ist dafür geeignet die oben definierten Eigenschaften hinreichend zu unterstützen. In Kapitel 2.1 wurden Klassifizierungen der unterschiedlichen Arten von adaptiven Verhalten beschrieben. Diese werden nun herangezogen und vor dem Hintergrund der benötigten Eigenschaften diskutiert und ausgewählt.

Adaptionsverhalten (antizipiert/nicht-antizipiert) (Kapitel 2.1.1)
Die Anforderungen an das System beschreiben ein System, welches anhand von festgelegten Eigenschaften der Umgebung und des eigenen Zustands eine Rekonfiguration vornehmen kann. Es sind somit nicht alle, die Adaptionsentscheidung beeinflussenden, Parameter a priori bekannt. In diesem Fall kann man zwar nicht davon sprechen dass alle Varianten und Kombinationen der Kontexte bekannt sind, aber man das passendste Szenario anhand der Kontexte auswählen kann. Deshalb kann man dieses Adaptionsverhalten dennoch *nicht-antizipiert* nennen.

Adaptionsebene (parametrisch/kompositionell) (Abschnitt 2.1.5.1)
Das Optimieren der Gebrauchstauglichkeit bzw. das selbstständige Rekonfigurieren und auch die flexible Veränderbarkeit der eigenen Funktionalität bedarf einer mächtigen und flexiblen Adaptionsarchitektur. Die rein *parametrische Adaption* ist somit für diese Ansprüche nicht ausreichend. Somit wird die *kompositionelle* bzw. *architekturbasierte* Adaption gewählt.

Adaptionsentscheidung (regelbasiert/zielorientiert/nutzenor.) (Kapitel 2.1.5.2)
Situation-Aktions-Regeln eigenen sich nur bedingt für die Implementierung dynamischer Adaption. Eine Zielorienierte Adaptionsentscheidung bedarf eines konkret messbaren Ziels, wie beispielsweise die Anzahl der übertragenen Bilder pro Sekunde um bewerten zu können, inwiefern das Ziel erreicht wurde. In der aktuellen Arbeit wird ein Szenario betrachtet, in dem eine größere Anzahl Parameter zu einer Bewertung der Qualität herangezogen werden. Das Ziel ist hierbei das Optimieren der Qualiät in dem jeweiligen Kontext - somit müssen die Alternativen einzeln bewertet und miteinander verglichen werden können. Dazu eignet sich die *nutzenorientierte* Adaption optimal, da durch eine Nutzenfunktion Vergleichbarkeit geschaffen wird.

Auswirkungsvorhersage (stark/schwach) (Kapitel 2.1.5.3)
Im Zuge der Arbeit sollen mit Hilfe der *kompositionellen* Adaption Komponenten ersetzt, verändert, oder entfernt werden können. Dies entspricht einer starken Adaption. Um die potentiell umfangreichen Auswirkungen auf ein Softwaresystem durch solche architektonischen Einschnitte abschätzen und in die Adaptionsentscheidung miteinbeziehen zu können, benötigt man eine Instanz, die fähig ist die Auswirkungen auf das System zu beurteilen.

Adaptionszeitpunkt 1 (synchron/asynchron) (Kapitel 2.1.5.4)
Bei synchroner Adaption findet die Rekonfiguration mit bzw. nach dem Aufruf einer Funktion statt. Sie eignet sich deshalb im wesentlichen für Adaptionen, die von der Art des Aufrufs beeinflusst werden und die Auswirkungen auf kommende Aufrufe haben soll. In dem Szenario dieser Arbeit soll jedoch jeder Aufruf auf einem optimal für die aktuelle Umgebung konfiguriertem System ausgeführt werden. Die Adaption ist somit vom Umgebungs- und

Ressourcenkontext abhängig und muss auf Änderungen dieser Kontexte reagieren, welches für einen asynchronen Adaptionszeitpunkt spricht.

Adaptionszeitpunkt 2 (proaktiv/reaktiv) (Kapitel 2.1.5.4)
Grundsätzlich ist im Rahmen dieser Arbeit der *reaktive* Ansatz vorzuziehen, da die Szenarienwechsel eines Benutzers schwer vorherzusagen sind. Zudem würde eine *proaktive* Rekonfiguration bedeuten, dass man in dem alten Szenario eine gewisse Zeit mit einer aktuell nicht optimalen Konfiguration arbeiten muss. Bei einer nutzenorientierten Adaption spielt zudem auch die Ausprägung der einzelnen Parameter eine Rolle, die in der Realität immer gewissen Schwankungen unterliegt. Eine *proaktive* Adaption macht nur dann Sinn wenn eine Adaption einer gewissen Vorbereitung bedarf, die jedoch den aktuellen Ablauf nicht beeinflusset. Zudem muss der Adaptionszeitpunkt recht exakt vorherzusagen sein, und der Umstand dass die veränderte Konfiguration ohne Verzögerung in dem neuen Szenario zur Verfügung steht einen großen Nutzen haben.

Damit sind die generellen Eigenschaften der Anwendung als adaptives System ermittelt. Nun lassen sich auch die benötigten Modelle des Umgebungskontexts, des (Geräte-)Ressourcen-Kontexts, der Benutzeranforderungen, und die Eigenschaften der Komponenten, erstellen.

5.2.3 Szenariomodell

Innerhalb des Konzepts soll eine *nutzenorientierte* Adaptionsentscheidung getroffen werden. Dafür werden die Faktoren, die mit einbezogen werden sollen, in vergleichbarer und vorgegebener Form verwendet. Deshalb benötigen wir Szenariomodelle mit ordinal- oder intervall-skalierten Werten um die Ausprägungen der einzelnen Zustände mit denen der Benutzeranforderungen zu vergleichen. Die *nutzenorientierte* Adaptionsentscheidung aggregiert alle vorhandenen Daten zu einer Kennzahl, die zur Bewertung der jeweiligen Komponente dient. Das Anforderungsmodell eines Szenarios ist dabei ein Quadrupel bestehend aus dem aktuellen Systemzustand, dem Umgebungszustand, den Anforderungen des Benutzers und den Qualitätseigenschaften der Komponenten. In den folgenden Abschnitten werden für jede dieser Kategorien Anforderungsmodelle erstellt, die aufgrund der Ordinal- oder Intervallskalierung vergleichbar sind.

Quadrupel des Szenariomodells:
$Szenario(Kontext_{Umgebung}, Kontext_{Ressourcen}, Kontext_{Benutzer}, Qual_{Komponenten})$

5.2.3.1 Umgebungskontext

Der Umgebungskontext stellt sich in diesem Szenario über die angebotene Bandbreite ($Umgebung_{inet}$) dar. In anderen Anwendungsszenarien könnten auch weitere messbare Parameter wie GPS-Position oder Temperatur relevant sein. Für jedes Szenario wird ein Wert auf einer Skala von 0-3 gewählt, der den jeweilige Umgebungskontext beschreibt. Textuell stehen die Werte für keine/wenig/mittlere/hohe Bandbreite. Bei den allgemeinen Szenarien A und C beschreibt „beliebig“, dass diese Szenarien mit jeder Ausprägung dieses Merkmals auftreten können, es aber für die Bewertung des Szenarios nicht relevant ist. Das resultierte Modell lässt sich mit $Kontext_{Umgebung}(Umgebung_{Inet})$ beschreiben.

Szenario	$Umgebung_{inet}$
Fachl. Szenario 1 „Werft“	niedrig (1)
Fachl. Szenario 2 „Hotel“	mittel (2)
Fachl. Szenario 3 „Büro“	hoch (3)
Allg. Szenario A „Akku“	beliebig (X)
Allg. Szenario B “Internet“	keine (0)
Allg. Szenario C „System“	beliebig (X)

Tabelle 5.8: $Kontext_{Umgebung}(Umgebung_{Inet})$

5.2.3.2 Ressoucenkontext

Der Ressourcenkontext beschreibt, welche veränderbaren technologischen Parameter des Endgeräts die Adaptionsentscheidung beeinflussen können. Sich ändernde Systemzustände sind mitverantwortlich für die Notwendigkeit einer Rekonfiguration zur Laufzeit und werden auch *self* genannt. Für den Batteriestand ($Ressourcen_{strom}$) wird für jedes Szenario ein Wert auf einer Skala von 0-2 gewählt. Diese Werte stehen für niedrigen Batteriestand (0), Batteriebetrieb (nicht niedrig) (1), und Netzbetrieb (2). Die mit „beliebig“ gekennzeichneten Werte sind wieder nicht relevant für die Bewertung des Szenarios. Der auch eintretende Zustand eines leeren Akkus wird hier nicht berücksichtigt, da mit leerer Batterie auch keine Rekonfiguration durchgeführt werden kann. Die Systemauslastung ($Ressourcen_{last}$) wird vereinfacht nur mit Null oder Eins bemessen. Die Null bedeutet geringe oder normale Auslastung und die Eins bedeutet hohe Auslastung. Falls die Last keine Rolle spielt wird der Zustand wie bekannt mit „X“ gekennzeichnet. Daraus resultiert das verwendete Modell: $Kontext_{Ressourcen}(Ressourcen_{Strom}, Ressourcen_{Last})$.

Szenario	$Ressourcen_{Strom}$	$Ressourcen_{Last}$
Fachl. Szenario 1 „Werft“	Batteriebetrieb (1)	normale Last (0)
Fachl. Szenario 2 „Hotel“	Netzbetrieb (2)	normale Last (0)
Fachl. Szenario 3 „Büro“	Netzbetrieb (2)	normale Last (0)
Allg. Szenario A „Akku“	niedriger Batteriestand (0)	beliebige Last (X)
Allg. Szenario B “Internet“	beliebiger Batteriestand (X)	beliebige Last (X)
Allg. Szenario C „System“	beliebiger Batteriestand (X)	hohe Last (1)

Tabelle 5.9: $Kontext_{Ressourcen}(Ressourcen_{Strom}, Ressourcen_{Last})$

Der Umgebungs- und der Ressourcenkontext definieren die Randbedingungen, unter denen sich die Anforderungen des Benutzers ändern. Deshalb wird nun definiert unter welchen Umständen, bzw. in welchem Szenario, welche Benutzeranforderungen gelten.

5.2.3.3 Benutzerkontext

Da die Motivation dieser Arbeit unter anderem in dem Optimieren der Gebrauchstauglichkeit für den Anwender besteht, ist die Berücksichtigung nicht-funktionaler Qualitätseigenschaften von großer Bedeutung. Die Bewertung des aktuellen Zustands, welche Eigenschaften (wie Geschwindigkeit oder Ressourcenverbrauch) als wie wichtig bewertet werden wird in *Benutzerkontexten* festgehalten. Zunächst muss definiert werden, welche Parameter der *Benutzerkontext* beinhalten soll. Orientiert an den sechs Kategorien der ISO 9126 (Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Wartbarkeit, und Übertragbarkeit, siehe 2.3.1) sowie dem Anspruch an die externen Charakteristiken findet zunächst eine Auswahl der wichtigen Qualitätseigenschaften statt. Aus diesen resultieren letztlich die benötigten QoS-Eigenschaften der Komponenten. Das mobile Endgerät verfügt über sich ändernde Leistungsressourcen. Durch Änderung der äußeren Gegebenheiten (z.B. Wechsel von WLAN auf 3G) kann sich die verfügbare Qualität der Internetverbindung schnell ändern. Diese Ressourcen (siehe 5.2.3.2) haben natürlich in der Regel keinen Einfluss auf die Benutzbarkeit oder die Richtigkeit einer Komponente. So richten sich die Anforderungen des Benutzers im Rahmen dieser Arbeit zunächst nach verfügbaren Leistungsressourcen. Laut dem Qualitätsmodell ISO/IEC 9126-1 entspricht die (nicht-funktionale) Optimierung der Anwendung bezüglich der Leistungsressourcen dem Abschnitt Effizienz. Dieser beinhaltet die Eigenschaften, die das Verhältnis zwischen Leistungsstärke und Zeitverhalten unter gegebenen Umständen beschreiben. Daraus resultierten die drei Eigenschaften: Zeitverhalten ($Benutzer_{Zeit}$), Verbrauchsverhalten ($Benutzer_{Verb}$), und Effizienzkonformität. Effizienzkonformität, beziehungsweise der Grad, in dem die Software Normen oder Vereinbarungen zur Effizienz erfüllt, spielen für dieses Konzept jedoch keine Rolle. Es kann aber sein, dass der Grad der Richtigkeit ($Benutzer_{Richtigk}$) (bzw. Genauigkeit) je nach Anwendungsfall bzw.

Szenario variieren soll. Diese Eigenschaften entstammen dem Bereich *Funktionalität*. Die anderen Teilmerkmale des Bereichs Funktionalität spielen im Rahmen dieser Arbeit keine Rolle: *Interoperabilität* wird nicht benötigt, da die Anwendung nicht mit anderen Anwendungen verbunden wird, *Sicherheit* und *Ordnungsmäßigkeit* mögen zwar in der Realität von Bedeutung sein, sind aber für die Arbeit irrelevant. *Angemessenheit* ist zwar wichtig für die Umsetzung der funktionalen Anforderung des Benutzers, wird aber vorrausgesetzt. Daraus resultiert das Tripel $Kontext_{Benutzer}(Benutzer_{Zeit}, Benutzer_{Verb}, Benutzer_{Richtigk})$.

Für jedes Szenario gilt es nun eine mindestens ordinalskalierte Ausprägung der drei Werte zu finden, um einen Nützlichkeitsquotienten berechnen zu können. Die Werte stehen hier für die Gewichtung der einzelnen Anforderungen zu jedem Szenario. Beispielsweise ist dem Benutzer auf der Werft das Zeitverhalten besonders wichtig, das Verbrauchsverhalten mittelwichtig, und die Richtigkeit nur wenig wichtig.

Für den Entwurfs muss festgelegt werden wann welcher Benutzerkontext aktiv werden soll. Dazu beschreibt man für jeden Benutzerkontext die Kontexte für welchen dieser gilt. Es stellt sich zudem die Frage, wie die Benutzerkontexte erhoben werden - sicher hat nicht jeder Benutzer die gleichen Ansprüche. Denkbar sind sowohl vordefinierte Kontexteinstellungen die durchschnittlichen Benutzererwartungen entsprechen, oder eine Schnittstelle mit deren Hilfe der Benutzer selbstständig seine Anforderungen die bei eintretenden Ereignissen (z.B. sehr geringer Batteriestand) oder innerhalb definierter Kontexte (wenig Internet, Netzbetrieb) gelten eintragen kann. Zunächst wird jedoch definiert, welche Eigenschaften der Umgebung und des inneren Zustands in den jeweiligen Szenarien gelten.

Konzept: Szenarioeigenschaften

- Beim *fachlichen Szenario 1* kann man davon ausgehen, dass auf der Werft die Internetverbindung schlecht ist und sich das Gerät im Batteriebetrieb befindet.
- Für das *fachliche Szenario 2* gelten die Bedingungen, dass das Gerät am Stromkabel angeschlossen ist, und die Internetverbindung mindestens mittelmäßig ist.
- Im *fachlichen Szenario 3* verfügt der Benutzer über unbegrenzten Strom sowie eine gute Internetverbindung
- Das *allgemeine Szenario A* bedingt, dass das Gerät einen niedrigen Akkustand hat und auch nicht am Stromkabel angeschlossen ist. Da dies die anderen Szenarien überlagert, ist die Qualität der Internetverbindung irrelevant.
- Für das allgemeine *Szenario B* gilt, dass keine Internetverbindung vorhanden ist. Die anderen Parameter sind wie auch bei Szenario A zunächst unbedeutend.

- Analog zu A und B, charakterisiert das *allgemeine Szenario C* einzig eine hohe Systemlast.

Nun kann vor der Adaptionsentscheidung festgestellt werden, in welchem Szenario der Benutzer sich befindet und dementsprechend die richtigen Anforderungen respektive der passende Benutzerkontext ausgewählt werden. Es wird hier an dieser Stelle also zur Benutzerkontextauswahl ein regelbasierter Ansatz gewählt. Da diese Auswahl aber zunächst am Verhalten des Systems nichts ändert, bleibt die Adaptionsentscheidung nutzenorientiert. Für dieses Konzept gehen wir zunächst von vordefinierten Benutzerkontexten aus, die aus den beschriebenen Szenarien hervorgehen.

Szenario	$Benutzer_{Zeit}$	$Benutzer_{Verb}$	$Benutzer_{Richtigk}$
Fachl. Sz. 1 „Werft“	hoch (3)	mittel (2)	wenig (1)
Fachl. Sz. 2 „Hotel“	mittel (2)	mittel (2)	mittel (2)
Fachl. Sz. 3 „Büro“	wenig (1)	wenig (1)	hoch (3)
Allg. Sz. A „Akku“	hoch (3)	hoch (3)	nicht (0)
Allg. Sz. B “Internet“	nicht (0)	nicht (0)	nicht (0)
Allg. Sz. C „System“	nicht (0)	hoch (3)	nicht (0)

Tabelle 5.10: $Kontext_{Benutzer}(Benutzer_{Zeit}, Benutzer_{Verb}, Benutzer_{Richtigk})$

In der Praxis lässt sich jedoch nicht damit rechnen, dass die Kombination aus aktuellem System- und Umgebungszustand genau auf ein Szenario passt. Im Hotel kann nämlich durchaus auch mal ein guter, oder im Büro ein mittelmäßiger Internetempfang herrschen. Hierbei kann das passendste Szenario gefunden werden, indem man die Abweichungen aller Szenarienparameter von den aktuellen Zuständen berechnet und das Szenario mit der geringsten Abweichung auswählt.

5.2.3.4 Qualitätseigenschaften der Komponenten

Die Qualitätseigenschaften sind der letzte Teil des Quadrupels, auf dessen Basis die Adaptionsentscheidung getroffen wird. Wie im Rahmen des vorigen Abschnitts beschrieben, sind die Qualitätseigenschaften der Komponenten neben den drei Kontextarten der vierte Faktor um einen Nützlichkeitsindex zu einem jeweiligen Zeitpunkt zu bestimmten. Im Rahmen dieser Arbeit werden drei beispielartige Komponenten spezifiziert, die die Anforderungen des Benutzers abdecken sollen und an denen die Rekonfiguration demonstriert werden kann. Die Eigenschaften der Komponenten leiten sich dabei aus den Qualitätsanforderungen des Benutzers ab ($Komponente_{Richtigk}$, $Komponente_{Zeit}$, $Komponente_{Verb}$) (siehe 5.2.3.3), wobei das Verbrauchsverhalten unterteilt wird in die

Umgebungs- und Ressourceneigenschaften Bandbreite ($Komponente_{Inet}$), Akkuverbrauch ($Komponente_{Strom}$), und Systemlast ($Komponente_{Last}$). Die Eigenschaften werden mit „gut“, „mittel“, oder „schlecht“ beschrieben. Dabei ist zu beachten, dass „gut“ bei $Komponente_{Zeit}$ eine *niedrige* Zeit meint, ein „gut“ bei $Komponente_{Richtigk}$ jedoch eine *hohe* Richtigkeit. $Qual_{Komponenten}(Komponente_{Zeit}, Komponente_{Strom}, Komponente_{Inet}, Komponente_{Last})$

Komponentenname	$Komponente_{Zeit}$	$Komponente_{Strom}$	$Komponente_{Inet}$	$Komponente_{Last}$	$Komponente_{Richtigk}$
A „Volle Leistung“	gut	mittel	mittel	schlecht	schlecht
B „Hohe. Qual.“	schlecht	mittel	schlecht	schlecht	gut
C „Durchschnitt“	mittel	mittel	mittel	mittel	mittel

Tabelle 5.11: $Qual_{Komponenten}(...)$

Nachdem nun die Modelle, die für eine nutzenorientierte Adaptionsentscheidung benötigt werden, erstellt wurde, müssen die ausgewählten adaptiven Mechanismen zu einem Programmfluss verknüpft werden. Dies geschieht über den im Grundlagenkapitel (2.1.3) erläuterten Regelkreis.

5.2.4 Kontrollfluss

Der Kontrollfluss eines adaptiven Systems bildet in der Regel wie erläutert eine Art Regelkreis - einen *closed control loop*. Dieser beinhaltet zunächst vier Phasen: *collect*, *analyze*, *decide* und *act*. (vgl. Kapitel 2.1.3, siehe Abbildung 5.2) Diese vier Phasen werden in diesem Konzept verwendet um die ausgewählten Mechanismen miteinander zu verbinden. Die Adaptionsdauer und -Häufigkeit wird dabei von der *Act*-Phase protokolliert und dem Entscheidungsmechanismus zur Verfügung gestellt. In den folgenden Abschnitten wird jeder Prozess und die jeweuligen Unterprozesse aus dem für dieses Konzept erstellten Regelkreis dargestellt und erläutert.

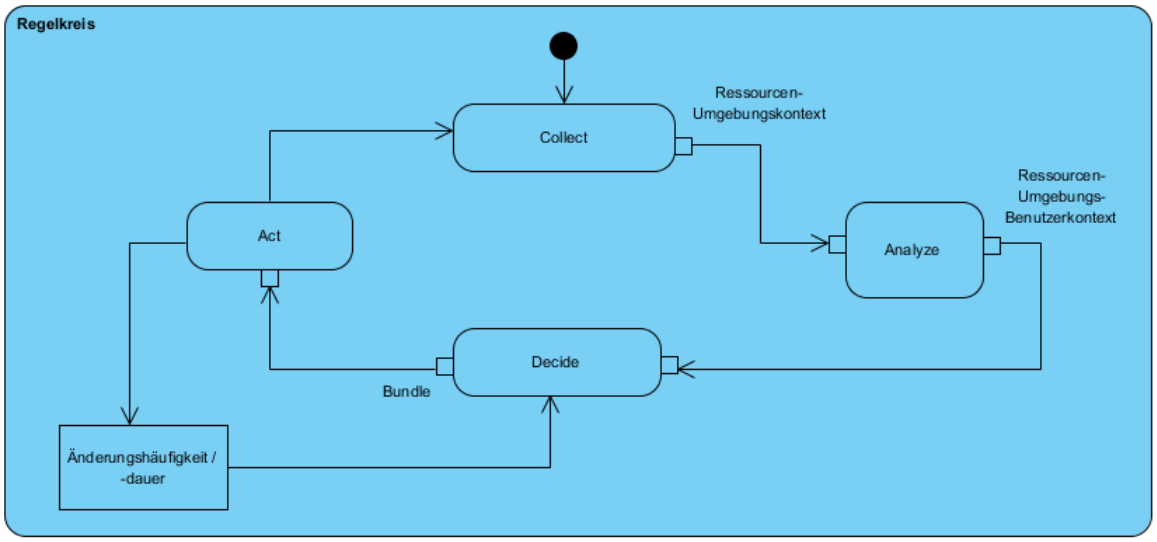

Abbildung 5.2: Konzept - Regelkreis

5.2.4.1 *collect*

Die *collect*-Phase beinhaltet in diesem Konzept das Sammeln aller Parameter des Benutzer- und des Ressourcenkontexts. Daten des Benutzerkontext, sprich das Eingreifen des Benutzers, werden abseits des Funktionsaufrufs nicht erwartet, da diese Anforderungen a priori in der Anwendung festgelegt sind. Die Daten werden auf die vorgegebenen Anforderungsmodelle aus Abschnitt 5.2.3 abgebildet, um diese in eine numerische und intervallskalierte Form zu bringen. (siehe Abbildung 5.3)

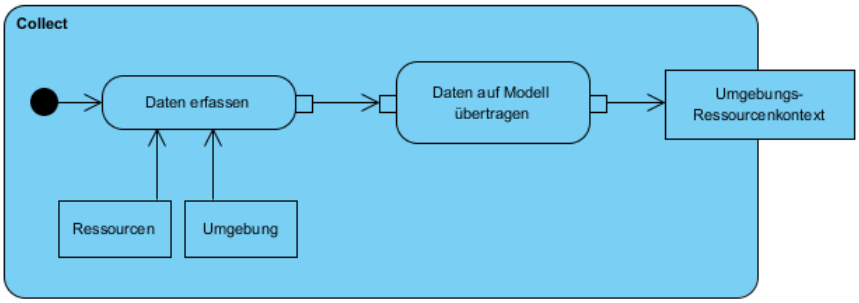

Abbildung 5.3: Konzept - *collect*

5.2.4.2 *analyze*

In der *analyze*-Phase wird überprüft ob anhand der gesammelten Daten ein Kontextwechsel vorliegt, und eine Adaptionsentscheidung zu treffen ist (*reaktive* Adaption). Aufgrund des asynchronen Adaptionszeitpunktes wird die *decide*-Phase daher ausschließlich als Resultat der vorherigen *analyze*-Phase ausgeführt. Andernfalls werden weiter Daten gesammelt, und diese analysiert, bis sich die gesammelten Daten signifikant geändert haben. Hier muss deshalb der ursprüngliche Regelkreis die Möglichkeit der Rückkehr in die *collect*-Phase als Resultat jeder anderen Phase erweitert werden. (siehe Abbildung 5.4). Falls ein Kontextwechsel vorliegt, muss der passendste Benutzerkontext, der die zu erfüllenden Anforderungen und deren Gewichtung enthält, ausgewählt werden. Für das Konzept sollen die Abweichungen der Kontextparameter aufsummiert werden und der Kontext mit der geringsten Abweichung gewählt werden.

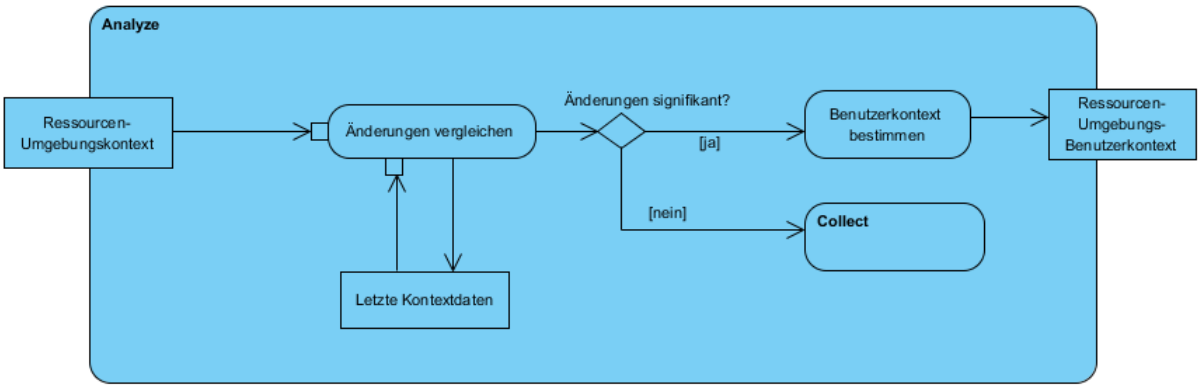

Abbildung 5.4: Konzept - *analyze*

5.2.4.3 *decide*

In der *decide*-Phase wird nun die Entscheidung getroffen, ob eine Rekonfiguration durchzuführen ist. Dabei werden zunächst die vorhandenen Komponenten, welche die funktionalen Anforderungen Benutzers erfüllen, ermittelt. Mit Hilfe der funktionalen und nicht-funktionalen Eigenschaften werden, zusammen mit den gesammelten Daten der Kontexte, entsprechende Kennzahlen berechnet, die der Nützlichkeit der Komponente in dem aktuellen Kontext entsprechen (*nutzenorientierte Adaptionsentscheidung*). Da durch die kompositionelle Adaption primär eine starke Adaption vorliegt, wird zur endgültigen Adaptionsentscheidung noch eine Auswirkungsvorhersage benötigt. Die dazu benötigten Daten werden in der *act*-Phase, die z.B. die Dauer der Rekonfiguration messen kann, ermittelt. Eine weitere Lösung wäre eine Schätzung auf Basis der aktuellen Systemkonfiguration. (siehe Abbildung 5.5)

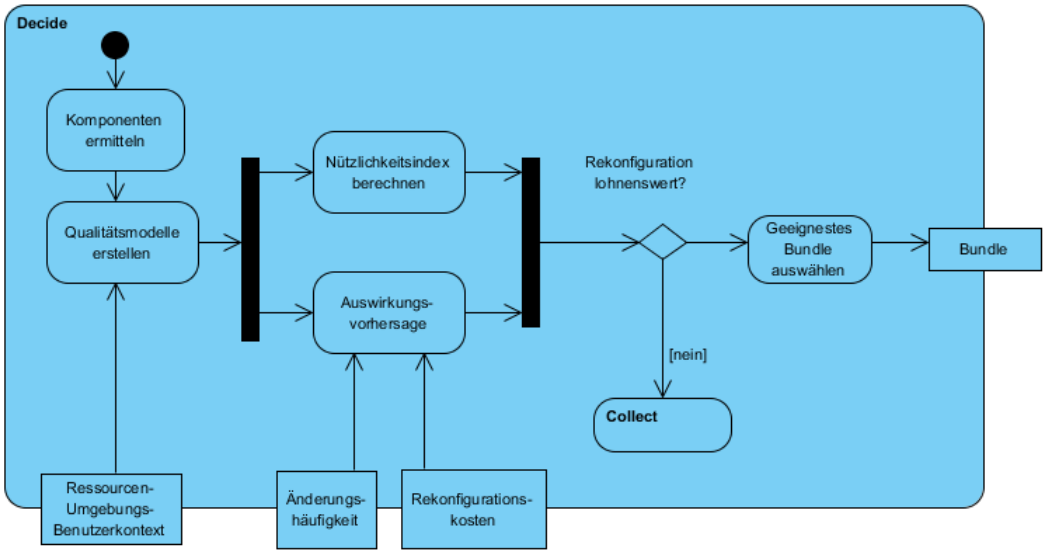

Abbildung 5.5: Konzept - *decide*

Die Nützlichkeitsfunktion ist neben den adaptiven Mechanismen die wichtigste Komponente der Anwendung. Auf Basis der im Verlauf dieses Kapitels definierten Parameter werden Funktionen aufgestellt, die die einzelnen Werte zu einem Nützlichkeitsindex aggregieren.

Der Umgebungsindex ($Index_{umgebung}$) gibt an wie gut eine Komponente für die aktuelle Umgebung geeignet ist. Die Anforderung des Benutzers bezüglich des Zeitverhaltens gewichten das Teilergebnis.

$$Umgebung_{inet_{norm}} = \frac{Umgebung_{inet}}{Umgebung_{inet_{max}}}$$

$$Komponente_{inet_{norm}} = \frac{Komponente_{inet}}{Komponente_{inet_{max}}}$$

$$Index_{umgebung} = User_{Zeit} \times \frac{Umgebung_{inet_{norm}}}{Komponente_{inet_{norm}}}$$

Der Ressourcenindex ($Index_{ressourcen}$) gibt an wie gut eine Komponente für den aktuellen Ressourcenzustand geeignet ist. Die Anforderungen des Benutzers aus dem Benutzerkontext gewichten dabei ebenso die Teilergebnisse.

$$Ressourcen_{strom_{norm}} = \frac{Ressourcen_{strom}}{Ressourcen_{strom_{max}}}$$

$$Komponente_{strom_{norm}} = \frac{Komponente_{strom}}{Komponente_{strom_{max}}}$$

$$Ressourcen_{last_{norm}} = \frac{Ressourcen_{last}}{Ressourcen_{last_{max}}}$$

$$Komponente_{last_{norm}} = \frac{Komponente_{last}}{Komponente_{last_{max}}}$$

$$Index_{Ress} = User_{Verb} \times \frac{Ressourcen_{strom_{norm}}}{Komponente_{strom_{norm}}} + User_{Zeit} \times \frac{Ressourcen_{last_{norm}}}{Komponente_{last_{norm}}}$$

Der Richtigkeitsindex ($Index_{Richtigk}$) gibt an wie die Richtigkeitseigenschaft der Komponente den Bedürfnissen des Benutzers entspricht.

$$Komponente_{richtigk_{norm}} = \frac{Komponente_{richtigk}}{Komponente_{richtigk_{max}}}$$

$$Index_{Richtigk} = User_{Richtigk} \times Komponente_{richtigk_{norm}}$$

Der Gesamtindex spiegelt nun die Gebrauchstauglichkeit der Komponente in diesem Szenario unter den Anforderungen des Benutzers wieder. Anhand dieses Indexes wird nun entschieden, ob der Wert einer anderen Komponente signifikant höher ist als der gerade eingesetzten.

$$Index_{Gesamt} = Index_{Richtigk} + Index_{Ress} + Index_{Umgebung}$$

Für eine vollständige Nützlichkeitsfunktion fehlt nun noch die Vorhersage etwaiger Auswirkungen. In diesem Fall werden dazu die bisherigen durchschnittlichen Zeitkosten einer Rekonfiguration und die Änderungshäufigkeit ausgewertet. Dabei bieten sich zwei zu betrachtende Werte an: (1) Die letzte Rekonfigurationsdauer im Vergleich zum Abstand der Rekonfigurationen. Hierbei wird vereinfach angenommen, dass sich die Rekonfiguration lohnt falls der Abstand der letzten Rekonfiguration die Rekonfigurationsdauer um Faktor Zehn übersteigt. Dieser Parameter lässt sich jedoch empirisch oder berechnend verbessern. (2) Die Tendenz der Abstände der Rekonfigurationen. Dabei betrachtet man Entwicklung der

durchschnittlichen Abstände der Rekonfigurationszeitpunkte um somit den Trend abschätzen zu können. Im Rahmen dieser Arbeit wurde dabei die *Regressionsanalyse* eingesetzt. Auf Basis von den Wertepaaren (x, y) wird die Steigung m zu den einzelnen x-Werten berechnet werden. Dabei sei y der Abstand zwischen dem Ereignis und dem jetzigen Zeitpunkt und x die abgezählte Nummer des Ereignisses. Die Steigung m_x beschreibt dabei wie stark sich der Abstand zwischen zwei Rekonfigurationen an Position x verändert (vgl. Opfer (2002)).

$$m_x = \frac{\Delta y}{\Delta x} = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

Nun wird das arithmetische Mittel aller bisherigen Steigungen berechnet. j bezeichnet dabei die Anzahl der Rekonfigurationen.

$$\overline{m_j} = \frac{1}{x} \sum_{j=1}^{x} m_j$$

Das Ergebnis sind Wertepaare $\left(j, \overline{m_j}\right)$ die den durchschnittlichen Abstand der Rekonfigurationszeitpunkten der letzten j-Rekonfigurationen wiederspiegeln. Aus denen kann nun mittels der *Regressionsanalyse* eine Funktion der Form $y = A + B * x$ modelliert werden, deren Steigung die Tendenz der Abstände wiederspiegelt. Berechnet werden dabei zunächst die Größen $\overline{A}$ und $\overline{B}$, welche man als *Regressionskoeffizienten* bezeichnet, als Schätzung für die „wahren“ Parameter A und B. Dabei gilt $i \equiv x$, sprich wenn z.B. $i = 2$ dann $x_2 = 2$. (vgl. Opfer (2002))

$$\overline{B} = \frac{(\frac{1}{N} * \sum_{k=1}^{N} x_k * m_k) - (\frac{1}{N} * \sum_{k=1}^{N} x_k) * (\frac{1}{N} * \sum_{k=1}^{N} m_k)}{(\frac{1}{N} * \sum_{k=1}^{N} x_k^2) - (\frac{1}{N} * \sum_{k=1}^{N} x_k)^2)}$$

und

$$\overline{A} = (\frac{1}{N} * \sum_{k=1}^{N} m_k) - \overline{B} * (\frac{1}{N} * \sum_{k=1}^{N} x_k)$$

In Abhängigkeit von der berechneten Tendenz bzw. Steigung der Regressionsgraden($\overline{B}$) und den Rekonfigurationskosten kann nun entschieden werden, ob sich eine Rekonfiguration lohnt. Vereinfacht wird im Rahmen dieser Arbeit angenommen, dass bei sinkender Tendenz es sich nicht mehr lohnt und bei steigender Tendenz es sich lohnt. Nachdem die Entscheidung getroffen wurde, ob eine Rekonfiguration durchgeführt wird, folgen nun entweder die *collect*-Phase oder die eigentlich Rekonfiguration, respektive *act*-Phase.

5.2.4.4 *act*

In der *act*-Phase wird schließlich die Adaption durchgeführt und zudem Daten über selbige für den Zugriff der *decide*-Phase hinterlegt. Dazu können wie genannt Adaptionshäufigkeit, -kosten, oder -dauer zählen. (siehe Abbildung 5.6)

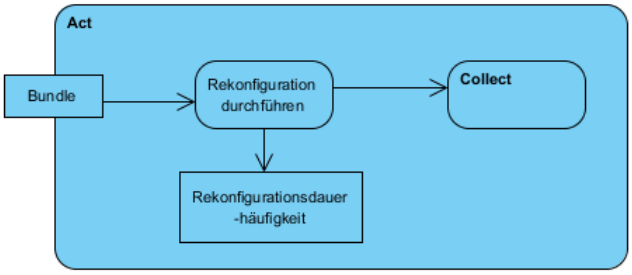

Abbildung 5.6: Konzept - *act*

5.3 Zusammenfassung Konzept

In diesem Kapitel wurden nach dem informellen Vorstellen eines Szenarios dessen Anwendungsfälle erstellt. Daraus konnten abstrahierte quantifizierbare Qualitätsmodelle erstellt werden, die die entscheidenden Einflüsse auf eine Rekonfigurationsentscheidung darstellen: Die Anforderungen des Benutzers, die Güte der im Rahmen des Szenarios vorhandenen Ressourcen, den Umgebungszustand und die Qualitätseigenschaften der Komponenten. Mit Hilfe der *Self-**-Eigenschaften und der ISO 9126 konnten fünf Qualitätsmerkmale identifiziert werden, die mit Hilfe der Anwendung adressiert werden müssen. Danach wurden auf Basis der gewonnen Informationen die Eigenschaften des adaptiven Systems identifiziert, die das Erreichen der Ziele der Anwendung - das Optimieren der Gebrauchstauglichkeit - unterstützen. Dabei wurde festgestellt, dass ein System mit kompositioneller, nicht-antizipierter und starker Adaption benötigt wird, welches eine nutzenorientierte, asynchrone, und reaktive Adaptionsentscheidung verwenden soll. Anschließend wurden diese Eigenschaften auf den geschlossenen Regelkreis abgebildet, um so die Prozesse und Komponenten der Anwendung entwickeln zu können. Im folgenden Kapitel wird nun dieses Konzept mit Hilfe eines Komponentenframeworks auf einer mobilen Plattform umgesetzt, um die verwendeten Konzepte und Ideen zu prüfen. Die Idee ein Komponentenframework zu verwenden wurde dabei von der Entscheidung eine kompositionelle Adaption zu verwenden bekräftigt. Um die Übersicht zu erhöhen sind die Ergebnisse des Konzepts nochmal tabellarisch zusammengefasst.

Resultierende adaptive Mechanismen

Folgende Mechanismen haben sich im Laufe dieses Kapitels ergeben und sollen innerhalb des Regelkreises eingesetzt werden. In der Tabelle sind neben dem Namen noch eine zusammengefasste Begründung und die daraus resultierende (technologische) Konsequenz aufgeführt.

Mechanismus	Begründung	Konsequenz
Kompositionelle Adaption	Hohe benötigte Mächtigkeit	Technologie zum Austausch von Komponenten, z.B. ein Komponentenframework benötigt
Nicht-antizipierte Adaption	Nicht alle Varianten und Kombinationen der Kontexte sind vorher bekannt oder können berücksichtigt werden	Flexible Bestimmung des passendsten Benutzerkontexts
Starke Adaption	Aufwändige architekturbasierte Adaption	Berücksichtigen einer Auswirkungsvorhersage wegen hohem Rekonfigurationsaufwand
Nutzenorientierte Adaptionsentscheidung	Anforderung ist die Optimierung des Nutzens	Entwurf einer Nützlichkeitsfunktion
Asynchrone Adaptions-Entscheidung	Das System muss passend konfiguriert sein bevor die Funktionalität angefordert wird	Kontextänderungen müssen Adaptionsauslöser sein
Reaktive Adaptionsentscheidung	Nächster Adaptionszeitpunkt nicht vorhersehbar	Reaktion auf Kontextwechsel

Tabelle 5.12: Resultierende adaptive Mechanismen

Kontextmodelle

In diesem Konzept sind mehrere Modelle definiert worden, die die verschiedenen Kontexte und die Qualitätseigenschaften der Komponenten beschreiben. Die Modelle werden alle zur Adaptionsentscheidung herangezogen. Wobei in der *Analyze*-Phase aus Umgebungs- und Ressourcenkontext noch der passende Benutzerkontext ausgewählt wird. Das Szenario wird in diesem Konzept als Quadrupel abgebildet:

$$Szenario(Kontext_{Umgebung}, Kontext_{Ressourcen}, Kontext_{Benutzer}, Qual_{Komponenten})$$

Die Kontexte setzen sich dabei wie folgt zusammen:

Name	Formel	Beschreibung
Umgebungskontext	$Kontext_{Umgebung}(Umgebung_{Inet})$	Der Umgebungskontext bestimmt sich über die Qualität der vorhandenen Internetanbindung.
Ressourcenkontext	$Kontext_{Ressourcen}(Ressourcen_{Last}, Ressourcen_{Strom})$	Der Ressourcenkontext ist eine Kombination aus aktueller Systemlast und Stromversorgung.
Benutzerkontext	$Kontext_{Benutzer}(Benutzer_{Zeit}, Benutzer_{Verb}, Benutzer_{Richtigk})$	Der Benutzerkontext setzt sich aus dem Zeitverhalten, dem Verbrauchsverhalten, und der Richtigkeit zusammen.
Qualitätseigenschaften der Komponenten	$Qual_{Komponenten}($ $Komponente_{Zeit}, Komponente_{Strom}$ $, Komponente_{Inet}, Komponente_{Last},$ $Komponente_{Richtigk})$	Die Qualität einer Komponente bestimmt sich durch das Zeitverhalten, den Stromverbrauch, den Bandbreitenbedarf und die verursachte Last.

Tabelle 5.13: Szenariomodell und Kontexte

Im nächsten Kapitel wird auf Basis des Konzepts eine Architektur für einen Entwurf entwickelt, der dann auf einer konkreten mobilen Plattform umgesetzt wird. Es wird anschließend in einer Simulationsumgebung verifiziert, ob der Entwurf den Anforderungen gerecht werden kann, und für die unterschiedlichen Szenarien die passenden Komponenten auswählt.

Kapitel 6

Entwurf

Der Entwurf einer mobilen Anwendung soll die Realisierbarkeit des entwickelten Konzepts demonstrieren. Zudem liefert der Entwurf erste Aufschlüsse über den Nutzen. Auf Basis des Konzepts wurde deshalb *ROSt* (Ressourcen-Orientiertes adaptives System) entworfen, welches in diesem Kapitel vorgestellt wird. Dieses ist dabei wie folgt strukturiert: zunächst werden Technologie und technologische Architektur, sowie deren Eigenheiten bezüglich der Implementierung, vorgestellt. Anschließend werden erwähnenswerte Punkte aufgeführt, die sich im Laufe der Implementierung als wichtig hervortaten.

6.1 Architektur

Die Architektur des Systems orientiert sich stark an den im Konzept vorgestellten Modellen. Diese müssen zunächst in Form eines abstrakten Systementwurfs konkretisiert werden. Der Regelkreis wird dabei als fachlich geschlossene Komponente (*RostBundle*) umgesetzt. Dies hat den Vorteil, dass er somit problemlos in andere Umgebungen integriert werden kann und somit wiederverwendbar ist. Die Anwendung besteht aus fünf Komponenten:

- `QualityBundle`: Ein Bundle mit hohem Ressourcenverbrauch aber hoher funktionaler Güte. (vgl. Abschnitt 5.2.3.4)
- `PerformanceBundle`: Ein Bundle mit mittlerem Ressourcenverbrauch und niedriger Richtigkeit, aber sehr gutem Zeitverhalten.
- `MediumBundle`: Ein Bundle mit durchwegs mittelmäßigen Eigenschaften. (vgl. Abschnitt 5.2.3.4)
- `ROStBundle`: Das Bundle, welches die adaptiven Mechanismen und den Regelkreis beinhaltet, die Rekonfigurationsentscheidung trifft, und die Rekonfiguration durchführt. (vgl. Abschnitt 5.2.4)

- Mobile Anwendung: Die Anwendung für das mobile Betriebssystem, welche als Hostanwendung für das Komponentenframework und als Basis für die grafische Oberfläche der Schiffsprüferanwendung dient.

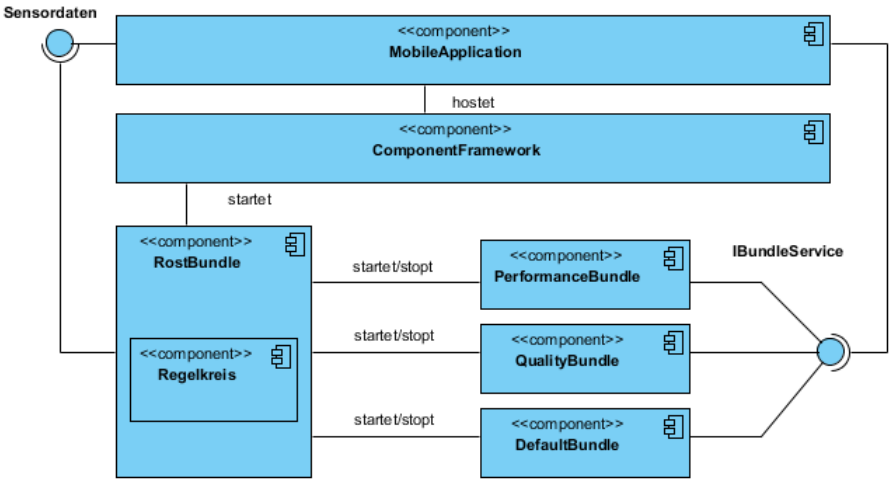

Abbildung 6.1: Komponenten der Anwendung

Das RostBundle beinhaltet den Regelkreis und seine typischen vier Phasen. Die vier Phasen werden durch vier eigene Komponenten abgebildet, die die Ergebnisse der jeweiligen Phase an die nachfolgende weitergeben. (vgl. Konzept 5.2.4).

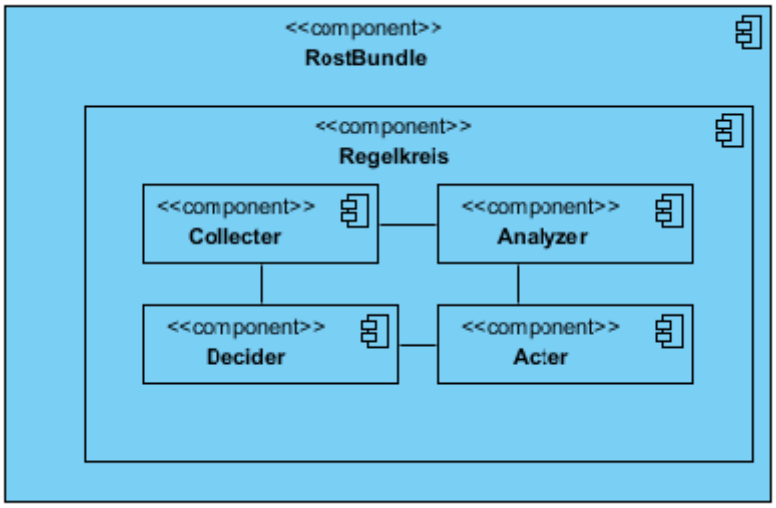

Abbildung 6.2: `RostBundle` Komponenten

Technische Eigenschaften des Entwurfs werden in den folgenden Abschnitten vorgestellt.

6.2 Realisierung

ROSt bzw. das ROSt-Bundle fungiert als eine Komponente, die autonom neben der Hostanwendung und den Komponenten, die die anzupassende Funktionalität bereit stellen, das reflexive Verhalten und die Rekonfiguration über das OSGi-Framework realisiert. Die Technologie der Realisierung fußt dabei auf den aus Abschnitt 3.1 hervorgegangenen Technologien Android als mobiles Betriebssystem und Apache Felix als konkrete OSGi-Implementierung. Die Abbildung des Konzepts erfordert natürlich Anpassungen an die technologischen Möglichkeiten. Insbesondere aufgrund der in Java limitiert unterstützten Polymorphie in Kombination mit dem dynamischen Classloading des Komponentenframeworks, aber auch aufgrund anderer Eigenheiten der verwendeten Technologie, bedurfte es einiger technischer Umwege, auf die im Verlauf des Kapitels teilweise eingegangen wird. Für die fachliche Aussagekräftigkeit spielt dies jedoch keine Rolle. Folgende zwei Abschnitte erläutern, wie die verwendeten Technologien für die Realisierung der adaptiven Mechanismen eingesetzt wurden.

6.2.1 Mobile Plattform

Die mobile Plattform stellt systemeigene Mechanismen zur Verfügung(siehe Grundlagen „Android“, Kapitel 3.3), um über die Sensorik die benötigten Daten für den Ressourcen-

und Umgebungskontext zu erfassen. In diesem Beispiel sind das die verfügbare Bandbreite, der Batteriestand, sowie die aktuelle Systemlast. Im Falle von Android kann dabei teilweise auf proprietäre Services zurück gegriffen werden, die beispielsweise den Batteriestand per *Broadcast* verbreiten. Zum Überwachen der prozentualen Systemauslastung muss man jedoch die Datei `/proc/stat` aus dem Proc-Dateisystem auswerten. Sie enthält in den *cpu*-Zeilen Werte für *User*, *System*, *Nice* und *Idle*-Zeiten seit Systemstart. Bei einem überlasteten System ist ein solches Auslesen natürlich nicht sonderlich zuverlässig. Die Art der Internetverbindung kann wiederum über einen Systemservice erfragt werden. Dabei können beispielsweise über das *android.net.NetworkInfo*-Objekt der Typ der Verbindung (WiFi,Mobile,..) und die theoretische Bandbreite ausgelesen werden. Zudem kann überprüft werden, ob aktuell eine Internetverbindung besteht. Die Messung der effektiv zur Verfügung stehenden Bandbreite, in der Praxis natürlich relevanter, benötigt jedoch weitere bekannte Mechanismen, die nicht Teil dieses Entwurfs sein sollen.

Die Realisierung dieser unterschiedlichen Wege zum Erfassen der benötigten Daten geschieht diesem Entwurf über Adapter, die die oben aufgeführten Aufgaben kapseln. Der *Collector*, sprich das Modul dass für das Datensammeln und Überführen dieser Daten in eine ordinal- oder intervallskalierte Form für die fachlichen Modelle zuständig ist, kann nun in der *collect*-Phase auf die Adapter zugreifen und den aktuellen Stand auslesen.

6.2.2 Komponentenframework

Das Komponentenframework übernimmt die dynamische Rekonfiguration der einzelnen Komponenten respektive Bundles. In diesem Entwurf sind dabei die Komponenten a priori bekannt - eine komplett dynamische Abfrage über ein OSGi-Service-Repository ist jedoch auch realisierbar, sodass auch Fremdkomponenten für die Rekonfiguration verwendet werden könnten. Die Bundles repräsentieren dabei unterschiedliche Implementierungen der selben funktionalen Schnittstelle. Für das vorgegebene Szenario besteht die Schnittstelle beispielhaft aus einer Methode für das Komprimieren und Versenden eines Prüfberichts inklusive Bild-Dateien. Über Annotationen an den jeweiligen Klassen werden die nichtfunktionalen Eigenschaften sowie die Funktionsgüte der Komponente spezifiziert. Dies dient dazu, dass die Bundles miteinander verglichen werden können. In Abbildung 6.3 ist ein Service mit Schnittstelle und Annotation aufgeführt. Es ist festzustellen, dass für die Verwendung von OSGi-Bundles lediglich die Klasse mit ihren Qualitätsattributen annotiert werden muss - der Rest ist in jedem OSGi-Bundle vorhanden. Somit könnte so eine Qualitätsannotation auch ohne Kenntnis von Implementierungsdetails durchgeführt wird, wobei natürlich die Frage woher die Quanfizierung dieser Eigenschaften stammen, separat zu beantworten ist. Hierbei sind aber auch aus dem Regelkreis zur Laufzeit ausgeführte Profiling-Methoden

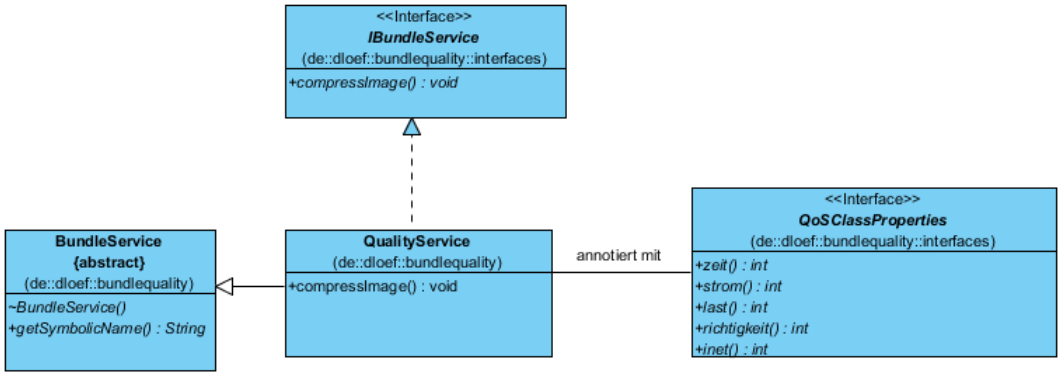

Abbildung 6.3: `QualityBundle` Klassendiagramm

denkbar, die dynamisch die Eigenschaften des Bundles ermitteln können. Somit könnte bei dem Einsatz einer Steuersoftware wie ROSt, jedes OSGi-Bundle, welches für die Ausführung der erwarteten Funktion geeignet ist, verwendet werden.

Der Regelkreis zur Realisierung des adaptiven Verhaltens sind auch jeweils in OSGi-Bundles gekapselt. Dies bringt den Nebeneffekt mit, dass auch diese Module nach einem Versionswechsel problemlos ausgetauscht werden könnten. Jede Phase des Regelkreises ist dabei in einem eigenen *Singelton*-Objekt gekapselt.

6.2.3 Qualitätsattribute und Modelle

Die Kontextmodelle (Benutzer-, Umgebungs-, und Ressourcenkontext) sind mit einfachen *POJOs*[17]realisiert, deren Attribute den funktionalen bzw. nicht-funktionalen Eigenschaften und den Anforderungen entsprechen. Die Qualitätsattribute der Komponenten (siehe Kapitel 5.2.3.3) werden bei ROSt mit Hilfe von *Annotationen*[18] an den Klassen, die die Implementierung der `IBundleService`-Schnittstelle darstellen, hinterlegt. Anschließend können sie dann mittels *Reflections*[19] zur Laufzeit ausgelesen werden. Folgend ein Codeausschnitt der Annotation und einer annotierten Klasse.

[17]POJO steht für *Plain Old Java Object*.

[18]In Java ist eine Annotation ein Sprachelement, welches die Einbindung von Metadaten in den Quelltext erlaubt.

[19]Das Reflection-Modell erlaubt es, Klassen und Objekte, die zur Laufzeit von der JVM im Speicher gehalten werden, zu untersuchen und in begrenztem Umfang zu modifizieren. Ullenboom (2009)

Ausschnitt 1: Annotation Qualitätseigenschaften

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public  @interface QoSClassProperties {

    int zeit()  default  0;
    int strom()  default 0;
    int last()  default 0;
    int richtigkeit() default 0;
    int inet()  default 0;
}
```

Im Vergleich zum Konzept (siehe Abschnitt 5.2.3.4) wurden die textuellen Werte (gut, mittel, schlecht) durch eine numerische Repräsentation von 1-10 (10=schlecht, 1=gut) ersetzt.

Ausschnitt 2: Classheader mit Annotation

```
@QoSClassProperties(last=8, richtigkeit=10 ,strom=7 ,zeit=1 ,inet=2)
public class PerformanceService
extends BundleService implements IBundleService
{
(...)
}
```

6.2.4 Ablauf

Im folgenden Kapitel wird der Ablauf des Programms von der ersten Inbetriebnahme bis zum laufenden Betrieb erläutert.

6.2.4.1 Inbetriebnahme

Da die virtuelle Maschine Androids (DalvikVM) nativ keinen Java-Bytecode ausführen kann, müssen in den OSGi-Bundles und im JAR-File des OSGi-Frameworks zusätzlich zum Java-Bytecode der Dalvik-Bytecode vorhanden sein. Dazu übersetzt man den vorhandenen Java-Bytecode mit einem Crosscompiler und fügt ihn der JAR-Datei hinzu. Anschließend müssen die Bundles auf das Androidgerät übertragen werden. Die genauen Befehle können im Anhang, im Abschnitt A, eingesehen werden.

Jede Androidanwendung startet mit ihrer *MainActivity*. Die mobile Anwendung, welche

ihr Verhalten nun anpassen soll, stellt aus diesem Grund die Hostanwendung für das eingebettete OSGi-Framework dar. In ihrer *MainActivity* wird die Apache-Felix-Instanz gestartet, welche die OSGi-Bundles verwaltet. Das ROSt-Bundle, sprich die Verwaltung der übrigen Bundles, wird dabei direkt installiert und aktiviert. Über den *Classpath* oder ein *Bundlerepository* können nun weitere Bundles installiert werden. Diese fügen dann ihre zu exportierende Serviceschnittstelle dem Bundlekontext hinzu. So kann man über die Felix-Instanz die zur Verfügung stehenden Services erfragen. Der Regelkreis ist nun bereit auf Kontextänderungen zu reagieren und startet seinen Zyklus.

6.2.4.2 Zyklus

In diesem Abschnitt wird der grobe Durchlauf eines Zyklus innerhalb des Regelkreises von ROSt beschrieben (siehe auch Abbildung 6.4). Durch das standardmäßige Starten des Bundles *RostBundle* durch das OSGi-Framework wird mit Hilfe des *BundleActivators* eine Instanz der Klasse Regelkreis erzeugt, welche für die Erzeugung der weiteren Komponenten des Regelkreises verantwortlich ist. Über den *Activator* und den Regelkreis sind die Instanzen der einzelnen Komponenten als *Singletons* bundleweit verfügbar. Zudem enthält die Regelkreis-Klasse die hinterlegten Benutzerkontexte. Jeder Benutzerkontext enthält die vom Benutzer gestellten Anforderungen (Zeitverhalten, Verbrauchsverhalten, Richtigkeit) und unter welchen Umständen (Internetqualität, Batteriestand, aktuelle Systemlast) welcher Benutzerkontext aktiv werden soll.

Der Regelkreis startet mit dem Eintritt in die *collect*-Phase. Der Collector erzeugt nun einen BroadcastReceiver für den Batteriezustand, der diesen innerhalb des *Intents* über den BatteryManager auslesen kann. Zudem wird durch das parsen der Datei */proc/stat* die Systemlast ausgelesen. Anschließend wird über den ConnectivityManager, auf den über den Systemservice zugegriffen werden kann die Art der aktuellen Netzwerkverbindung (WiFi, Mobile 3G, kein Netzwerk,...) abgefragt werden. Anhand dieser Daten können und die aktuelle Umgebungs- und Ressourcenkontexte erstellt werden, welche diese Daten in ordinal- und intervallskalierter Form enthalten.

In der darauf folgenden *analyze*-Phase wird nun überprüft, ob es Änderungen zwischen den aktuellen und den letzten Kontexten gab. Sofern es keine Änderungen gab, kann auf eine weitere Verarbeitung verzichtet werden, und es findet wieder ein Übergang in die *collect*-Phase statt. Ansonsten wird auf Basis der Kontexte ausgewählt, welches der Szenarien eingetroffen ist, und welcher passendste Benutzerkontext ausgewählt. Da in der Praxis nicht zu erwarten ist, dass für jede mögliche Kombination der Kontextparameter ein hinterlegtes Szenario inklusive Benutzerkontext existiert, muss das am ehesten passende ermittelt werden. In ROSt wurde das mit der Summe der absoluten Abweichungen der normalisierten Kontextwerte für jede Szenario-Benutzerkontext-Kombination ermittelt. Falls der jeweilige

Kontextwert (z.B. Internet) für ein Benuterkontext als irrelevant gekennzeichnet wurde, wird er natürlich nicht berücksichtigt. Nach der erfolgreichen Bestimmung des passendsten Szenarios, können alle Kontexte an die *Decider*-Komponente weitergereicht werden.

Innerhalb des *Deciders* wird zunächst nach geeigneten, im Framework installierten Bundles gesucht, die der funktionalen Schnittstelle *IBundleService* entsprechen. Deren Qualitätseigenschaften können nun entnommen und auf ein internes Modell (*KomponentenQoS*) übertragen werden. Auf Basis einer Liste aller gültigen Bundles kann nur für jedes Bundle der Qualitätsindex nach den in Kapitel 5.2.4.3 beschriebenen Formeln berechnet werden. Nachdem das günstigste Bundle bestimmt werden konnte, und dieses sich von dem aktuell genutzten Bundle unterscheidet, wird anhand der Indexdifferenz zum aktuell installierten Bundles, der Steigung der Regressionsgerade, und der letzten Rekonfigurationsdauer abgeschätzt ob eine Rekonfiguration lohnenswert ist. Falls ja, kann nun die *Acter*-Komponente das entsprechende Bundle starten. Dabei werden sowohl Zeitpunkt als auch Dauer der Rekonfiguration festgehalten. Anschließend kehrt die Anwendung in die *collect*-Phase zurück. Das folgende Sequenzdiagramm visualisiert den Ablauf. Aus Gründen der Übersichtlickeit beschränkt sich das Diagramm auf die wichtigsten Aufrufe (siehe 6.4).

6.3 Fazit

Das Konzept wurde in einen lauffähigen Entwurf auf Androidbasis umgesetzt. Damit wurde gezeigt, dass ein Einsatz des Konzepts in der Praxis möglich ist, und nicht durch unlösbare technischen Hürden verhindert wird. Das Konzept konnte ohne relevante Änderungen in die Androidumgebung mit dem OSGi-Framework übernommen werden, und musste lediglich um die technischen Implementierungsdetails, beispielsweise bezüglich des Sammelns der Kontexte erweitert werden. Die Aussagekraft der Inbetriebnahme des Entwurfs ist jedoch in dieser Form beschränkt. Die Frage, ob das Konzept aber definitiv die definierten Anforderungen erfüllen kann, lässt sich damit nicht hinreichend beantworten. Dazu bedarf es einer systematischeren Vorgehensweise - idealerweise in einer simulierten Umgebung.

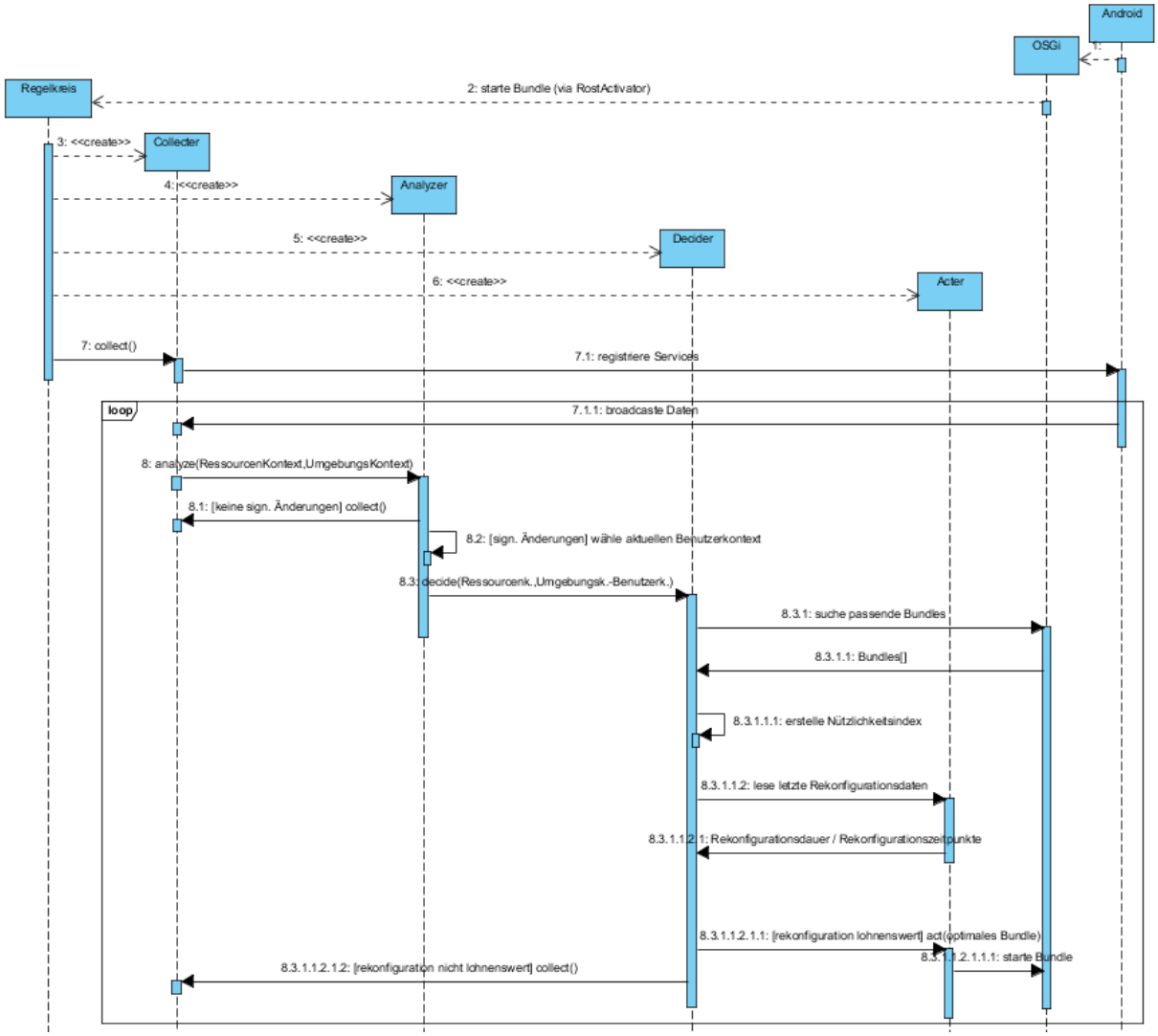

Abbildung 6.4: Sequenzdiagramm ROSt

Kapitel 7

Simulation und Verifikation

Um zu verifizieren ob das Konzept und der Entwurf die gestellten Anforderungen erfüllen, wird der Entwurf aus dem vorherigen Kapitel in eine Simulationsumgebung eingebettet. Simulationsumgebungen gestatten, mit einem strukturierten Ansatz die Parameter der Kontexte gezielt zu verändern, und dadurch die Funktionsweise der Anwendung zu prüfen. Im Vergleich mit dem Simulator und dem Android-Endgerät ermöglicht die Simulationsumgebung also eine systematischere und umfassendere Verifikation. Zum Schluss werden die durchgeführten Versuche auswertet, und die Ergebnisse mit den anfangs aufgestellten Anforderungen verglichen und bewertet.

7.1 Aufbau der Simulationsumgebung

Durch die Simulationsumgebung müssen sich die Parameter der Kontexte von außen steuern lassen. Aufgrund dessen werden die Adapter zur Erfassung der Umgebungskontexte durch Adapter der Simulationsumgebung ausgetauscht. Abbildung 7.1 visualisiert den Aufbau. Zu den bisher realisierten Komponenten wird eine protokollierende Komponente benötigt um den Programmablauf nachvollziehen zu können. Zudem braucht man eine Komponente die die einzelnen Szenarien simuliert, welche dann letztlich dem Sensor-Adapter die Simulationsdaten liefert. In den folgenden Abschnitten werden die Versuchsaufbaue erläutert.

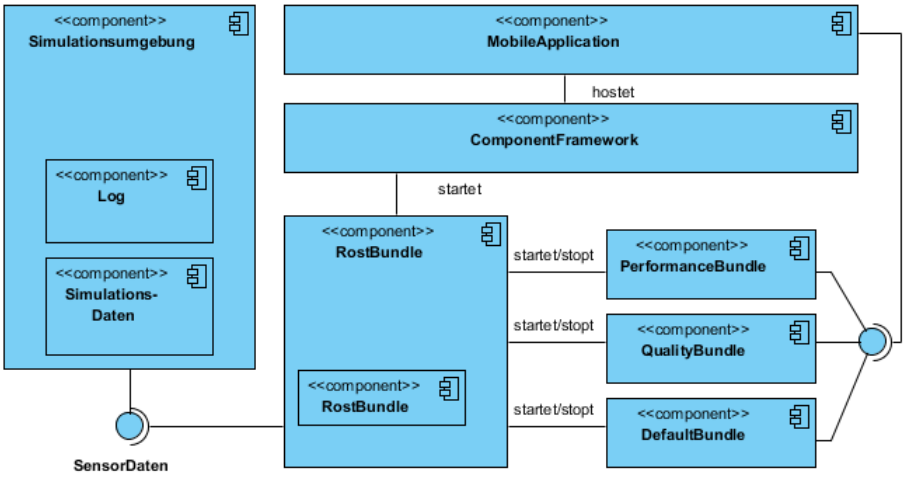

Abbildung 7.1: Aufbau Simulationsumgebung

7.2 Versuche

Die Simulation beinhaltet drei Versuchsaufbaue. Innerhalb der ersten zwei Versuchsaufbaue werden die in Kapitel (siehe 5.1.2) definierten Anwendungsfälle nacheinander mit zwei unterschiedlich langen Rekonfigurationsdauern ausgeführt. Beim dritten Versuchsaufbau werden vorher nicht definierte Varianten der Kontexte simuliert. Die *Logging*-Komponente der Simulationsumgebung protokolliert dabei den Ablauf. Zur Erinnerung werden zunächst die erhobenen Benutzerkontexte aufgeführt:

Szenario	$Benutzer_{Zeit}$	$Benutzer_{Verb}$	$Benutzer_{Richtigk}$
Fachl. Sz. 1 „Werft“	hoch (3)	mittel (2)	wenig (1)
Fachl. Sz. 2 „Hotel“	mittel (2)	mittel (2)	mittel (2)
Fachl. Sz. 3 „Büro“	wenig (1)	wenig (1)	hoch (3)
Allg. Sz. A „Akku“	hoch (3)	hoch (3)	nicht (0)
Allg. Sz. B “Internet“	nicht (0)	nicht (0)	nicht (0)
Allg. Sz. C „System“	nicht (0)	hoch (3)	nicht (0)

Tabelle 7.1: $Kontext_{Benutzer}(Benutzer_{Zeit}, Benutzer_{Verb}, Benutzer_{Richtigk})$

7.2.1 Erster Versuchsaufbau: Schnelle Rekonfiguration

In dem ersten Versuchsdurchlauf wird eine kurze Rekonfigurationszeit (100ms) simuliert. Das Protokoll wird folgend chronologisch aufgeführt und erläutert.

Anwendungsfall	$Ressourcen_{Strom}$	$Ressourcen_{Last}$	$Umgebung_{Inet}$
Fachl. Szenario 1 „Werft“	Akkubetrieb (1)	normale Last (0)	niedrig (1)

```
COLLECTER: Sammle Kontextdaten
SIMULATOR: Simulierter Ressourcenkontext: Last->0 Strom->1
SIMULATOR: Simulierter Umgebungskontext: Internet->1
ANALYZER: Noch keine vergleichbaren Kontexte gespeichert
ANALYZER: Benutzer ist im Kontext: Werft
DECIDER: Analysiere verfügbare Bundles
DECIDER: de.dloef.bundleperformance.PerformanceService: zeit=1, last=8, richtigkeit=10, strom=7, inet=2
DECIDER: de.dloef.bundlequality.QualityService: zeit=9, last=5, richtigkeit=1, strom=8, inet=5
DECIDER: de.dloef.bundlemedium.MediumService: zeit=4, last=4, richtigkeit=4, strom=4, inet=4
DECIDER: Qualitaetsindex fuer de.dloef.bundleperformance.PerformanceService ist 6.76
DECIDER: Qualitaetsindex fuer de.dloef.bundlemedium.MediumService ist 5.83
DECIDER: Qualitaetsindex fuer de.dloef.bundlequality.QualityService ist 6.58
DECIDER: Das Bundle mit dem hoechsten Index ist de.dloef.bundleperformance.PerformanceService
ACTER: Rekonfiguriere
ACTER: Rekonfiguration durchgeführt. Dauer: 100ms
...
```

Im ersten Szenario „Werft“ ist dem Benutzer die Performanz des Geräts besonders wichtig, da die Übermittlung der ersten Berichtsentwürfe schnell vonstatten gehen soll. Der *PerformanceService* wird deshalb ausgewählt. Da bisher kein anderes Bundle aktiviert ist, wird auf die Prüfung ob sich die Rekonfiguration lohnt verzichtet.

Anwendungsfall	$Ressourcen_{Strom}$	$Ressourcen_{Last}$	$Umgebung_{Inet}$
Fachl. Szenario 2 „Hotel“	Netzbetrieb (2)	normale Last (0)	mittel (2)

```
...
COLLECTER: Sammle Kontextdaten
SIMULATOR: Simulierter Ressourcenkontext: Last->0 Strom->2
SIMULATOR: Simulierter Umgebungskontext: Internet->2
ANALYZER: Kontexte haben sich geändert
ANALYZER: Benutzer ist im Kontext: Hotel
DECIDER: Analysiere verfügbare Bundles
DECIDER: de.dloef.bundleperformance.PerformanceService: zeit=1, last=8, richtigkeit=10, strom=7, inet=2
DECIDER: de.dloef.bundlequality.QualityService: zeit=9, last=5, richtigkeit=1, strom=8, inet=5
DECIDER: de.dloef.bundlemedium.MediumService: zeit=4, last=4, richtigkeit=4, strom=4, inet=4
DECIDER: Qualitaetsindex fuer de.dloef.bundleperformance.PerformanceService ist 6.86
DECIDER: Qualitaetsindex fuer de.dloef.bundlemedium.MediumService ist 9.93
DECIDER: Qualitaetsindex fuer de.dloef.bundlequality.QualityService ist 8.2
DECIDER: Das Bundle mit dem hoechstem Index ist de.dloef.bundlemedium.MediumService
DECIDER: Das neue Bundle hat einen 45,1% bessereren Index als das installierte
ACTER: Rekonfiguriere
ACTER: Rekonfiguration durchgeführt. Dauer: 100ms
...
```

Im zweiten Szenario existiert ein feste Stromversorgung. Zudem ist dem Benutzer Richtigkeit und Geschwindigkeit etwa gleich wichtig. Die *Decider*-Komponente stellt fest, dass somit der *MediumService* einen 45,1 Prozent höheren Nützlichkeitsindex hat, als der bisher verwendete *PerformanceService*. Dies bedeutet aber nicht, dass das Bundle 45,1 Prozent schneller

oder richtiger ist, sondern bezieht sich lediglich auf das Verhältnis zwischen den Indices. Die Steigerung reicht der Komponente um eine positive Entscheidung zu fällen, da der aktuelle Schwellwert auf 10 Prozent eingestellt ist. Die Rekonfigurationshäufigkeitstendenz wird noch nicht berechnet, da erst eine Rekonfiguration vorgenommen wurde.

Anwendungsfall	$Ressourcen_{Strom}$	$Ressourcen_{Last}$	$Umgebung_{Inet}$
Fachl. Szenario 3 „Büro“	Netzbetrieb (2)	normale Last (0)	hoch (3)

```
...
COLLECTER: Sammle Kontextdaten
SIMULATOR: Simulierter Ressourcenkontext: Last->0 Strom->2
SIMULATOR: Simulierter Umgebungskontext: Internet->3
ANALYZER: Kontexte haben sich geändert
ANALYZER: Benutzer ist im Kontext: Büro
DECIDER: Analysiere verfügbare Bundles
DECIDER: de.dloef.bundleperformance.PerformanceService: zeit=1, last=8, richtigkeit=10, strom=7, inet=2
DECIDER: de.dloef.bundlequality.QualityService: zeit=9, last=5, richtigkeit=1, strom=8, inet=5
DECIDER: de.dloef.bundlemedium.MediumService: zeit=4, last=4, richtigkeit=4, strom=4, inet=4
DECIDER: Qualitaetsindex fuer de.dloef.bundleperformance.PerformanceService ist 7.43
DECIDER: Qualitaetsindex fuer de.dloef.bundlemedium.MediumService ist 7.5
DECIDER: Qualitaetsindex fuer de.dloef.bundlequality.QualityService ist 13.25
DECIDER: Das Bundle mit dem hoechstem Index ist de.dloef.bundlequality.QualityService
DECIDER: Das neue Bundle ist 56,4% besser als das installierte
DECIDER: Die Rekonfigurationstendenz ist steigend. B_Strich ist 1.42 mit N=2
DECIDER: Rekonfiguriere!
ACTER: Rekonfiguration durchgeführt. Dauer: 100ms
...
```

In der Büroumgebung hat der Benutzer die höchsten Ansprüche an die Richtigkeit des Bundles. Dies schlägt sich im Nützlichkeitsindex des *QualityService* nieder, welcher in diesem Kontext mit Abstand am höchsten ist. Die Rekonfigurationstendenz wird bei dieser Rekonfiguration das erste Mal berechnet, da dazu mindestens zwei Rekonfigurationszeitpunkte benötigt werden. Die Rekonfigurationstendenz ist zwar steigend, aber die Steigung übertrifft noch nicht den gewählten Schwellwert.

Anwendungsfall	$Ressourcen_{Strom}$	$Ressourcen_{Last}$	$Umgebung_{Inet}$
Allg. Szenario A „Akku“	niedriger Batteriestand (0)	beliebige Last (X)	beliebig (X)

```
...
COLLECTER: Sammle Kontextdaten
SIMULATOR: Simulierter Ressourcenkontext: Last->0 Strom->0
SIMULATOR: Simulierter Umgebungskontext: Internet->3
ANALYZER: Kontexte haben sich geändert
ANALYZER: Benutzer ist im Kontext: Akku
DECIDER: Analysiere verfügbare Bundles
DECIDER: de.dloef.bundleperformance.PerformanceService: zeit=1, last=8, richtigkeit=10, strom=7, inet=2
DECIDER: de.dloef.bundlequality.QualityService: zeit=9, last=5, richtigkeit=1, strom=8, inet=5
DECIDER: de.dloef.bundlemedium.MediumService: zeit=4, last=4, richtigkeit=4, strom=4, inet=4
DECIDER: Qualitaetsindex fuer de.dloef.bundleperformance.PerformanceService ist 3
DECIDER: Qualitaetsindex fuer de.dloef.bundlemedium.MediumService ist 6
DECIDER: Qualitaetsindex fuer de.dloef.bundlequality.QualityService ist 2
DECIDER: Das Bundle mit dem hoechstem Index ist de.dloef.bundlemedium.MediumService
DECIDER: Rekonfiguriere!
```

```
ACTER: Rekonfiguration durchgeführt. Dauer: 100ms
...
```

Falls der Akkustand niedrig ist, wird das Bundle mit dem geringsten Stromverbrauch gewählt. Das ist in diesem Fall der *MediumService* (Akku=4). Da dies ein Sonderszenario ist und eine höhere Priorität hat, muss keine Rekonfigurationszeitpunkttendenz berechnet werden.

Anwendungsfall	$Ressourcen_{Strom}$	$Ressourcen_{Last}$	$Umgebung_{Inet}$
Allg. Szenario B “Internet“	beliebiger Batteriestand (X)	beliebige Last (X)	kein (0)

```
...
COLLECTER: Sammle Kontextdaten
SIMULATOR: Simulierter Ressourcenkontext: Last->0 Strom->2
SIMULATOR: Simulierter Umgebungskontext: Internet->0
ANALYZER: Kontexte haben sich geändert
ANALYZER: Benutzer ist im Kontext: Internet
DECIDER: Analysiere verfügbare Bundles
DECIDER: de.dloef.bundleperformance.PerformanceService: zeit=1, last=8, richtigkeit=10, strom=7, inet=2
DECIDER: de.dloef.bundlequality.QualityService: zeit=9, last=5, richtigkeit=1, strom=8, inet=5
DECIDER: de.dloef.bundlemedium.MediumService: zeit=4, last=4, richtigkeit=4, strom=4, inet=4
DECIDER: Qualitaetsindex fuer de.dloef.bundleperformance.PerformanceService ist 8
DECIDER: Qualitaetsindex fuer de.dloef.bundlemedium.MediumService ist 6
DECIDER: Qualitaetsindex fuer de.dloef.bundlequality.QualityService ist 5
DECIDER: Das Bundle mit dem hoechstem Index ist de.dloef.bundleperformance.PerformanceService
DECIDER: Rekonfiguriere!
ACTER: Rekonfiguration durchgeführt. Dauer: 100ms
...
```

Falls gerade keine oder fast keine Internetverbindung vorhanden ist, und kein Bundle installiert ist, was diesen Fall explizit behandeln kann, wird das Bundle mit dem günstigsten Bandbreitenbedarf ausgewählt. Dies ist in diesem Fall der *PerformanceService*.

Anwendungsfall	$Ressourcen_{Strom}$	$Ressourcen_{Last}$	$Umgebung_{Inet}$
Allg. Szenario C „System“	beliebiger Batteriestand (X)	hohe Last (1)	beliebig (X)

```
...
COLLECTER: Sammle Kontextdaten
SIMULATOR: Simulierter Ressourcenkontext: Last->1 Strom->2
SIMULATOR: Simulierter Umgebungskontext: Internet->3
ANALYZER: Kontexte haben sich geändert
ANALYZER: Benutzer ist im Kontext: System
DECIDER: Analysiere verfügbare Bundles
DECIDER: de.dloef.bundleperformance.PerformanceService: zeit=1, last=8, richtigkeit=10, strom=7, inet=2
DECIDER: de.dloef.bundlequality.QualityService: zeit=9, last=5, richtigkeit=1, strom=8, inet=5
DECIDER: de.dloef.bundlemedium.MediumService: zeit=4, last=4, richtigkeit=4, strom=4, inet=4
DECIDER: Qualitaetsindex fuer de.dloef.bundleperformance.PerformanceService ist 4.29
DECIDER: Qualitaetsindex fuer de.dloef.bundlemedium.MediumService ist 7.5
DECIDER: Qualitaetsindex fuer de.dloef.bundlequality.QualityService ist 3.75
DECIDER: Das Bundle mit dem hoechsten Index ist de.dloef.bundlemedium.MediumService
DECIDER: Das neue Bundle ist 50.0% besser als das installierte
DECIDER: Die Rekonfigurationstendenz ist steigend. B_Strich ist 1.12 mit N=3
DECIDER: Rekonfiguriere!
ACTER: Rekonfiguration durchgeführt. Dauer: 100ms
SIMULATOR: Simulation beendet
```

Aufgrund der geringen Systembelastung von vier, wird in diesem Fall wieder der *Medium-Service* gewählt. Die Rekonfigurationstendenz ist zwar steigend, aber die Steigung übertrifft noch nicht den gewählten Schwellwert, welcher in diesem Fall in Abhängigkeit von der Rekonfigurationsdauer zwei gewesen wäre.

7.2.2 Zweiter Versuchsaufbau: Langsame Rekonfiguration

Im zweiten Versuchsaufbau wird eine längere Rekonfigurationszeit simuliert. Die Abfolge und Anzahl der Anwendungsfälle wird dabei angepasst, damit gezielt die Mechanismen aktiviert werden, die die Rekonfigurationszeit in die Adaptionsentscheidung mit einbeziehen.

Anwendungsfall	$Ressourcen_{Strom}$	$Ressourcen_{Last}$	$Umgebung_{Inet}$
Allg. Szenario C „System“	beliebiger Batteriestand (X)	hohe Last (1)	beliebig (X)

```
COLLECTER: Sammle Kontextdaten
SIMULATOR: Simulierter Ressourcenkontext: Last->1 Strom->2
SIMULATOR: Simulierter Umgebungskontext: Internet->3
ANALYZER: Kontexte haben sich geändert
ANALYZER: Benutzer ist im Kontext: System
DECIDER: Analysiere verfügbare Bundles
DECIDER: de.dloef.bundleperformance.PerformanceService: zeit=1, last=8, richtigkeit=10, strom=7, inet=2
DECIDER: de.dloef.bundlequality.QualityService: zeit=9, last=5, richtigkeit=1, strom=8, inet=5
DECIDER: de.dloef.bundlemedium.MediumService: zeit=4, last=4, richtigkeit=4, strom=4, inet=4
DECIDER: Qualitaetsindex fuer de.dloef.bundleperformance.PerformanceService ist 4.29
DECIDER: Qualitaetsindex fuer de.dloef.bundlemedium.MediumService ist 7.5
DECIDER: Qualitaetsindex fuer de.dloef.bundlequality.QualityService ist 3.75
ECIDER: Das Bundle mit dem hoechsten Index ist de.dloef.bundlemedium.MediumService
DECIDER: Rekonfiguriere!
ACTER: Rekonfiguration durchgeführt. Dauer: 1000ms
...
```

Zunächst wird der *MediumService* gewählt. Die Rekonfigurationsdauer beträgt hierbei 1000ms.

Anwendungsfall	$Ressourcen_{Strom}$	$Ressourcen_{Last}$	$Umgebung_{Inet}$
Fachl. Szenario 1 „Werft“	Akkubetrieb (1)	normale Last (0)	niedrig (1)

```
COLLECTER: Sammle Kontextdaten
SIMULATOR: Simulierter Ressourcenkontext: Last->0 Strom->1
SIMULATOR: Simulierter Umgebungskontext: Internet->1
ANALYZER: Noch keine vergleichbaren Kontexte gespeichert
ANALYZER: Benutzer ist im Kontext: Werft
DECIDER: Analysiere verfügbare Bundles
DECIDER: de.dloef.bundleperformance.PerformanceService: zeit=1, last=8, richtigkeit=10, strom=7, inet=2
DECIDER: de.dloef.bundlequality.QualityService: zeit=9, last=5, richtigkeit=1, strom=8, inet=5
DECIDER: de.dloef.bundlemedium.MediumService: zeit=4, last=4, richtigkeit=4, strom=4, inet=4
DECIDER: Qualitaetsindex fuer de.dloef.bundleperformance.PerformanceService ist 6.76
DECIDER: Qualitaetsindex fuer de.dloef.bundlemedium.MediumService ist 5.83
```

```
DECIDER: Qualitaetsindex fuer de.dloef.bundlequality.QualityService ist 6.58
DECIDER: Das Bundle mit dem hoechsten Index ist de.dloef.bundleperformance.PerformanceService
DECIDER: Das neue Bundle ist 9.8% besser als das installierte
DECIDER: Durchschnittliche Rekonfigurationszeit: 1000ms.
DECIDER: Prozentualer Schwellwert für Rekonfiguration: 30%
DECIDER: Rekonfiguration nicht lohnenswert
...
```

Beim Übergang zum Szenario Werft wird zunächst wie im ersten Versuchsaufbau der *PerformanceService* ausgewählt. Dabei ist es für diesen Kontext 9.8 Prozent höher bewertet als der *MediumService*, der gerade verwendet wird. Der Algorithmus, der bestimmt, ob eine Rekonfigurationsentscheidung lohnenswert ist, berechnet dabei aus der durchschnittlichen Rekonfigurationszeit einen Schwellwert ($DurchschnittlicheRekonfigurationsdauer/33$), um den das neue Bundle besser geeignet sein muss. In diesem Fall müsste das neue Bundle mindestens 30 Prozent besser als das alte sein. Die Berechnungsvorschrift ist dabei relativ einfach gehalten und demonstriert lediglich an welcher Stelle die Rekonfigurationsdauer einbezogen werden sollte.

7.2.3 Dritter Versuchsaufbau: Unvorhergesehene Szenarien

Im dritten Versuchsaufbau soll getestet werden, ob das Konzept auch unvorhergesehene Szenarien verarbeiten kann. Dazu werden die Kontextparameter eines vorgefertigen Anwendungsfalls gezielt modifiziert. Im Unterschied zu den vorangegangenen Versuchen werden nun die Abweichungen der Kontexte im Log aufgeführt.

Anwendungsfall	$Ressourcen_{Strom}$	$Ressourcen_{Last}$	$Umgebung_{Inet}$
Fachl. Szenario 3 „Büro“	Netzbetrieb (2)	normale Last (0)	hoch (3)

```
COLLECTER: Sammle Kontextdaten
SIMULATOR: Simulierter Ressourcenkontext: Last->0 Strom->2
SIMULATOR: Simulierter Umgebungskontext: Internet->3
ANALYZER: Noch keine Vergleichsdaten
ANALYZER: Büro: AbweichungInet 0.0 AbweichungLast 0.0 AbweichungStrom 0.0 -> Summe: 0.0
ANALYZER: Werft AbweichungInet 0.66 AbweichungLast 0.0 AbweichungStrom 0.33 -> Summe: 0.99
ANALYZER: Hotel AbweichungInet 0.33 AbweichungLast 0.0 AbweichungStrom 0.0 -> Summe: 0.33
ANALYZER: Kontext Akku ist nicht eingetreten (AbweichungStrom = 0.66)
ANALYZER: Kontext Last ist nicht eingetreten (AbweichungLast = 0.33)
ANALYZER: Kontext Internet ist nicht eingetreten (AbweichungInternet = 0.66)
ANALYZER: Geringste Abweichung beträgt 0.0
ANALYZER: Benutzer ist im Kontext: Büro
(...)
COLLECTER: Sammle Kontextdaten!
...
```

Wie in den vorherigen Versuchen erkennt der *Analyzer* den Kontext „Büro“ korrekt. Die Sonderzustände werden ausgeschlossen, da sie sich nur durch eine Kontexteigenschaft (Batteriestand, Last, oder Internet) definieren. Falls der Akkustand deshalb nicht niedrig ist, wird auch der Kontext für den niedrigen Akkustand nicht aktiviert. Im nächsten Durchlauf wird der aktuelle Kontext leicht modifiziert.

Anwendungsfall	$Ressourcen_{Strom}$	$Ressourcen_{Last}$	$Umgebung_{Inet}$
Modifiziertes Szenario „Büro“ 1	Akkubetrieb (1)	normale Last (0)	hoch (3)

```
...
SIMULATOR: Simulierter Ressourcenkontext: Last->0 Strom->1
SIMULATOR: Simulierter Umgebungskontext: Internet->3
ANALYZER: Kontexte haben sich geändert
ANALYZER: Büro AbweichungInet 0.0 AbweichungLast 0.0 abweichungStrom 0.5 -> Summe: 0.5
ANALYZER: Werft AbweichungInet 0.6 AbweichungLast 0.0 abweichungStrom 0.16 -> Summe: 0.76
ANALYZER: Hotel AbweichungInet 0.33 AbweichungLast 0.0 AbweichungStrom 0.5 -> Summe: 0.83
ANALYZER: Kontext Akku ist nicht eingetreten (AbweichungStrom = 0.16)
ANALYZER: Kontext Last ist nicht eingetreten (AbweichungLast = 0.33)
ANALYZER: Kontext Internet ist nicht eingetreten (AbweichungInternet = 0.66)
ANALYZER: Beste Abweichung beträgt 0.5
ANALYZER: Benutzer ist im Kontext: Büro
(...)
COLLECTER: Sammle Kontextdaten!
...
```

Im Gegensatz zum unmodifizierten Anwendungsfall wurde die Stromzufuhr auf Batteriebetrieb umgestellt. Der Kontext Büro weist dabei die geringsten Abweichungen auf und wurde ausgewählt. An diesem Beispiel sieht man, dass die allgemeinen Szenarien (geringer Akkustand, kein Internet, hohe Systemlast) gesondert zu behandeln sind, und nicht durch die reine Abweichung abgedeckt werden können. Der Kontext Akku weist nämlich auch in der Summe die geringsten Abweichungen auf, da die Kontextparameter *Last* und *Internet* als beliebig gekennzeichnet wurden.

Anwendungsfall	$Ressourcen_{Strom}$	$Ressourcen_{Last}$	$Umgebung_{Inet}$
Modifiziertes Szenario „Büro“ 2	Netzbetrieb (2)	hohe Last (1)	hoch (3)

```
COLLECTER: Sammle Kontextdaten!
SIMULATOR: Simulierter Ressourcenkontext: Last->1 Strom->2
SIMULATOR: Simulierter Umgebungskontext: Internet->3
ANALYZER: Kontexte haben sich geändert
ANALYZER: Büro AbweichungInet 0.0 AbweichungLast 0.0 AbweichungStrom 0.0 -> Summe: 0.0
ANALYZER: Werft AbweichungInet  0.66 AbweichungLast 0.0 AbweichungStrom 0.33 -> Summe: 0.99
ANALYZER: Hotel AbweichungInet  0.33 AbweichungLast 0.0 AbweichungStrom 0.0 -> Summe: 0.33
ANALYZER: Kontext Akku ist nicht eingetreten (AbweichungStrom = 0.66)
ANALYZER: Kontext Last ist eingetreten (AbweichungLast = 0.0)
ANALYZER: Kontext Internet ist nicht eingetreten (AbweichungInternet = 0.66)
ANALYZER: Beste Abweichung beträgt 0.0
ANALYZER: Benutzer ist im Kontext: System
(...)
Simulation beendet
```

In diesem Beispiel gibt es zwei Kontexte, die die Abweichungssumme 0 haben. Im Falle Büro kommt das zustande, da hier die Last als „egal“ gekennzeichnet wurde. Da der Kon-

text „Last“ aber eine höhere Priorität hat wird er dem Kontext „Büro“ vorgezogen. Deshalb könnte der Parameter „Last“ auch mit 0 für den Kontext „Büro“ gekennzeichnet werden ohne Auswirkungen zu haben.

7.3 Auswertung

Folgend werden die einzelnen Versuche in der Simulationsumgebung ausgewertet und mit den Anforderungen verglichen. Die Anforderungen wurden in Abschnitt 5.1.2 mit Hilfe von Anwendungsfällen definiert. Zusätzlich wurden optionale Anforderungen erhoben, die ebenfalls betrachtet werden. Dabei soll verifiziert werden, ob die ausgewählten adaptiven Mechanismen des Konzepts für die Anwendungsfälle geeignet sind.

7.3.1 Erster Versuchsaufbau

Versuchsaufbau eins sollte verifizieren, ob das Konzept im Rahmen der Anwendungsfälle geeignet ist. Dabei konnten die Mechanismen für die kompositionelle Adaption, die Nützlichkeitsfunktion (nutzenorientierte Adaptionsentscheidung), die Asynchronität und die Reaktion auf Kontextwechsel erprobt werden. Der Versuch zeigte, dass zunächst jeder Kontext erkannt wurde, und anhand der Nützlichkeitsindices das passende Bundle ausgewählt wurde. Wie genau die Entscheidung zustande kam, wurde im obigen Abschnitt erläutert. Da die Anwendungsfälle alle angemessen behandelt wurden, konnten die bisher erprobten Mechanismen die Anforderungen erfüllen. Dabei kam es insbesondere darauf an, dass die kontextbasierte Rekonfigurationsentscheidung nachvollziehbar war.

7.3.2 Zweiter Versuchsaufbau

Im zweiten Versuchsaufbau wurde eine lange Rekonfigurationsdauer simuliert, um zu verifizieren, dass der Mechanismus die Auswirkungsvorhersage berücksichtigt. Falls der Mechanismus die Rekonfiguration als nicht lohnenswert erachtet, sollte diese nicht durchführt werden. Diese Vorhersage ist notwendig, da die kompositionelle Adaption eine aufwendige Adaptionsart ist und mitunter deswegen manchmal nicht lohnenswert sein kann. Es konnte demonstriert werden, dass das Konzept in der Lage ist, die Rekonfigurationsdauer miteinzubeziehen, was innerhalb einiger Umstände ein wichtiger Mechanismus ist. Damit konnte auch die optionale Anforderung nach der Angemessenheit (vgl. 5.1.4) erfüllt werden. Der passende Quellcode befindet sich im Anhang D.

7.3.3 Dritter Versuchsaufbau

Aus den Anwendungsfällen und den Self-*-Eigenschaften des zu enwickelnden Systems ergab sich, dass eine nicht-antizipierte Adaption vom Konzept unterstützt werden muss. Mit Versuchsaufbau drei wurden somit vorher nicht bekannte Umgebungen simuliert, um zu verifizieren, ob das Konzept auch hier funktionsfähig bleibt. Das Konzept berechnete die Abweichungen der aktuellen Kontexte von den hinterlegten Kontexten und wählte den passendsten Kontext aus. Dies hat in diesem Experiment gut funktioniert. Jedoch hängt das auch von der Art der hinterlegten Kontexte und Benutzeranforderungen ab. Sonderfälle, die die Gebrauchstauglichkeit der Anwendung stark gefährden, müssen höher priorisiert werden und betreffen in der Regel nur einen Parameter eines Kontextes. Innerhalb dieses Kontextmodells kann jeder Parameter einen Wert einnehmen, der zu einem Sonderfall führt. Es liegt nahe, dass dies für alle Parameter der Kontexte gilt, die wichtig für die Funktionalität des Geräts sind.

Die optionale Anforderung „Robustheit“, die verlangt dass selbst in nicht vorhersehbaren Situationen die Anwendung angemessen reagiert, konnte mit Hilfe des Versuchaufbaus verifiziert werden. In einigen Sonderfällen kann dabei möglicherweise ein nicht ganz passender Kontext und somit ein nicht optimales Bundle ausgewählt werden. Dabei lässt sich aber argumentieren, dass man diese Sonderfälle jedoch auch hinterlegen könnte und somit sicherstellen, dass dieser Kontext ausgewählt wird. Den Quellcode der Funktion die die Abweichung bestimmt und die Sonderfälle berücksichtigt, findet man im Anhang unter C.

7.4 Fazit

Die Verifikation des Entwurfs in der Simulationsumgebung konnte belegen, dass das Konzept die Anforderungen erfüllen kann. Die für die Anwendungsfälle ausgewählten adaptiven Mechanismen haben sich als gute konzeptionelle Basis bewiesen. Alle Anwendungsfälle konnten durch diese, wie erwartet, nachvollziehbar bearbeitet werden. Auch die optionalen Anforderungen konnten erfüllt werden (vgl. 5.1.4): durch die Annotationen lassen sich bereits vorhandene Komponenten wiederverwenden oder die Anwendung auch nach Auslieferung erweitern (*Erweiterbarkeit*). Durch das Komponentenframework ist eine einfache und schnelle Rekonfiguration möglich. Die Rekonfigurationskosten und die Häufigkeit der Rekonfigurationen werden beachtet, um die Rekonfiguration nur durchzuführen wenn sie angemessen scheint (*Angemessenheit*). Auch unvorhergesehene Szenarien werden bestmöglich behandelt und die naheliegensten Kontexte ausgewählt (*Robustheit*). Die Rekonfiguration werden asynchron ausgeführt, und geschehen autonom im Hintergrund der Anwendung (*Transparenz*). Die Mechanismen und die Einbettung in den geschlossenen Regelkreis sind also vorbehaltlos zu empfehlen. Jedoch ist an einigen Stellen noch Optimierungspotential vor-

handen. Für die Auswirkungsvorhersage, die Nützlichkeitsberechnung und die Rekonfigurationshäufigkeitstendenz könnten potentiell unter gewissen Szenarien andere Berechnungsvorschriften oder Schwellwerte besser geeignet sein, was an der fachlichen Bewertung des Konzepts jedoch nichts ändert.

Kapitel 8

Schluss

Dieses Kapitel gibt dem Leser einen Rückblick auf die verfasste Arbeit. Damit sollen Brücken zwischen den Kapiteln geschlagen werden: von der Motivation über die Grundlagen zum Konzept, und über den Entwurf zum Ergebnis und dem Vergleich mit den Anforderungen. Zunächst steht dabei eine kurze Zusammenfassung der einzelnen Kapitel und deren Verbindungen, dann das Fazit, und zu guter Letzt der Ausblick auf eine Weiterentwicklung.

8.1 Zusammenfassung

Die vorliegende Arbeit hatte das Ziel mit Hilfe von adaptiven Mechanismen und Techniken, die Anpassung mobiler Anwendungen zu unterstützen. Diese Anforderung entspringt der Flexibilität und Portabilität moderner mobiler Endgeräte. Mit den Grundlagen wurden die unterschiedlichen Mechanismen klassifiziert und deren Eigenschaften aufgezeigt. Für die Entwicklung eines fundierten Konzepts musste nun ein realitätsnahes Szenario erstellt werden, mit welchem sich die Anforderungen an das Konzept besonders gut demonstrieren ließen. So konnten anhand dieses Szenarios Anwendungsfälle und Benutzeranforderungen erhoben, und Parameter der Umgebungs-, Ressourcen- und Benutzerkontexte definiert werden. Auf Basis der identifizierten benötigten *Self-**-Eigenschaften, konnten die passenden adaptiven Mechanismen ausgewählt und zu einem Regelkreis zusammengeschlossen werden. Dabei fiel die Entscheidung zu gunsten kompositioneller nicht-antizipierter Adaption, mit nutzenorientierter starker Adaptionsentscheidung und asynchronen, reaktiven Auslösern. Der Regelkreis beinhaltet vier Phasen mit unterschiedlichen Funktionen, wie der *Decider*, der beispielsweise die Entscheidungsfindung ob eine Rekonfiguration lohnenswert ist, beinhaltet. Durch die Art des Rekonfigurationsansatzes mussten die Parameter der Kontexte miteinander verrechnet werden, um so den Nützlichkeitsindex der Komponenten zu bestimmen. Anhand einer Auswirkungsvorhersage auf Basis der Rekonfigurationsdauer und -abstände konnte abschließend entschieden werden, ob sich die Rekonfiguration lohnt. Als Demonstration wurde ein lauffähiger Entwurf des Konzepts mit den weit verbreiteten Technologien

Android und OSGi entwickelt. Das Komponentenframework wurde eingesetzt, da die Analyse ergeben hat, dass kompositionelle Rekonfiguration die passendste Adaptionsart ist, und ein solches Framework ideal für den Austausch von Komponenten ist. Der Entwurf zeigte, dass die beiden Technologien gut für die Problemstellung geeignet sind, das Konzept umsetzbar ist und die Anforderungen erfüllt. Um die Eignung des Konzepts zu verifizieren, wurde der Entwurf der Anwendung in eine Simulationsumgebung eingebettet. In dieser Umgebung wurden mit drei unterschiedlichen Versuchsaufbauen unterschiedliche Umgebungen und Zustände simuliert, die unterschiedliche Aspekte der Anwendung ansprachen. Es konnte demonstriert werden, dass die Mechanismen auch in unvorhergesehenen Umgebungen gut funktionieren, und die definierten Anwendungfälle erfüllen. Somit können die ursprünglich definierten Eigenschaften *self-optimizing*, *self-configuring*, und *self-awareness* mit Hilfe des Konzepts realisiert werden.

8.2 Fazit

Die Anpassung der Funktionalität einer mobilen Anwendung mittels adaptiven Mechanismen funktioniert. Die ausgewählten adaptiven Mechanismen haben sich in dem ersten Entwurf gut bewährt, obgleich an vielen Stellen noch Optimierungspotential vorhanden ist. Ziele der Arbeit waren das Entwicklen eines Konzepts, welches für einen definierten Typ von mobiler Anwendung (siehe 5.1.3) eine reflexive Optimierung und Rekonfiguration durch adaptive Mechanismen ermöglicht. Das Konzept musste nicht nur eine Reihe von praxisnahen Anwendungsfällen erfüllen, sondern idealerweise auch optionale Anforderungen, die den Einsatz eines solchen Konzepts in der Praxis begünstigen können (siehe 5.1.2 und 5.1.4). Je nach Anwendung kann der Einsatz des Konzepts einen hohen Nutzen mit sich bringen. Jedoch ist dieser, und der Nutzen der adaptiven Mechanismen klar vom Einsparpotential, bezüglich Ressourcen oder Internetkapazität, abhängig. Falls die zur Verfügung stehenden Komponenten grundsätzlich alle ein sehr gutes Verbrauchsverhalten aufweisen, oder die Funktionalität grundsätzlich nicht besonders hoch sein muss, ist der Nutzen einer Adaption dementsprechend gering. Falls die potentiellen Komponenten jedoch größere Unterschiede in Funktionsgüte und Verbrauchsverhalten aufweisen, ist eine dynamische Rekonfiguration durchaus sinnvoll. Am Beispiel des Entwurfs konnte gezeigt werden, dass eine solche Rekonfiguration sehr kostengünstig und transparent für den Benutzer durchgeführt werden kann, und kein Hindernis für das Konzept darstellt. Durch die geringen Modifikationen, die an den OSGi-Bundles durchgeführt werden müssen - die Annotation der Serviceklasse - wäre dies auch in der Praxis kostengünstig zu realisieren. Somit konnten auch die optionalen Anforderungen mit diesem Konzept erfüllt werden.

8.3 Ausblick

Adaptive Technologie tangiert viele unterschiedliche fachliche Domänen. Somit kann das bisherige Konzept an ebensovielen Stellen auf viele Arten optimiert und erweitert werden. Eine interessante Erweiterung für die mobile Anwendung (ROSt) wäre das Verwenden eines semantischen Ansatzes für die funktionale Anforderung eine passende Lösung zu finden. Damit könnten in vorher nicht zu antizipierenden Situationen Komponenten von Drittherstellern eingesetzt werden. Oft scheitert dies ansonsten an der Inkompabilität der Schnittstellen. Bei der Verwendung von wiederverwendbaren Komponenten kann zudem dem Programmierer viel Entwicklungsaufwand erspart werden. Damit diese Flexibilität nutzbar ist, muss jedoch ein tieferes Verständnis der Funktionalitäten gegeben sein. Nur so können die verschiedenen Realisierungsoptionen erkannt werden. Unabhängig von der Schnittstelle wird auch eine bestimmte Semantik der benötigten Funktionalitäten des Dienstes erwartet. Hier muss eine umfassende Flexibilität gegeben sein, da sonst keine sinnvolle Interoperabilität erreicht wird.

Die meisten bei ROSt eingesetzten Parameter und konkreten Anforderungsgewichtungen sind exemplarisch gewählt. An dieser Stelle gibt es noch hohes Optimierungspotential, ebenso bei der Berechnung der Qualitätsindices. Die verwendeten Formeln können bisher nur ein Anhaltspunkt sein, auf dem aufgebaut werden kann. Momentan ist die Bestimmung des geeigneten Benutzerkontexts über die aktuellen Umgebungskontexte geregelt. Es wäre denkbar dem Benutzer weitere Möglichkeiten zu bieten oder die Bestimmung zusätzlich über weitere Aspekte zu erweitern. Beispiele wären hier, ob ein bekanntes WLAN-Netz in der Nähe ist oder per GPS eine bekannte Position ermittelt werden kann. Über die Berücksichtigung wiederkehrender Wege zu ähnlichen Tageszeiten (beispielsweise der Weg zur Arbeit) könnte das Programm sich selbst bezüglich der Vorhersage des Rekonfigurationsaufkommens optimieren.

Weitere Möglichkeiten bieten sich in der Frage, wie sich die Qualitätsparameter für die Komponenten messen lassen. Hierbei sind Verfahren denkbar, die der Entwickler nach der Entwicklung seiner Komponente auf den Quellcode anwenden kann, sowie Verfahren, die mit Hilfe von Profilen, die über die Komponenten zur Laufzeit erstellt werden, die Qualitätseigenschaften dynamisch ermitteln. Eine Kombination von beidem ist natürlich auch denkbar.

Kapitel 9

Anhang

A Deployment-Befehle

Alle Werkzeuge sind durch die Installation des Android-SDKs verfügbar.

```
# Erzeugen des DalvikVM-Bytecodes eines OSGi-Bundles
dx --dex --output=classes.dex RostBundle-1.0-SNAPSHOT.jar

# Hinzufügen des Bytecodes zu einem OSGi-Bundle
aapt add RostBundle-1.0-SNAPSHOT.jar classes.dex

# Übertragen eines OSGi-Bundles zum Androidgerät
adb push RostBundle-1.0-SNAPSHOT.jar /data/felix

# Dalvik Debug Monitor Server (DDMS) für Log-Ausgaben
ddbs
```

B Klassendiagramm ROSt

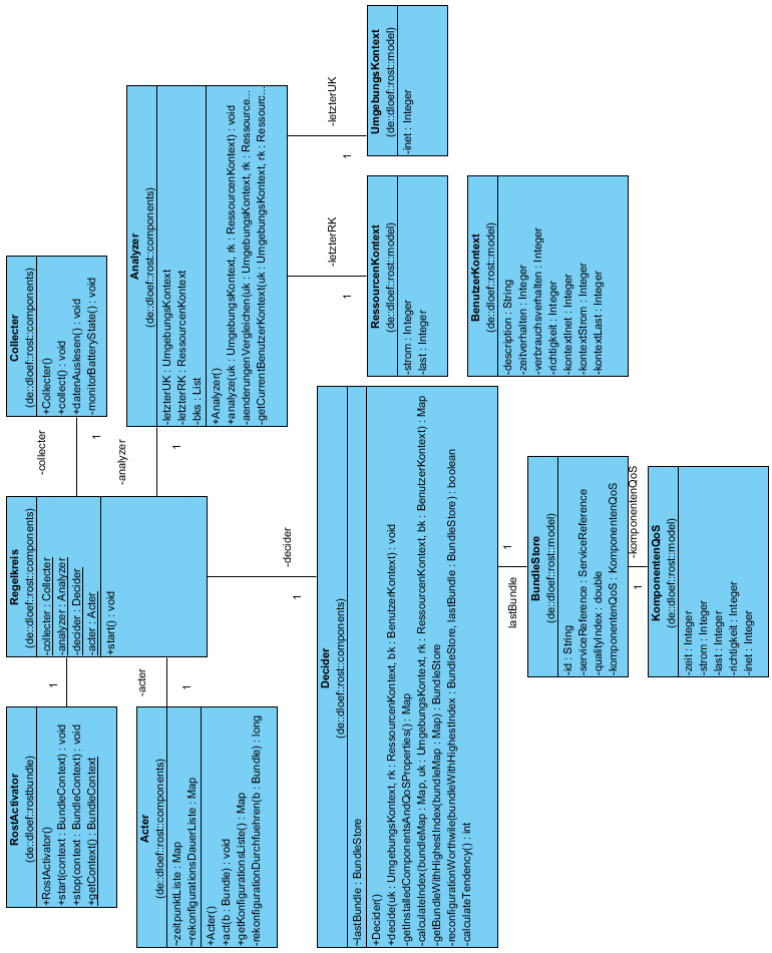

Abbildung B.1: ROSt-Bundle Klassendiagramm

C Berechnung der Kontextabweichungen

```
/**
 * @param uk der aktuelle Umgebungskontext
 * @param rk der aktuelle Ressourcenkontext
 * @return den passendsten Benutzerkontext
 */
private BenutzerKontext getCurrentBenutzerKontext(UmgebungsKontext uk, RessourcenKontext rk) {
    // Normierung der Kontexte
    double inetNorm = (double) uk.getInet() / (double) UmgebungsKontext.INET_MAX;
    double stromNorm = (double) rk.getStrom() / (double) RessourcenKontext.BATTERY_MAX;
    double lastNorm = (double) rk.getLast() / (double) RessourcenKontext.SYSTEM_MAX;
    double besteAbweichung = Double.MAX_VALUE;
    BenutzerKontext besterBenutzerKontext = bks.get(0);
    // Benutzerkontexte durchgehen
    for (BenutzerKontext benutzerKontext : bks) {
        double kontextInetNorm = (double) benutzerKontext.getKontextInet()
         / (double) BenutzerKontext.MAX;
        double kontextStromNorm = (double) benutzerKontext.getKontextStrom()
          / (double) BenutzerKontext.MAX;
        double kontextLastNorm = (double) benutzerKontext.getKontextLast()
          / (double) BenutzerKontext.MAX;
        double abweichungInet = Math.abs(inetNorm - kontextInetNorm);
        double abweichungLast = Math.abs(lastNorm - kontextLastNorm);
        double abweichungStrom = Math.abs(stromNorm - kontextStromNorm);
        double abweichungSumme = abweichungInet + abweichungLast + abweichungStrom;

        if (abweichungSumme < besteAbweichung) {
            besteAbweichung = abweichungSumme;
            besterBenutzerKontext = benutzerKontext;
        }
    }
    BenutzerKontext retval = pruefeSonderFaelle(besterBenutzerKontext, uk, rk);
    return retval;
}

/**
 * Gibt den Sonderkontext zurueck falls er eingetroffen ist
 * @param bk der bisher beste BenutzerKontext
 * @param uk der aktuelle Umgebungskontext
 * @param rk der aktuelle Ressourcenkontext
 * @return den passendsten Benutzerkontext
 */
private BenutzerKontext pruefeSonderfaelle
 (BenutzerKontext bk, UmgebungsKontext uk, RessourcenKontext rk) {
    BenutzerKontext besterBenutzerKontext = bk;
    double inetNorm = (double) uk.getInet() / (double) UmgebungsKontext.INET_MAX;
    double stromNorm = (double) rk.getStrom() / (double) RessourcenKontext.BATTERY_MAX;
    double lastNorm = (double) rk.getLast() / (double) RessourcenKontext.SYSTEM_MAX;
    for (BenutzerKontext benutzerKontext : sonderbks) {
        if (inetNorm == 0 && benutzerKontext.getDescription().equals("Internet")) {
            besterBenutzerKontext = benutzerKontext;
        } else if (lastNorm == 1 && benutzerKontext.getDescription().equals("System")) {
            besterBenutzerKontext = benutzerKontext;
        } else if (stromNorm == 0 && benutzerKontext.getDescription().equals("Akku")) {
            besterBenutzerKontext = benutzerKontext;
        }
    }
```

```
    return besterBenutzerKontext;
}
```

D Berechnung ob die Rekonfiguration lohnenswert ist

```
/* Berechnet ob die Rekonfiguration lohnenswert ist. Dabei wird das beste Bundle mit
* dem letzten Bundle ins Verhältnis gesetzt.
* @return ob die Rekonfiguration lohnt
*/
private boolean rekonfigurationLohnenswert(BundleStore newBundle, BundleStore lastBundle) {
    double prozentDiff = (newBundle.getQualityIndex() -
     lastBundle.getQualityIndex()) / lastBundle.getQualityIndex();
    // falls das neue Bundle den Schwellwert übertrifft und
    // die Tendenz der linearen Regression positiv ist gib true zurueck
    if (prozentDiff * 100 > (this.rekonfigurationDurchschnitt() / 33) && this.berechneTendenz() > 0) {
        return true;
    } else {
        return false;
    }
}

/**
 * Berechnet den Durchschnitt der letzten Rekonfigurationen
 * @return
 */
private double rekonfigurationDurchschnitt() {
    Map<Long, Long> konfigurationsListe = Regelkreis.getActer().getKonfigurationsListe();
    long sum = 0L;
    for (Long l : konfigurationsListe.values()) {
        sum += l;
    }
    double durchschnitt = ((double) sum) / konfigurationsListe.size();
    return durchschnitt;
}


/*
 * Berechnet mittels linerarer Regression die Entwicklung der Rekonfigurations
 * -abstände
 */
private int berechneTendenz() {
    List<Long> liste = Regelkreis.getActer().getZeitpunktListe();
    long N = liste.size();
    System.out.println("N: " + N);
    // Nur berechnen falls N groesser 1 ist
    if (N > 1) {
        List<Long> m = new ArrayList<Long>();
        List<Long> mStrich = new ArrayList<Long>();
        long[] yListe = new long[liste.size() + 1];
        int x = 1;
        int i = 0;
        for (Long zeitpunkt : liste) {
            yListe[i++] = System.currentTimeMillis() - zeitpunkt;
        }
        yListe[yListe.length - 1] = System.currentTimeMillis();

        // erstelle Steigungen
        for (int j = 1; j < yListe.length; j++) {
            m.add(yListe[j] - yListe[j - 1]);
        }

        // m strich
```

```
        for (int j = 1; j <= m.size(); j++) {
            mStrich.add(1 / j * getSum(m, j));
        }
        long sum1 = getSum1(mStrich);
        long sum2 = getSum2(N);
        long sum3 = getSum3(m);
        long sum4 = getSum4(N);
        long sum5 = getSum5(N);
        double durchn = 1.0 / N;
        double bstrich = (durchn * sum1 - (durchn * sum2 * durchn * sum3)) /
          (durchn * sum4 - (durchn * sum5));
    }
    return 1;
}

private long getSum1(List<Long> mStrich) {
    Long ret = 0L;
    int cnt = 1;
    for (Long long1 : mStrich) {
        ret += long1 * cnt;
        cnt++;
    }
    return ret;
}

private long getSum2(long N) {
    int ret = 0;
    for (int i = 1; i <= N; i++) {
        ret += i;
    }
    return (long) ret;
}

private long getSum3(List<Long> mStrich) {
    Long ret = 0L;
    for (Long long1 : mStrich) {
        ret += long1;
    }
    return ret;
}

private long getSum4(long N) {
    int ret = 0;
    for (int i = 1; i <= N; i++) {
        ret += i * i;
    }
    return (long) ret;
}

private long getSum5(long N) {
    int ret = 0;
    for (int i = 1; i <= N; i++) {
        ret += i;
    }
    return (long) ret * (long) ret;
}

private long getSum(List<Long> m, int j) {
    long ret = 0;
```

```
    for (int i = 0; i < j; i++) {
        ret += m.get(i);
    }
    return ret;
}
```

E Inhalt der CD-ROM

Der Arbeit ist eine CD-ROM beigefügt, die folgende Verzeichnisstruktur beinhaltet:

- **PDF** beinhaltet die vorliegende Arbeit als PDF.
- **Komponenten** beinhaltet die Anwendungskomponenten als jar-Archive.
- **HostSimulator** beinhaltet den Quellcode der Hostanwendung und den des Simulators.
- **Android** beinhaltet den Quellcode der Androidanwendung.

Literaturverzeichnis

[Aagedal 2001] AAGEDAL, Jan O.: *Quality of Service Support in Development of Distributed Systems*, University of Oslo, Dissertation, 2001

[Abbas und Kure 2010] ABBAS, Ash M. ; KURE, Oivind: Quality of Service in mobile ad hoc networks, a survey. In: *Int. J. Ad Hoc Ubiquitous Comput.* 6 (2010), July, S. 75–98. – URL http://dx.doi.org/10.1504/IJAHUC.2010.034322. – ISSN 1743-8225

[Alliance 2011a] ALLIANCE, Open H.: *Android Developer Portal*. Website. 2011. – URL http://developer.android.com/index.html

[Alliance 2011b] ALLIANCE, OSGi: *OSGi Service Platform Release 4 Specification*. R4. http://www.osgi.org: , 2011

[Amundsen u. a. 2004] AMUNDSEN, Sten ; LUND, Ketil ; ELIASSEN, Frank ; STAEHLI, Richard: QuA: Platform-Managed QoS for Component Architectures. In: *In Proceedings from Norwegian Informatics Conference (NIK*, 2004, S. 55–66

[Andresen 2004] ANDRESEN, Andreas: *Komponentenbasierte Software-Entwicklung mit MDA, UML 2 und XML*. 2., neu bearbeitete Auflage. Carl Hanser Verlag München Wien, 2004. – URL http://www.andreasandresen.de

[Atkinson u. a. 2003] ATKINSON ; BÄR ; BAYER ; BUNSE ; GIRARD ; GROSS ; KETTEMANN ; KOLB ; KÜHNE ; ROMBERG ; SENG ; SORY ; TOLZMANN: *Handbuch zur komponentenbasierten Softwareentwicklung*. Fraunhofer - Institut Experimentelles Software Engeneering, 2003

[Backhaus u. a. 2008] BACKHAUS, Klaus ; ERICHSON, Bernd ; PLINKE, Wulff ; WEIBER, Rolf: *Multivariate Analysemethoden: Eine anwendungsorientierte Einführung (Springer-Lehrbuch) (German Edition)*. 12., vollständig überarb. Aufl. Springer, 2008

[Balzert 1997] BALZERT, Helmut: *Lehrbuch der Software-Technik, Bd. 2: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung, inkl. 1 CD-ROM (Springers Lehrbucher Der Informatik) (German Edition)*. Spektrum Akademischer Verlag, nov 1997. – ISBN 3827400651

[Becker u. a. 2010] BECKER, Arno ; PANT, Marcus ; MÜLLER, David: *Android 2 - Grundlagen und Programmierung*. D-Punkt, 2010

[Boehm 1978] BOEHM, Barry: *Characteristics of software quality Vol 1*. Amsterdam, Holland : North-Holland, 1978 (TRW series on software technology)

[Brahnmath u. a. 2002] BRAHNMATH ; RAJE ; OLSON ; BRYANT ; AUGUSTON ; BURT: A quality of service catalog for software components, 2002 (Proc. Southeastern Software Engineering Conference)

[Brun u. a. 2009] BRUN, Yuriy ; SERUGENDO, Giovanna Di M. ; GACEK, Cristina ; GIESE, Holger M. ; KIENLE, Holger ; LITOIU, Marin ; MÜLLER, Hausi A. ; PEZZÈ, Mauro ; SHAW, Mary: Engineering Self-Adaptive Systems through Feedback Loops. In: CHENG, Betty H. C. (Hrsg.) ; LEMOS, Rogério de (Hrsg.) ; GIESE, Holger (Hrsg.) ; INVERARDI, Paola (Hrsg.) ; MAGEE, Jeff (Hrsg.): *Software Engineering for Self-Adaptive Systems*, Springer, 2009 (Lecture Notes in Computer Science), S. 48–70. – ISBN 978-3-642-02160-2

[Capra u. a. 2003] CAPRA, Licia ; EMMERICH, Wolfgang ; MASCOLO, Cecilia: CARISMA: Context-Aware Reflective Middleware System for Mobile Applications. In: *IEEE Trans. Softw. Eng.* 29 (2003), October, S. 929–945. – URL `http://portal.acm.org/citation.cfm?id=1435630.1436363`. – ISSN 0098-5589

[Cheng u. a. 2009] CHENG, Betty H. C. ; LEMOS, Rogério de ; GIESE, Holger ; INVERARDI, Paola ; MAGEE, Jeff ; ANDERSSON, Jesper ; BECKER, Basil ; BENCOMO, Nelly ; BRUN, Yuriy ; CUKIC, Bojan ; SERUGENDO, Giovanna Di M. ; DUSTDAR, Schahram ; FINKELSTEIN, Anthony ; GACEK, Cristina ; GEIHS, Kurt ; GRASSI, Vincenzo ; KARSAI, Gabor ; KIENLE, Holger M. ; KRAMER, Jeff ; LITOIU, Marin ; MALEK, Sam ; MIRANDOLA, Raffaela ; MÜLLER, Hausi A. ; PARK, Sooyong ; SHAW, Mary ; TICHY, Matthias ; TIVOLI, Massimo ; WEYNS, Danny ; WHITTLE, Jon: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: CHENG, Betty H. C. (Hrsg.) ; LEMOS, Rogério de (Hrsg.) ; GIESE, Holger (Hrsg.) ; INVERARDI, Paola (Hrsg.) ; MAGEE, Jeff (Hrsg.): *Software Engineering for Self-Adaptive Systems* Bd. 5525, Springer, 2009, S. 1–26. – ISBN 978-3-642-02160-2

[wen Cheng u. a. 2004] CHENG, Shang wen ; HUANG, An cheng ; GARLAN, David ; SCHMERL, Bradley ; STEENKISTE, Peter: Rainbow: Architecture-based self-adaptation with reusable infrastructure. In: *IEEE Computer* 37 (2004), S. 46–54

[Dobson u. a. 2006] DOBSON, Simon ; DENAZIS, Spyros ; FERNANDEZ, Antonio ; GAITI, Dominique ; GELENBE, Erol ; MASSACCI, Fabio ; NIXON, Paddy ; SAFFRE, Fabrice ; SCHMIDT, Nikita ; ZAMBONELLI, Franco: A survey of autonomic communications. In: *ACM Trans. Auton. Adapt. Syst.* 1 (2006), December, S. 223–259. – URL `http://doi.acm.org/10.1145/1186778.1186782`. – ISSN 1556-4665

[Fahrmeir und Hamerle 1984] FAHRMEIR, Ludwig (Hrsg.) ; HAMERLE, Alfred (Hrsg.): *Multivariate statistische Verfahren*. Berlin : de Gruyter, 1984

[Frolund und Koistinen 1998] FROLUND, Svend ; KOISTINEN, Jari: Quality of services specification in distributed object systems design. In: *Proceedings of the 4th conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 4*. Berkeley, CA, USA : USENIX Association, 1998 (COOTS 98)

[Ganek und Corbi 2003] GANEK, A.G. ; CORBI, T.A.: The Dawning of the Autonomic Computing Era. In: *IBM Systems Journal* 42 (2003), Nr. 1, S. 5–18

[Gebauer u. a. 2007] GEBAUER, Judith ; TANG, Ya ; BAIMAI, Chaiwat: User Requirements of Mobile Technology–Results from a Content Analysis of User Reviews / University of Illinois at Urbana-Champaign, College of Business. Research Papers in Economics, Oktober 2007 (07-0107). – Working Papers. – URL `http://ideas.repec.org/p/ecl/illbus/07-0107.html`

[Geihs 2007] GEIHS, Kurt: Selbst-adaptive Software. In: *InformatikSpektrum* 31 (2007), Nr. 2, S. 133–145

[Giner u. a. 2009] GINER, Pau ; CETINA, Carlos ; FONS, Joan ; PELECHANO, Vicente: Building Self-adaptive Services for Ambient Assisted Living. In: *Proceedings of the 10th International Work-Conference on Artificial Neural Networks: Part II: Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living*. Berlin, Heidelberg : Springer-Verlag, 2009 (IWANN '09), S. 740–747. – ISBN 978-3-642-02480-1

[Gopalakrishna 2004] GOPALAKRISHNA, Praveen: *Modelling QoS Parameters in Component-Based Systems*, Purdue University, Diplomarbeit, 2004

[Gouda und Herman 1991] GOUDA, Mohamed G. ; HERMAN, Ted: Adaptive Programming. In: *IEEE Trans. Softw. Eng.* 17 (1991), September, S. 911–921. – URL `http://portal.acm.org/citation.cfm?id=126262.126271`. – ISSN 0098-5589

[Grace u. a. 2003] GRACE, Paul ; BLAIR, Gordon S. ; SAMUEL, Sam: ReMMoC: A Reflective Middleware to Support Mobile Client Interoperability. In: *Interoperability, Proc. International Symposium of Distributed Objects and Applications (DOA03*, 2003

[Grassi u. a. 2007] GRASSI, Vincenzo ; MIRANDOLA, Raffaela ; SABETTA, Antonino: A Model-Driven Approach to Performability Analysis of Dynamically Reconfigurable Component-Based Systems. In: *WSOP' 07*, ACM, 2007, S. 0

[Gross u. a. 1999] GROSS, Thomas ; STEENKISTE, Peter ; SUBHLOK, Jaspal: Adaptive Distributed Applications on Heterogeneous Networks. In: *Proceedings of the Eighth*

Heterogeneous Computing Workshop. Washington, DC, USA : IEEE Computer Society, 1999 (HCW '99), S. 209–. – URL http://portal.acm.org/citation.cfm?id=795690.797897. – ISBN 0-7695-0107-9

[Gruhn und Thiel 2000] GRUHN, Volker ; THIEL, Andreas: *Komponentenmodelle. DCOM, Javabeans, Enterprise Java Beans, CORBA*. Addison-Wesley, 2000

[Heineman und Councill 2001] HEINEMAN, George T. (Hrsg.) ; COUNCILL, William T. (Hrsg.): *Component-based software engineering: putting the pieces together*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2001. – ISBN 0-201-70485-4

[Huebscher und McCann 2008] HUEBSCHER, Markus C. ; MCCANN, Julie A.: A survey of autonomic computing - degrees, models and applications. In: *ACM Comput. Surv.* 40 (2008), August, S. 7:1–7:28. – ISSN 0360-0300

[Hug 2001] HUG, Karlheiz: *Module, Klassen, Verträge : ein Lehrbuch zur komponentenbasierten Softwarekonstruktion mit Component Pascal*. Vieweg, 2001

[IBM-Research 2011] IBM-RESEARCH: *The 8 Elements*. Website. 2011

[ISO/IEC 2001] ISO/IEC: *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001

[Juran 1988] JURAN, J. M.: *Juran's Quality Control Handbook*. 4th. Mcgraw-Hill (Tx), 1988

[Kan 1994] KAN, Stephen H.: *Metrics and Models in Software Quality Engineering*. 1st. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1994. – ISBN 0201633396

[Kephart und Chess 2003a] KEPHART, Jeffrey O. ; CHESS, David M.: The Vision of Autonomic Computing. In: *Computer* 36 (2003), January, S. 41–50. – URL http://dx.doi.org/10.1109/MC.2003.1160055. – ISSN 0018-9162

[Kephart und Chess 2003b] KEPHART, Jeffrey O. ; CHESS, David M.: The Vision of Autonomic Computing. In: *Computer* 36 (2003), January, S. 41–50. – URL http://dx.doi.org/10.1109/MC.2003.1160055. – ISSN 0018-9162

[Kipling 1902] KIPLING, Rudyard: *Just So Stories for Little Children*.

[Kokar u. a. 1999] KOKAR, Mieczyslaw M. ; BACLAWSKI, Kenneth ; ERACAR, Yonet A.: Control Theory-Based Foundations of Self-Controlling Software. In: *IEEE Intelligent Systems* 14 (1999), May, S. 37–45. – URL http://dx.doi.org/10.1109/5254.769883. – ISSN 1541-1672

[Kramer und Magee 2007] KRAMER, Jeff ; MAGEE, Jeff: Self-Managed Systems: an Architectural Challenge. In: *2007 Future of Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2007 (FOSE '07), S. 259–268. – URL http://dx.doi.org/10.1109/FOSE.2007.19. – ISBN 0-7695-2829-5

[Kurose und Ross 2008] KUROSE, James F. ; ROSS, Keith W.: *Computernetzwerke - der Top-Down-Ansatz (4. Aufl.)*. Pearson Studium, 2008. – 1–896 S. – ISBN 978-3-8273-7330-4

[Laddaga 1999] LADDAGA, Robert: Guest Editor's Introduction: Creating Robust Software through Self-Adaptation. In: *IEEE Intelligent Systems* 14 (1999), May, S. 26–29. – URL http://dx.doi.org/10.1109/MIS.1999.769879. – ISSN 1541-1672

[Laddaga 2000] LADDAGA, Robert: Active software. In: *Proceedings of the first international workshop on Self-adaptive software*. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2000 (IWSAS' 2000), S. 11–26. – URL http://portal.acm.org/citation.cfm?id=375094.375105. – ISBN 3-540-41655-2

[Li u. a. 2000] LI, Baochun ; JEON, Won ; KALTER, William ; NAHRSTEDT, Klara ; SEO, Jun hyuk: Adaptive Middleware Architecture for a Distributed Omni-Directional Visual Tracking System. In: *Proceedings of SPIE/ACM MMCN 2000*, 2000, S. 101–112

[McCall 1977] MCCALL: Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality. / GENERAL ELECTRIC CO SUNNYVALE CALIF. 1977. – Technical Report

[McConnell 2004] MCCONNELL, Steve: *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2004. – ISBN 0735619670

[McIlroy 1968] MCILROY, M. D.: Mass-produced software components. In: *Proc. NATO Conf. on Software Engineering, Garmisch, Germany* (1968)

[McKinley u. a. 2004] MCKINLEY, Philip. K. ; SADJADI, Seyed M. ; KASTEN, Eric P. ; CHENG, Betty H. C.: A Taxonomy of Compositional Adaptation / Michigan State University. 2004. – Forschungsbericht

[Nahrstedt u. a. 2001] NAHRSTEDT, Klara ; XU, Dongyan ; WICHADAKUL, Duangdao ; LI, Baochun: QoS-Aware Middleware for Ubiquitous and Heterogeneous Environments. In: *IEEE Communications Magazine* 39 (2001), S. 140–148

[Neema und Ledeczi 2001] NEEMA, Sandeep ; LEDECZI, Akos: Constraint-guided self-adaptation. In: *IN: IWSAS* (2001), S. 39–51

[Norvig 1998] NORVIG, Peter: *Decision Theory: The Language of Adaptive Agent Software*. Internet. 1998

[Opfer 2002] OPFER, Gerhard: *Numerische Mathematik für Anfänger, 4. Auflage*. Vieweg Friedr. + Sohn Verlag, 2002

[OSGi-Alliance 2011] OSGI-ALLIANCE: *Website*. Website. 2011. – URL `http://www.osgi.org`

[Parashar und Hariri 2005] PARASHAR, Manish ; HARIRI, Salim: Autonomic computing: An overview. In: *Unconventional Programming Paradigms*, Springer Verlag, 2005, S. 247–259

[Perry 1987] PERRY, William: *Effective methods for EDI quality assurance*. New Jersey,USA : Prentice-Hall, 1987 (TRW series on software technology)

[Pressman 2005] PRESSMAN, Scott: *Software engineering: a practitioner's approach (6th ed.)*. New York, NY, USA : McGraw-Hill Education, Inc., 2005

[Robertson und Williams 2006] ROBERTSON, Paul ; WILLIAMS, Brian: Automatic recovery from software failure. In: *Commun. ACM* 49 (2006), March, S. 41–47. – URL `http://doi.acm.org/10.1145/1118178.1118200`. – ISSN 0001-0782

[Rouvoy u. a. 2008] ROUVOY, Romain ; ELIASSEN, Frank ; FLOCH, Jacqueline ; HALLSTEINSEN, Svein O. ; STAV, Erlend: Composing Components and Services Using a Planning-Based Adaptation Middleware. In: PAUTASSO, Cesare (Hrsg.) ; TANTER Éric (Hrsg.): *Software Composition*, Springer, 2008 (Lecture Notes in Computer Science), S. 52–67

[Salehie und Tahvildari 2009] SALEHIE, Mazeiar ; TAHVILDARI, Ladan: Self-adaptive software: Landscape and research challenges. In: *ACM Trans. Auton. Adapt. Syst.* 4 (2009), May, S. 14:1–14:42. – URL `http://doi.acm.org/10.1145/1516533.1516538`. – ISSN 1556-4665

[Sastry und Bodson 1989] SASTRY, Shankar ; BODSON, Marc: *Adaptive Control: Stability, Convergence, and Robustness*. Prentice-Hall, 1989. – ISBN 0-13-004326-5

[Schilit u. a. 1994] SCHILIT, Bill N. ; ADAMS, Norman ; WANT, Roy: Context-Aware Computing Applications. In: *IN PROCEEDINGS OF THE WORKSHOP ON MOBILE COMPUTING SYSTEMS AND APPLICATIONS*, IEEE Computer Society, 1994, S. 85–90

[Siedersleben 2004] SIEDERSLEBEN, Johannes (Hrsg.): *Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar*. Heidelberg : dpunkt, 2004

[Streichert 2007] STREICHERT, Thilo: *Self-Adaptive Hardware/Software Reconfigurable Networks - Concepts, Methods, and Implementation*, Universität Erlangen, Dissertation, 2007

[Szyperski 2002] SZYPERSKI, Clemens: *Component Software: Beyond Object-Oriented Programming*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002. – ISBN 0201745720

[Tesauro u. a. 2004] TESAURO, Gerald ; CHESS, David M. ; WALSH, William E. ; DAS, Rajarshi ; SEGAL, Alla ; WHALLEY, Ian ; KEPHART, Jeffrey O. ; WHITE, Steve R.: A Multi-Agent Systems Approach to Autonomic Computing. In: *Autonomous Agents and Multiagent Systems, International Joint Conference on* 1 (2004), S. 464–471. ISBN 0-7695-2092-8

[Truyen und Joosen 2008] TRUYEN, Eddy ; JOOSEN, Wouter: *Towards an Aspect-Oriented Architecture for Self-Adaptive Frameworks*. 2008

[Ullenboom 2009] ULLENBOOM, Christian: *Java ist auch eine Insel*. Eighth. Galileo Computing, 2009. – URL `http://openbook.galileocomputing.de/javainsel8/`. – ISBN 978-3-8362-1371-4

[Van Lamsweerde 2001] VAN LAMSWEERDE, Axel: Goal-Oriented Requirements Engineering: A Guided Tour. In: *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*. Washington, DC, USA : IEEE Computer Society, 2001 (RE '01), S. 249–. – URL `http://portal.acm.org/citation.cfm?id=882477.883624`

[Weinberg 1992] WEINBERG, Gerald M.: *Quality software management (Vol. 1): systems thinking*. New York, NY, USA : Dorset House Publishing Co., Inc., 1992. – ISBN 0-932633-22-6

[Wütherich u. a. 2008] WÜTHERICH, Gerd ; HARTMANN, Nils ; KOLB, Bernd ; LÜBKEN, Matthias ; DPUNKT.VERLAG (Hrsg.): *Die OSGi Service Platform: Eine Einführung mit Eclipse Equinox*. 1. Heidelberg, Germany : dpunkt, 2008

[Wu u. a. 2010] WU, Yuankai ; WU, Yijian ; PENG, Xin ; ZHAO, Wenyun: Implementing Self-Adaptive Software Architecture by Reflective Component Model and Dynamic AOP: A Case Study. In: *Proceedings of the 2010 10th International Conference on Quality Software*. Washington, DC, USA : IEEE Computer Society, 2010 (QSIC '10), S. 288–293. – URL `http://dx.doi.org/10.1109/QSIC.2010.56`. – ISBN 978-0-7695-4131-0

[Zhou u. a. 2005] ZHOU, Jia ; COOPER, Kendra ; YEN, I ling: Rule-Base Technique for Component Adaptation to Support QoSbased Reconfiguration. In: *Proceedings of ISORC*, 2005, S. 426–433

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 28. Juni 2011	
Ort, Datum	Unterschrift