Modellierung des Komponentenzustandes von JavaScript-basierten Mashup-Komponenten

Großer Beleg
Technische Universität Dresden
Januar 2013

Christoph Pohl

Betreuer: Dipl.-Medieninf. Oliver Mroß
Hochschullehrer: Prof. Dr.-Ing. Klaus Meißner

Fakultät Informatik
Institut für Software- und Multimediatechnik
Seniorprofessur für Multimediatechnik

MMT
Erklärung


Dresden, 15. Januar 2013

Unterschrift
 Aufgabenstellung

Diese Seite muss vor dem Binden der gedruckten Fassung der Arbeit durch die von Herrn Meißner und dem Studenten eigenhändig unterschriebene originale Aufgabenstellung ersetzt werden. Das zweite abzugebende gebundene Exemplar soll stattdessen eine Kopie dieser originalen Aufgabenstellung enthalten.
# Inhaltsverzeichnis

## 1 Einleitung
1.1 Motivation ........................................... 1
1.2 Zielstellung ........................................... 2
1.3 Aufbau der Arbeit ..................................... 2

## 2 Grundlagen
2.1 Komposite Webanwendungen ............................. 4
2.2 UI-Migration ........................................... 5
   2.2.1 Begriffsdefinition und -abgrenzung ................. 5
   2.2.2 Teilkonzepte der UI-Migration ...................... 7
   2.2.3 Durchführung der UI-Migration im Überblick ....... 9
   2.2.4 Zusammenfassung .................................. 11
2.3 Softwarekomponenten .................................. 12
   2.3.1 Anforderungen an Softwarekomponenten ............ 12
   2.3.2 Komponentenmodelle ................................ 14
   2.3.3 Das Komponentenmodell von CRUISe ................. 15
2.4 CRUISe-Architektur im Überblick ...................... 16
   2.4.1 Das Komponentenrepositorium ...................... 18
   2.4.2 Die Laufzeitumgebung ............................. 18
   2.4.3 Defizite von CRUISe ............................. 19
2.5 Zusammenfassung ........................................ 20

## 3 Stand der Forschung und Technik
3.1 Austausch von Komponenten in CRUISe ................ 21
   3.1.1 Sicherstellung der Isolation ....................... 22
   3.1.2 Ablauf des Komponentenaustausches ............... 23
   3.1.3 Transfer des Komponentenzustandes ................. 25
   3.1.4 Zusammenfassung .................................... 26
3.2 Atomarität und Zustandserhaltung in CoBRA ............ 27
   3.2.1 Atomarität mit Hilfe des Protection-Proxy-Patterns .... 28
   3.2.2 Zustandserhaltung mit Hilfe des Memento-Patterns .... 29
   3.2.3 Zusammenfassung .................................... 31
3.3 Die OPEN Migration Service Platform .................. 31
   3.3.1 Architektur der OPEN-Plattform im Überblick ........ 32
   3.3.2 Migration von Komponenten in OPEN ................ 34
   3.3.3 Herausforderungen bei der Serialisierung von JS-Variablen .. 36
   3.3.4 Zusammenfassung .................................... 39
3.4 COSMOD: Generierung von Zustandsgraphen ............. 40
3.5 Registrierung auftretender (DOM-Baum-)Events .......... 41
3.6 Zusammenfassung ........................................ 42

Copyright TU Dresden, Christoph Pohl
4 Konzeption der UI-Migration .......................... 44
  4.1 Anforderungsanalyse ................................ 44
    4.1.1 Funktionale Anforderungen ........................ 44
    4.1.2 Nicht-funktionale Anforderungen .................... 45
  4.2 Evaluation der untersuchten Ansätze ..................... 46
    4.2.1 Kriterienkatalog .................................. 46
    4.2.2 Bewertung der untersuchten Konzepte ................ 47
    4.2.3 Zusammenfassung und Schlussfolgerung ............... 55
  4.3 Überblick des Gesamtkonzeptes ........................ 57
    4.3.1 Anwendungsszenario ............................... 57
    4.3.2 Abgrenzung der Verantwortlichkeiten ................. 58
    4.3.3 Übersicht aller Teilkonzepte der UI-Migration ....... 59
    4.3.4 Vorbetrachtungen zur UI-Migration .................. 61
    4.3.5 Ausgangslage für den weiteren Verlauf der Konzeption . 66
  4.4 Aufbau des verwendeten Zustandsmodells .................. 67
    4.4.1 Kategorisierung von Zustandsinformationen .......... 67
  4.5 Aufgabenverteilung der beteiligten Komponenten .......... 69
  4.6 Extraktion der Zustandsinformationen ................... 70
    4.6.1 Wahl einer Zustandsextraktionsmethode ............... 70
    4.6.2 Mutation-Event-Detection mit Hilfe des State Handlers 71
  4.7 Transfer der Zustandsdaten ............................. 74
    4.7.1 Einleitung der Transferphase ........................ 74
    4.7.2 Ausgangsgerät → Migration Server → Zielgerät ....... 75
  4.8 Wiederherstellung des UI-Komponentenzustandes .......... 76
    4.8.1 Die verfügbaren Zustandsinformationen ............... 77
    4.8.2 Ablauf der Zustandsinjektion ........................ 77
  4.9 Zusammenfassung ........................................ 79

5 Implementation .......................................... 81
  5.1 Das verwendete Implementierungsframework ............... 81
  5.2 Erweiterung der CRUISe-Thin-Server-Runtime .............. 81
    5.2.1 Anpassungen innerhalb des Application Managers ....... 82
  5.3 Senderlogik: Zustandserfassung und -extraktion .......... 84
    5.3.1 DOM-Mutation-Observer in JavaScript ................ 85
  5.4 Empfängerlogik: Zustandswiederherstellung ............... 87
    5.4.1 Erweiterung der Komponentenschnittstelle .......... 87
  5.5 Beispielanwendung ....................................... 88
  5.6 Zusammenfassung ......................................... 90

6 Zusammenfassung ......................................... 91
  6.1 Ergebnisse ............................................. 91
  6.2 Ausblick ................................................ 92

A Anhang .................................................. i

Literaturverzeichnis ......................................... vi
## Abbildungsverzeichnis

<table>
<thead>
<tr>
<th>Nummer</th>
<th>Titel</th>
<th>Seitennummer</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Eine Anwendung auf drei heterogenen Gerätetypen [Res12]</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Mögliche Architektur für eine UI-Migration</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Möglicher Ablauf einer UI-Migration</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Mögliche schematische Darstellung einer Softwarekomponente</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Komposition - direkte Kopplung</td>
<td>13</td>
</tr>
<tr>
<td>2.6</td>
<td>Komposition - lose Kopplung</td>
<td>14</td>
</tr>
<tr>
<td>2.7</td>
<td>CRUISe - Komponentenmodell</td>
<td>16</td>
</tr>
<tr>
<td>2.8</td>
<td>CRUISe - Architekturübersicht [Pie09]</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Protection proxy pattern in CoBRA [IFM08]</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Adaptsprozess in CoBRA [IFM08]</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Struktur des Memento-Patterns [Gam+95]</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>Sequenzdiagramm des Memento-Patterns [Gam+95]</td>
<td>30</td>
</tr>
<tr>
<td>3.5</td>
<td>OPEN - Architektur [Pat11]</td>
<td>33</td>
</tr>
<tr>
<td>3.6</td>
<td>OPEN - Migration eines PC-Spiels [Pat11]</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>Grobkonzept der UI-Migration</td>
<td>58</td>
</tr>
<tr>
<td>4.2</td>
<td>Detailansicht der beteiligten Komponenten</td>
<td>59</td>
</tr>
<tr>
<td>4.3</td>
<td>Informationen über die zu übertragenden Zustandsinformationen je Anwendungsszenario</td>
<td>68</td>
</tr>
<tr>
<td>4.4</td>
<td>Zusammenhänge der beteiligten CRUISe-Komponenten</td>
<td>69</td>
</tr>
<tr>
<td>5.1</td>
<td>Erweiterung der CRUISe- Thin-Server-Runtime</td>
<td>82</td>
</tr>
<tr>
<td>5.2</td>
<td>Klasseendiagramm der JavaScript-basierten Basisklasse</td>
<td>84</td>
</tr>
<tr>
<td>5.3</td>
<td>Screenshot der Beispielanwendung</td>
<td>89</td>
</tr>
<tr>
<td>A.1</td>
<td>Evaluation der untersuchten Forschungsprojekte</td>
<td>i</td>
</tr>
<tr>
<td>A.2</td>
<td>Grundlegende Prozessschritte der UI-Migration</td>
<td>ii</td>
</tr>
<tr>
<td>A.3</td>
<td>Prozessschritte der konzeptionierten Zustandserfassung</td>
<td>iii</td>
</tr>
<tr>
<td>A.4</td>
<td>Aufbau des Kommunikationsprotokolls</td>
<td>iv</td>
</tr>
<tr>
<td>A.5</td>
<td>Prozessschritte der konzeptionierten Zustandsinjektion</td>
<td>v</td>
</tr>
</tbody>
</table>
# Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Semantische Typisierung</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Objektreferenzen</td>
<td>37</td>
</tr>
<tr>
<td>3.3</td>
<td>Zykliche Referenzen</td>
<td>37</td>
</tr>
<tr>
<td>3.4</td>
<td>Timer</td>
<td>38</td>
</tr>
<tr>
<td>3.5</td>
<td>Dynamische Wertzuweisung</td>
<td>39</td>
</tr>
<tr>
<td>3.6</td>
<td>Referenzen auf DOM-Knoten</td>
<td>39</td>
</tr>
<tr>
<td>5.1</td>
<td>Erweiterung des Konstruktors von dem Application Manager</td>
<td>82</td>
</tr>
<tr>
<td>5.2</td>
<td>Instantiierung der notwendigen Migrations-Komponenten</td>
<td>83</td>
</tr>
<tr>
<td>5.3</td>
<td>Registrierung eines System-Kanals</td>
<td>83</td>
</tr>
<tr>
<td>5.4</td>
<td>Hinzufügen eines Events zum Kanal</td>
<td>83</td>
</tr>
<tr>
<td>5.5</td>
<td>Registrierung eines Event-Handlers</td>
<td>84</td>
</tr>
<tr>
<td>5.6</td>
<td>Instantiierung eines DOM-Mutation-Observers in JavaScript</td>
<td>85</td>
</tr>
<tr>
<td>5.7</td>
<td>Kategorisierung der signalisierten Zustandsveränderungen</td>
<td>86</td>
</tr>
<tr>
<td>5.8</td>
<td>Bekanntmachung von Zustandsveränderungen</td>
<td>86</td>
</tr>
<tr>
<td>5.9</td>
<td>Funktions zur Injektion des Komponentenzustandes</td>
<td>87</td>
</tr>
</tbody>
</table>
1 Einleitung

1.1 Motivation

dient nun als Steuerungskomponente und ermöglicht es zwischen den einzelnen Folien hin und herzublättern. Zusätzlich zum Tablet ermöglicht es Christoph seinem Kollegen, welcher auf ein markantes Detail im Video aufmerksam machen möchte, die Präsentation mit seinem Smartphone fernzusteuern. Beide Personen können demzufolge unabhängig voneinander die auf dem Smart-TV angezeigte Präsentation fernsteuern.


### 1.2 Zielstellung


### 1.3 Aufbau der Arbeit

Nachdem in diesem Kapitel bereits die Motivation und Zielstellung der Arbeit festgehalten wurden, sollen in Kapitel 2 wichtige Grundlagen erklärt werden. Neben der Erläuterung von Begriffen wie **UI-Migration** und **Softwarekomponente**, werden unter anderem Komponentenmodelle und -zustände sowie deren Schnittstellen und deren


Kapitel 5 widmet sich der prototypischen Umsetzung des zuvor entwickelten Konzeptes und geht dabei auf entscheidende Implementierungsdetails ein. In Anlehnung an das zuvor spezifizierte Anwendungsszenario, soll in diesem Zusammenhang eine Beispielanwendung entwickelt werden, welche die Funktionalität des Konzeptes verifiziert.

Abschließend fasst Kapitel 6 die Erkenntnisse und Ergebnisse der vorliegenden Arbeit zusammen, beschreibt eventuell offen gebliebenen Fragestellungen und gibt einen Ausblick auf weiterführende Themen bezüglich der UI-Migration im Umfeld des CRUISe-Projektes.
2 Grundlagen


2.1 Komposite Webanwendungen

Eine Anwendung, welche aus prinzipiell unabhängigen, wiederverwendbaren Einzelbestandteilen (Komponenten) zusammengesetzt ist und in einem Webbrowser läuft, wird als komposite Webanwendung bezeichnet \[\text{Rad11}\]. Dabei werden vorhandene Daten, Anwendungslogik und gegebenenfalls Bestandteile der Benutzerschnittstelle (User-Interface-Komponenten, siehe Abschnitt 2.2), welche über plattform- und sprachunabhängige Schnittstellen von Diensten bereitgestellt werden, kombiniert und integriert.


Ein Modell für die Konstruktion von Webanwendungen sind Mashups. Ein Mashup erbringt durch die Integration und Kombination vorhandener Funktionalitäten sowie Daten verschiedener Quellen einen Mehrwert und wird als hybride, situative Webanwendung angesehen \[\text{Mer06}\]. Feeds, Web Services und andere heterogene Datenquellen werden unter Verwendung von öffentlichen Schnittstellen abgefragt und client- oder serverseitig komponiert. Ursprünglich dienten Mashups ausschließlich der Kombination von Anwendungslogik und Daten. Aktuelle Forschungsansätze hingegen übertragen dieses Prinzip zusätzlich auf die Präsentationsschicht \[\text{Dan+07}\]. Im Rahmen dieser Arbeit soll CRUISe im Fokus der Untersuchungen stehen. Dabei
wird speziell auf die Übertragung einzelner UI-Komponenten zwischen zwei Geräten eingegangen. Dieser Vorgang wird unter dem Begriff \textit{UI-Migration} zusammengefasst und im nachfolgenden Abschnitt näher untersucht.

2.2 UI-Migration


2.2.1 Begriffsdefinition und -abgrenzung


Abbildung 2.1: Eine Anwendung auf drei heterogenen Gerätetypen [Res12]

Bei einer Übertragung der Benutzerschnittstelle spielt der Zustand der UI eine wesentliche Rolle, denn das Ziel dieser Migrationsart ist es, die Benutzerschnittstelle
Modellierung des Komponentenzustandes 2 Grundlagen

inklusive aller eingegebenen Daten und bisher ausgeführten Operationen von einem Ausgangsgerät auf ein Zielgerät zu übertragen. Um die Notwendigkeit der Zustands-erhaltung bei der UI-Migration zu verdeutlichen, wird im Folgenden ein kurzes Anwendungsszenario beschrieben:


1. **Zustandserhaltung.** Nach einer erfolgreichen Migration muss der Nutzer das User Interface auf dem Zielgerät nicht neu starten oder Daten ein weiteres Mal eingeben, um den Programmzustand des Ausgangsgerätes zu erreichen. Aus diesem Grund muss der aktuelle Zustand der Software des Ausgangsgerätes zuerst serialisiert, danach extrahiert und anschließend an das Zielgerät übertragen werden, um ihn an dieser Stelle wieder zu injizieren (siehe Kapitel 2.2.2).

tens während einer Migration von einem Desktop-PC zu einem mobilen Gerät
Anwendung statt, da zuvor unterstützte Operationen der Desktop-Plattform
aufgrund der Leistungsbeschränkungen mobiler Endgeräte nicht unterstützt
werden. Dies resultiert in einer Verringerung der möglichen Aufgabenvielfalt.
Des Weiteren ist es in bestimmten Fällen – bei eingeschränkter Netzwerkkon-
nektivität beispielsweise – notwendig, den eigentlichen Inhalt zu modifizieren
in dem Teile entfernt, hinzugefügt oder verändert werden. Dies kann sich zum
Beispiel durch eine Zusammenfassung eines langen Textes äußern, oder durch
die Ersetzung eines Multimedia-Objekts durch eine textbasierte Beschreibung.

Nach dieser einführenden Begriffsdefinition und -abgrenzung folgt nun eine Beschrei-
bung wichtiger Teilkonzepte des UI-Migrationsprozesses.

2.2.2 Teilkonzepte der UI-Migration

Silvia Berti et al. unterscheiden in [SS06] zwischen verschiedenen Arten der UI-
Migration:

- Bei der **totalen Migration** wird das gesamte UI zwischen zwei Geräten mi-
griert. Dies ermöglicht dem Benutzer zwischen Geräten zu wechseln, welche die
notwendigen Spezifikationen für die Ausführung der Software aufweisen. Die
Laufzeitumgebung ist dabei verantwortlich einen unterbrechungsfreien Pro-
grammablauf zu gewährleisten und Interface-Adaptionen an die verschiedenen
Plattformen durchzuführen.

- Im Gegensatz dazu, werden bei der **partiellen Migration** (auch partial-
migration genannt) nur Teilbereiche der Benutzerschnittstelle des Ausgangs-
gerätes herausgelöst und auf das Zielgerät migriert. Der Rest bleibt dabei auf
dem Ausgangsgerät erhalten. In [BP04] findet diese Art der Migration be-
spielsweise Anwendung. Dabei wird zwischen den Bereichen Benutzerinterak-
tion und Informationspräsentation unterschieden, welche sich jeweils auf einem
Gerät befinden. Wie im Anwendungsszenario aus Kapitel 1.1 beschrieben, ist
es zum Beispiel möglich eine Präsentationskomponente von einem Tablet auf
ein Smart-TV zu migrieren, wobei nach der UI-Migration das Tablet aus-
schließlich zur Steuerung und der Smart-TV lediglich zur Darstellung genutzt
werden.

- Bei der **verteilenden Migration** ist die Benutzerschnittstelle nach dem Mi-
grationsprozess komplett auf zwei oder mehr Geräte aufgeteilt.

- Wenn es hingegen notwendig ist das Interface von verschiedenen Ausgangs-
geräten auf ein einzelnes Zielgerät zu übertragen, wird die aggregierende
Migration angewendet.

Aufgrund der Tatsache, dass es wichtig ist einzelne Bestandteile der Benutzerschnitt-
stelle aus einer Gesamtanwendung herauslösen zu können, wird im Kontext der Auf-
gabenstellung ausschließlich die partielle Migration im Fokus dieser Forschungsarbeit
stehen. Neben diesen verschiedenen Migrationsarten, werden in [SS06] weitere Di-
mensionen migrierbarer Benutzerschnittstellen definiert, welche im Folgenden näher
betrachtet werden.
Der Aktivierungstyp beschreibt, wie eine Migration ausgelöst wird. Hierbei wird zwischen „on demand“ und „automatic“ unterschieden. Bei der erstgenannten Variante entscheidet der Nutzer wann und wie migriert werden soll. Im Gegensatz dazu entscheidet bei der automatischen Migration das System zu welchem Zeitpunkt der Vorgang gestartet wird und wählt eigenständig das Zielgerät aus. Da die Aktivierungsart der Migration den prinzipiellen Ablauf des Migrationsprozesses nicht beeinflusst, wird sie an dieser Stelle nicht näher untersucht.

Bei den Migrationsmodalitäten wird zwischen Mono- und Transmodalität unterschieden. Wenn die Benutzerschnittstelle einer Web-Anwendung beispielsweise von einem graphischen User-Interface eines Desktop-PCs in ein Sprachinterface eines PDAs migriert wurde, so liegt eine Transmodalität vor. Hat sich hingegen die Eingabeform nach der Migration nicht geändert, dann war die Migration monomodal. Eine explizite Unterscheidung dieser Migrationsmodalitäten wird in dieser Arbeit nicht unternommen, da es für die Lösung der Aufgabenstellung nicht von Bedeutung ist.

Des Weiteren wird in [SS06] zwischen precomputed UIs und runtime generated UIs unterschieden. Hierbei wird festgelegt, ob die Benutzerschnittstellen für jede Art von Gerät vordefiniert sind (precomputed), oder während der Laufzeit durch Adaptionsmechanismen für migrierende UI-Elemente entsprechend der Zielgerätespezifikationen generiert werden (runtime generated). Im Kontext dieser Forschungsarbeit wird ausschließlich die erstgenannte Variante betrachtet.


Im Zusammenhang mit der eben beschriebenen Zustandserhaltung ist der Aspekt der Zustandsmodellierung ein weiteres wichtiges Konzept der UI-Migration. Dabei beschreibt das Zustandsmodell, in welcher (möglichst technologie- und plattformunabhängigen) Form und Struktur die extrahierten Informationen serialisiert werden sollen, um von heterogenen Anwendungen verarbeitet werden zu können. Dieses Szenario tritt beispielsweise dann ein, wenn eine Flash-Komponente, welche auf einem Desktop-PC ausgeführt wird, auf ein Smartphone migriert werden soll, welches keine Flash-Unterstützung bietet. In diesem Zusammenhang wird während der Migration die Flash-Komponente durch eine JavaScript-Komponente ausgetauscht und der Zustand der Ausgangskomponente auf die Zielkomponente übertragen. Dabei muss beachtet werden, dass die Zustandsinformationen sowohl von der Ausgangskomponente, als auch von der empfangenden Komponente so interpretiert werden können,

2.2.3 Durchführung der UI-Migration im Überblick


Nach dieser einführenden Beschreibung der Architektur, wird im folgenden Ab-
schnitt ein möglicher Ablauf der UI-Migration auf Grundlage der Abbildung 2.3 erläutert. Der Ausgangspunkt ist dabei eine vom Nutzer initierte (oder automatische) Anfrage eine UI-Komponente zwischen zwei Geräten zu migrieren.

Abbildung 2.3: Möglicher Ablauf einer UI-Migration


3. Wurden passende UI-Komponenten spezifiziert, bestimmt eine serverseitige Schätzfunktion die Last, welche die zu migrierende UI-Komponente auf dem

4. Nachdem der Server eine passende UI-Komponente gefunden hat, werden alle beteiligten Komponenten auf dem Ausgangsgerät pausiert, damit die Anwendung keinen undefinierten Zustand erreicht.


Die vorliegende Forschungsarbeit siedelt sich im letzten Drittel des soeben präsentierten UI-Migrationsprozesses an. Demzufolge stehen sowohl die Zustandsextraktion und -serialisierung im Ausgangsgerät sowie die Übertragung dieser Daten an den Server (Schritt 5), als auch die Zustandsinjektion im Zielgerät (Schritt 6) im Mittelpunkt der Untersuchungen.

2.2.4 Zusammenfassung

welche Veränderungen an der Architektur von CRUISe möglicherweise vorgenommen werden müssen, um den beschriebenen UI-Migrationsvorgang erfolgreich durchführen zu können. Aus diesem Grund befasst sich das folgende Grundlagenkapitel zunächst mit Softwarekomponenten und Komponentenmodellen im Allgemeinen und wird anschließend auf die Umsetzung dieser Konzepte in CRUISe näher eingehen.

2.3 Softwarekomponenten


Obwohl der Begriff der Softwarekomponente schon länger existiert, gibt es keine eindeutige Definition. Clemens Szyperski und Desmond Francis D’Souza definieren den Begriff wie folgt:

„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.“ [CD98]

„A software component is a coherent package of software implementations that has explicit and well-specified interfaces for services it provides; has explicit and well-specified interfaces for services it expects; and can be composed with other components, perhaps customizing some of their properties, without modifying the components themselves.“ [DSB99]

Szyperski und D’Souza beschreiben eine Softwarekomponente als kontextunabhängige Einheit mit wohldefinierten Schnittstellen für Services die sie bereitstellt sowie Services die sie erwartet (siehe Abbildung 2.4). Des Weiteren kann sie mit anderen Komponenten gekoppelt werden und bietet Konfigurationsmöglichkeiten für einige ihrer Eigenschaften ohne Änderungen der Komponente selbst. Aus diesen Definitionen kristallisieren sich Merkmale heraus, mit deren Hilfe im nächsten Abschnitt Softwarekomponenten genauer klassifiziert werden können.

![Abbildung 2.4: Mögliche schematische Darstellung einer Softwarekomponente](image)

2.3.1 Anforderungen an Softwarekomponenten

Die Anforderungen an Softwarekomponenten lassen sich in 3 Eigenschaften unterteilen, die in diesem Abschnitt näher betrachtet werden:
1. Wiederverwendbarkeit

2. Kompositionsfähigkeit

3. Migrierbarkeit

Um einen möglichst hohen Grad an Wiederverwendung zu erreichen ist es notwendig, dass Softwarekomponenten eine eindeutige Beschreibung ihrer Schnittstelle bereitstellen. Das bedeutet, dass die eigentlichen Funktionen bekannt gegeben werden sowie Möglichkeiten beschrieben werden, wie diese angesprochen werden können. Dabei wird die eigentliche Implementierung gekapselt und geheimgehalten (Black-Box-Paradigma). Je weniger Abhängigkeiten bestehen, desto einfacher ist die Wiederverwendung in anderen Systemen.

Bei der Komposition von Softwarekomponenten werden grundsätzlich zwei Formen der Kopplung unterschieden: Bei der direkten Kopplung werden zwei einzelne Komponenten so miteinander verbunden, dass dadurch eine neue entsteht [Böh09] (siehe Abbildung 2.5).

Abbildung 2.5: Komposition - direkte Kopplung


Abbildung 2.6: Komposition - lose Kopplung

2.3.2 Komponentenmodelle

Um eine komponentenbasierte Softwareentwicklung zu erleichtern, existieren verschiedene Komponentenmodelle, welche Anne Thomas in [Tho98] wie folgt definiert:
"A component model defines the basic architecture of a component, specifying the structure of its interfaces and the mechanisms by which it interacts with its container and with other components. The component model provides guidelines to create and implement components that can work together to form a larger application. Application builders can combine components from different developers or different vendors to construct an application." [Tho98]


2.3.3 Das Komponentenmodell von CRUISe

In dem Komponentenmodell von CRUISe wird eine Komponente als eine unabhängige Einheit einer Anwendung angesehen, welche Daten, Business-Logik und Benutzerschnittstellen enthalten kann [PPM10]. Prinzipiell werden verschiedene Typen unterschieden, welche nachfolgend kurz erläutert werden:

- **UI-Komponenten** sind für die graphische Darstellung von Inhalten verantwortlich und werden durch UI-Services bereitgestellt. Beispiele für diese Art von Komponenten sind Kartenanzeigen oder auch Bildergalerien.


- **Logik-Komponenten** dienen der Transformation, Manipulation und Aggregation von Daten zwischen einzelnen Komponenten und können zum Beispiel für die Berechnung der Geokoordinaten aus einer Postadresse genutzt werden. Diese Daten könnte dann ein Kartendienst weiterverarbeiten.

- **Service-Komponenten** besitzen keine Benutzerschnittstelle und stellen einen Zugriff auf entfernte Ressourcen, wie Backend-Dienste, zur Verfügung.


Abbildung 2.7: CRUISe - Komponentenmodell

Dieser Abschnitt hat damit begonnen den Begriff Softwarekomponente allgemein zu definieren und ist anschließend auf die Anforderungen an diese eingegangen. Im weiteren Verlauf wurde der Begriff des Komponentenmodells beschrieben und die Umsetzung dieses Konzeptes in CRUISe näher betrachtet. Der nachfolgende Abschnitt gibt einen Überblick über die Architektur von CRUISe und geht dabei unter anderem auf die Laufzeitumgebung sowie die bestehenden Defizite der Architektur hinsichtlich der UI-Migration ein.

2.4 CRUISe-Architektur im Überblick

Das Forschungsprojekt CRUISe beschäftigt sich mit der modellgetriebenen Entwicklung von kompositen, kontextsensitiven Webanwendungen. Ziel dabei ist die Übertragung des SOA1-Paradigmas auf die Komponenten der Benutzeroberfläche. Dabei soll die gesamte Anwendung aus wiederverwendbaren und konfigurierbaren Komponenten bestehen, welche durch User Interface Services (UIS) als Dienst zur Verfügung gestellt werden [PPM10]. Der folgende Abschnitt wird auf Grundlage der Abbildung 2.8 den Aufbau sowie die Arbeitsweise der CRUISe-Architektur beschreiben [Sch11].

Mit Hilfe eines Autorenwerkzeuges modelliert der Nutzer zunächst die Webanwendung und erstellt damit das Kompositionsmodell (Composition Model), in welchem die notwendigen funktionalen Eigenschaften und Konfigurationsparameter der UI-Komponenten sowie ihre Beziehungen zueinander enthalten sind. Das Kompositionsmodell setzt sich aus mehreren weiteren Modellen zusammen, welche im Folgenden kurz beschrieben werden:

---

1 Serviceorientierte Architektur
- Das **Conceptual Model** besteht aus vier Containern für Datentypen, Informationen zur äußeren Schnittstelle zur Umgebung, visuelle Stile (entkoppeln visuelle Eigenschaften und macht sie wiederverwendbar) und Komponenten (alle funktionellen Bausteine einer Anwendung).


- Das **Layoutmodel** beschreibt die visuelle Anordnung der UI-Komponenten auf der Oberfläche und kann mit Instanzen vordefinierter Layout-Klassen, aus Frameworks wie Java Swing oder SWT, gefüllt werden, um beispielsweise eine absolute, pixelgenaue Positionierung zu ermöglichen.

- Im **Screenflow Model** wird die Navigation innerhalb einer kompositen Anwendung modelliert und dadurch können verschiedene Sichten auf die Anwendung und ihre Daten, abhängig von dem Anwendungszustand oder dem Nutzer-/Endgerätekontext, definiert werden.

![Diagram of CRUISe Architekturübersicht](Pie09)

Im Kompositionsmodell werden abstrakte Komponentenbeschreibungen in Form von Templates eingesetzt, welche später durch konkrete UI-Komponenten ersetzt werden. Alle verfügbaren Komponenten werden durch die Semantic Mashup Component Description Language (SMCDL) spezifiziert und beim Component Repository, welches im nächsten Abschnitt näher betrachtet wird registriert. Dies gewährleistet das Auffinden von Komponenten durch den eben erwähnten Integration Service.
2.4.1 Das Komponentenrepositorium


Zusammenfassend kann festgestellt werden, dass das Component Repository über eine definierte Schnittstelle die Komponentenbeschreibungen verwaltet und für das Auffinden von benötigten Komponenten sowie deren Bewertung verantwortlich ist. Der folgende Abschnitt beschreibt die Laufzeitumgebung von CRUISe und geht dabei auf wichtige Bestandteile von ihr näher ein.

2.4.2 Die Laufzeitumgebung

Für die Steuerung der Ereignis- und Datenflüsse zwischen den jeweiligen Komponenten ist die Thin-Server-Runtime (TSR) - die Laufzeitumgebung für CRUISe-Anwendungen - verantwortlich. Sie läuft komplett clientseitig im Browser und ist, wie beschrieben, unter anderem für die Kommunikation zwischen den Komponenten verantwortlich. Die essenziellen Bestandteile der TSR werden in diesem Zusammenhang nachfolgend näher erläutert [Sch11]:

- Für die Anforderung und Verwaltung der Komponenten ist der Component Manager verantwortlich. Er regelt demnach den Lebenszyklus jeder Komponente von der Integration, über die Instanzierung, Initialisierung und Ausführung, bis letztendlich zur Destruktion.
• Der **Integration Manager** ist das Bindeglied zwischen der Client Runtime und dem CoRe. Er nimmt Anforderungen des Component Managers entgegen und leitet sie an das Komponentenrepositorium weiter. Als Antwort erhält er vom CoRe Informationen, welche zur fehlerfreien Instanziierung und Visualisierung unentbehrlich sind. Dazu zählen der Integrationscode und Abhängigkeiten von externen Bibliotheken und Stylesheets.


• Für die Initialisierung der eben definierten Managerbestandteile ist der **Application Manager** verantwortlich, welcher den Rahmen der CRUISe Client Runtime bildet und eine Sicherung des Anwendungsstandes ermöglicht. Dieser ist durch die aktuelle Konfiguration der Komponenten, ihren Zuständen sowie globalen Variablen definiert.

Nachdem dieser Abschnitt einen Überblick über die Laufzeitumgebung von CRUISe gegeben hat, fasst der folgende Abschnitt die Defizite der Architektur, hinsichtlich der Migrationsfähigkeit zusammen.

### 2.4.3 Defizite von CRUISe

Nachdem ein Grundverständnis für die UI-Migration geschaffen, Softwarekomponenten und Komponentenmodelle sowie die Umsetzung dieser Konzepte im Forschungsprojekt CRUISe näher betrachtet wurden, folgt in diesem Abschnitt eine Identifikation derzeitiger Probleme und Defizite hinsichtlich der UI-Migration. Dabei liegt der Fokus auf der Extraktion und Serialisierung der Zustandsdaten einer Komponente sowie deren Injektion in eine andere Komponente.

• Wie in Abschnitt 2.3.3 beschrieben, wird die interne Struktur einer Komponente durch Events, Properties und Operations von dem Komponentenmodell teilweise vordefiniert. Die eigentliche Implementierung wird nicht vorgegeben (Black-Box-Paradigma). Demnach obliegt es dem Komponentenentwickler, ob er den sichtbaren Komponentenzustand (Properties) sehr feingranular oder sehr grobgranular bis gar nicht sichtbar modelliert. Im Prozess der Migration von UI-Komponenten ist generell ein sehr feingranularer Komponentenzustand nichtwichtig, um die Fortführung der migrierten Anwendungskomponente am Zielgerät gewährleisten zu können. Demnach kann es nach dem bisherigen Komponentenmodell zu Situationen kommen, in denen eine UI-Migration nicht ausgeführt werden kann.

• Des Weiteren bieten UI-Komponenten von CRUISe zur Zeit ausschließlich die Methoden *getProperty* und *setProperty*, welche lediglich den Zugriff auf den

- Darüber hinaus fehlt ein plattformneutrales und technologieunabhängiges Zustandsmodell. Es dient der Serialisierung von Zustandsdaten und ermöglicht den Austauschprozess heterogener Komponenten. Hierbei muss berücksichtigt werden, dass die interne Struktur der Komponenten bei einem möglichen Komponentenaustausch während der Migration variieren kann.


### 2.5 Zusammenfassung


Das nächste Kapitel präsentiert den aktuellen Stand der Forschung und Technik im Hinblick auf die Zustandserfassung und anschließende Extraktion, Serialisierung sowie Injektion dieser Zustandsdaten. Dabei wird zuerst auf den Ablauf des Komponentenaustausches in CRUIS-e eingegangen und anschließend werden die Forschungsprojekte CoBRA, OPEN und COSMOD näher vorgestellt. Der Fokus wird dabei stets auf Methoden und Vorgehensweisen gelegt, die eine UI-Migration ermöglichen können.
3 Stand der Forschung und Technik


Daran anschließend folgt eine Betrachtung des Forschungsprojektes CoBRA, wobei der Fokus, ähnlich wie bei CRUISe, auf die Atomarität sowie die Zustandserhaltung während der UI-Migration gelegt wird. In diesem Zusammenhang befasst sich der Abschnitt 3.2.2 mit der Zustandservassung durch Properties und illustriert die Verwendung des Entwurfsmusters Memento.


Abschnitt 3.4 befasst sich mit dem Forschungsprojekt COSMOD und beschreibt in diesem Kontext eine Möglichkeit zur Generierung von Zustandsgraphen.
3 Stand der Forschung und Technik


Eine Zusammenfassung der Untersuchungsergebnisse wird das Kapitel abschließen und veranschaulicht dabei allgemeine Erkenntnisse sowie nutzbare Ansätze für die eigene Konzeption, welche im weiteren Verlauf dieser Arbeit auf CRUISe übertragen werden könnten.

### 3.1 Austausch von Komponenten in CRUISe


#### 3.1.1 Sicherstellung der Isolation

sie nach Abschluss des Austauschprozesses in der richtigen Reihenfolge weiter abgearbeitet werden. Dieses Verfahren stellt die Konsistenz sowie die Integrität der Gesamtanwendung sicher und kann für die UI-Migration übernommen werden. Der folgende Abschnitt beschreibt den Ablauf des Komponentenaustausches und geht dabei auf das erweiterte Phasen-Commit-Protokoll ein.

### 3.1.2 Ablauf des Komponentenaustausches


1. **Block** Der Adaption Manager aktiviert vor jedem Austauschprozess eine Synchronisationssperre, um die Isolation zu gewährleisten und die Anwendung vor dem Erreichen eines inkonsistenten Zustandes zu bewahren. Außerdem wird die Ausführung weiterer Adaptionen bis zur Fertigstellung des Austauschprozesses ausgesetzt.

2. **Prepare** Diese Phase bereitet den Austausch der inaktiven Komponente A vor.
   1. Zuerst wird der Wrapper der Komponente A blockiert, um das Weiterleiten von eintreffenden Nachrichten an die Komponente zu unterbinden.

3. **Check** Der weitere Verlauf wird durch das Ergebnis der Prepare-Phasen bestimmt:
   (a) Sobald die Komponente A und der Component Manager ready signalisieren, beginnt die Commit-Phase (4).
   (b) Bei einem Timeout oder im Fehlerfall (auch während des Commit - wird die Abort-Phase (5) eingeleitet.)
4. **Commit** Die folgenden Schritte werden für den Austausch von Komponenten A durch Komponente B abgearbeitet:

1. Ist die Komponente A eine sichtbare UI-Komponente, wird sie durch den Aufruf ihrer Methode `hide()` versteckt.
3. Nun ruft der Component Manager die Komponentenbeschreibung von Komponente B vom CoRe ab und liest die enthaltenen Informationen ein.
4. Für die Komponente B wird ein Wrapper instantiiert, welcher ebenfalls initial blockiert ist und somit keine Nachrichten weiterleitet.
5. Komponente B wird instantiiert.
8. Anschließend erfolgt der Zustandstransfer zwischen den beiden Komponenten, welcher in Abschnitt 3.1.3 beschrieben wird.
9. Falls Komponente A zuvor sichtbar war, wird durch den Aufruf der Methode `show()` die Komponente B sichtbar gemacht.

5. **Abort** Der Component Manager behandelt Fehler, die während der Prepare- oder Commit-Phase auftreten können.

- **Sicherungsfehler** Wenn Komponente A keinen sicheren Zustand erreicht und somit die Prepare-Phase fehlschlägt, wird die Blockierung aufgehoben und der Austauschprozess abgebrochen.
- **Ladefehler** Falls Komponente B, aufgrund von Verbindungsproblemen zum Beispiel, nicht korrekt geladen werden kann, wird Komponente A wieder aktiviert und gegebenenfalls eingeblendet.
- **Instanzierungsfehler** Kann Komponente B nicht instantiiert werden, wird Komponente A ebenfalls wieder aktiviert und der Wrapper von Komponente B wird entfernt.
- **Integrationsfehler** Sollte die Integration fehlerhaft verlaufen, so müssen alle Verweise zurück auf den Wrapper von Komponente A gesetzt werden und der Abbruch wie oben erfolgen.
- **Initialisierungsfehler** Bei einem Initialisierungsfehler wird wie bei einer fehlerhaften Integration vorgegangen.
6. Unblock  In dieser letzten Phase werden die Adaptionss- und Kommunikationsperren gelöst.


3.1.3 Transfer des Komponentenzustandes

Der Zustand einer Komponente ist in CRUISe durch die Summe der Properties sowie durch deren Wert charakterisiert. Andere Informationen sollen der Kompositionsumgebung, aufgrund der Unabhängigkeit der Komponentenimplementierung beispielsweise, nicht zur Verfügung stehen.

Die Sicherung sowie die Wiederherstellung von Zuständen wird mit Hilfe des Memento-Entwurfsmusters realisiert, welches das Zusammenspiel von Orginator (Komponenten) und Caretaker (Adaption Manager) beschreibt [Gam+95]. Das Auslesen der Belegungen aller Properties von Komponente A sowie das Setzen dieser in Komponente B sieht das Komponentenmodell von CRUISe die Methoden getProperty(key) und setProperty(key) vor. Durch das optionale Attribut transient kann der Komponentenentwickler Properties in der MCDL als flüchtig kennzeichnen. Damit gibt er an, dass diese Properties bei einem Austauschprozess nicht gesichert werden müssen, falls sie sich zum Beispiel durch interne Berechnungen ergeben. Die folgenden Punkte beschreiben überblicksweise den Zustandstransfer bei einem Austauschvorgang:


2. Anschließend werden die ermittelten Eigenschaften durch den Aufruf der Methode getProperty(key) ausgelesen und gesichert.


Es besteht die Möglichkeit, dass Komponente A mehr Properties besitzt, als die Vorlage verlangt. Demnach ist es ausgeschlossen, dass alle Eigenschaften auf Komponente B übertragen werden können. Zunächst ist dies unproblematisch, da alle prinzipiell notwendigen Properties repräsentiert sein sollten und somit die restlichen
ignoriert werden können. Langfristig kann dieses Vorgehen allerdings zum Datenverlust führen, wenn beispielsweise ein späterer Wechsel zurück zu Komponente A vollzogen wird. Da eine Zwischenspeicherung der unnötigen Eigenschaften von Komponente A durch die Laufzeitumgebung bis zu einem späteren Einsatz zu Inkon sistenzen führen kann, lässt sich das Problem unter den gegebenen Randbedingungen nicht vollständig lösen.

Nachdem dieser Abschnitt beschrieben hat, in welcher Art und Weise im Forschungsprojekt CRUISe der Zustand einer UI-Komponente von einem Ausgangsgerät auf ein Zielgerät übertragen wird, soll der nächste Abschnitt beschreiben, wie das Problem der Wertzuweisung unterschiedlich benannter Properties zweier Komponenten in CRUISe gelöst wurde.

**Mapping von Zustandsvariablen**

Aufgrund der Tatsache, dass in den meisten Fällen eines Komponentenaustausches die involvierten UI-Komponenten von verschiedenen Komponentenentwicklern implementiert wurden, können diese beiden Komponenten bspw. unterschiedliche Variablennamen besitzen. Das daraus resultierende Problem ist die Frage, mit welchen Mitteln festgestellt werden kann, welcher Eigenschaftswert der Ursprungs komponente an welchen Eigenschaftswert der Zielkomponente übergeben werden muss.

In CRUISe wird dieses Problem von dem *Mediator*, welcher Teil der Laufzeitumgebung ist, mit Hilfe des *Semantic Matchings* gelöst. Dieses Vorgehen wird durch die Verwendung der *Semantic Mashup Component Description Language* (siehe Abschnitt 2.4) ermöglicht. Mit Hilfe dieser Beschreibungssprache ist es beispielsweise möglich, eine Property mit beliebigem Namen semantisch zu typisieren. Dies wird an dem nachfolgenden Codebeispiel 3.1 demonstriert:

```
<property name="Standort" type='vvo:Location'/>
```

Listing 3.1: Semantische Typisierung


### 3.1.4 Zusammenfassung

Zusammenfassend lässt sich sagen, dass der Austauschprozess einer Komponente in CRUISe eine gute Grundlage für den UI-Migrationsvorgang bereitstellt. Das Proxy-Konzept beispielsweise, gewährleistet die Isolation des Vorganges sowie die Schnitt-


Nachdem in diesem Abschnitt der Komponentenaustausch in CRUISe näher betrachtet wurde, soll der folgende Abschnitt die Architektur CoBRA in Hinblick auf eine dynamische Adaption untersuchen und dabei ebenfalls den Fokus auf die Zustandserfassung, -extraktion, -übertragung und -injektion setzen.

### 3.2 Atomarität und Zustandserhaltung in CoBRA

Irmert et al. präsentieren in [IFM08] die Architektur CoBRA (Component Based Runtime Adaptable). CoBRA ermöglicht es Softwarekomponenten sich selbst zu verwalten sowie sich dynamisch, während der Laufzeit, zu adaptieren. Wie in [IFM08] beschrieben, setzt eine dynamische Adaption, bzw. der Austausch einer Service-Implementierung während der Laufzeit, die folgenden Punkte voraus:

- **Transparenz:** Der Austausch eines Services muss für die beteiligten Anwendungen transparent ablaufen. Dabei soll es für den Komponentenentwickler nicht notwendig sein, zusätzliche Methoden für die Ersetzung der Abhängigkeiten implementieren zu müssen.

- **Atomarität:** Der Prozess darf nicht unterbrochen werden. Aus diesem Grund dürfen andere Services oder Anwendungen nicht auf die auszutauschende Komponente zugreifen, währenddessen sie sich in dem Austauschvorgang befindet.

- **Zustandserhaltung:** Wichtige Attribute eines Services dürfen während des Austauschprozesses nicht verloren gehen. Demnach ist es wichtig, den Zustand der ursprünglichen Komponentenversion zu sichern und ihn in die neue Version zu injizieren.

- **Management des Lebenszyklus:** Der gesamte Austauschvorgang, dazu zählt die Zustandsextraktion, die Ersetzung der Komponente sowie die Zustandsinjektion, muss von der Laufzeitumgebung koordiniert und überwacht werden.

Diese Voraussetzungen lassen sich komplett auf den UI-Migrationsprozess übertragen und zeigen ein weiteres Mal die Notwendigkeit der Atomarität und Zustandserhaltung.
serhaltung. Die folgenden zwei Abschnitte beschreiben wie diese eben erwähnten Voraussetzungen in CoBRA realisiert werden und gehen dabei speziell auf die Verwendung bzw. auf die Umsetzung des Proxy- und Memento-Patterns ein, welche auch in CRUISe implementiert sind.

3.2.1 Atomarität mit Hilfe des Protection-Proxy-Patterns


Abbildung 3.1: Protection proxy pattern in CoBRA [IFM08]


Abbildung 3.2 zeigt in Form eines Zeitdiagramms den gesamten Adapptionsvorgang, welcher in 3 Phasen aufgeteilt ist (store phase, switch phase sowie restore phase) und als nächste näher betrachtet wird:


In der store phase wird der ermittelte Komponentenzustand des Ausgangsservices
(B1) in einem State-Container-Objekt gespeichert. Realisiert wird dieser Vorgang mit Hilfe des Memento Patterns [Gam+95], welches im folgenden Abschnitt 3.2.2 näher betrachtet wird.


Daran anschließend wird in der restore phase der Zustand wiederhergestellt und der Proxy über die Injektion des neuen Services benachrichtigt, damit er seine Referenzen aktualisiert und die Blockierung eingehender Anfragen deaktiviert. Damit ist der Adaptionprozess abgeschlossen und konnte sowohl atomar, als auch transparent durchgeführt werden.

Im folgenden Abschnitt wird das verwendete Memento-Entwurfsmuster [Gam+95] näher beschrieben.

Abbildung 3.2: Adaptionprozess in CoBRA [IFM08]

### 3.2.2 Zustandserhaltung mit Hilfe des Memento-Patterns


Copyright TU Dresden, Christoph Pohl 29
In Abbildung 3.3 ist die grundlegende Struktur des Memento-Entwurfsmusters zu sehen. Die beteiligten Komponenten sowie deren Aufgaben und Funktionen werden jetzt näher beschrieben:

- Der **Originator** repräsentiert eine Komponente, welche die Funktionen `SetMemento(Memento m)` und `CreateMemento()` beinhaltet sowie deren Zustand (`state`). Die erstgenannte Funktion dient dabei der Wiederherstellung eines Zustandes. Die Funktion `CreateMemento()` erstellt ein Memento-Objekt, welches den derzeitigen Zustand der Komponente beinhaltet (`return new Memento(state)`).

- Das **Memento**-Objekt speichert wie eben beschrieben den Zustand der jeweiligen Komponente und besitzt 2 Interfaces. Ein Interface wird von dem **Originator** genutzt, welcher vollen Zugriff auf die Daten des Memento-Objektes besitzt, um bspw. einen Komponentenzustand wiederherzustellen. Das zweite Interface wird vom **Caretaker** genutzt, welcher im nächsten Punkt erläutert wird.

- Der **Caretaker** ist für die Aufbewahrung des Memento-Objektes zuständig und besitzt keine Rechte es zu bearbeiten oder zu untersuchen.

Nachdem die beteiligten Komponenten des Memento-Patterns beschrieben wurden, soll das folgende Sequenzdiagramm ein Verständnis für das Zusammenspiel aller
Komponenten schaffen. Wie in Abbildung 3.4 dargestellt, fordert der Caretaker mit Hilfe der `CreateMemento()`-Methode von dem Originator ein Memento-Objekt an. Der Originator erstellt darauf ein neues Memento-Objekt, befüllt dieses mit seinem aktuellen Zustand (`setState()`) und gibt es anschließend an den Caretaker zurück. Für den Fall, dass der Caretaker eine Zustandswiederherstellung auslöst (`SetMemento(Memento)`), holt sich der Originator, mit Hilfe der Methode `GetState()`, von dem Memento-Objekt den gespeicherten Zustand und stellt diesen wieder her.

### 3.2.3 Zusammenfassung


### 3.3 Die OPEN Migration Service Platform


Aufgrund dieser Anforderung entschieden sich die Entwickler dazu keine kompletten Programminstanzen zu migrieren. Stattdessen wird ausschließlich der Anwendungszustand der Ausgangsanwendung extrahiert. Anschließend wird diese Instanz beendet und eine neue Instanz, inklusive dem extrahierten Zustand, auf der Zielplattform, welche ausreichend Ressourcen hat, gestartet. Dieser Ansatz setzt voraus, dass für jede Plattform eine passende Version der Anwendung zur Verfügung gestellt wird. Im Gegensatz dazu, muss der Anwendungszustand allgemeingültig bleiben, damit er von jeder Implementation der Anwendung interpretiert werden kann, unabhängig davon ob alle Bestandteile des Anwendungszustandes genutzt werden. Die OPEN-Plattform soll dabei so viel wie möglich Migrationsfunktionalitäten von...
der eigentlichen Anwendung übernehmen, um dem Anwendungsentwickler Implementierungsarbeit zu ersparen. Der folgende Abschnitt dient der überblicksartigen Beschreibung der Architektur der OPEN-Plattform.

3.3.1 Architektur der OPEN-Plattform im Überblick


Die OPEN-Adaptoren

Die in Abbildung 3.5 gezeigte Architekturübersicht beinhaltet unter anderem den internen Aufbau eines OPEN-Clients und zeigt, dass die OPEN-Plattform konventionelle (Client-Server-)Web-Anwendungen unterstützt. Der Client ist in diesem speziellen Beispiel eine Kombination aus einem Terminal (ein Smartphone) und einem Application Server. Sowohl Client, als auch Server besitzen Teile der Anwendung sowie ein Set von OPEN-Adaptoren. Diese Adaptoren sind allgemeingültig für alle Anwendungen und demnach mehrfach verwendbar, solange es die zugrundeliegende Plattform zulässt. Des Weiteren kommunizieren sie mit den benötigten Serverkomponenten (Open Server Component) und implementieren so diverse Migrationsfunktionalitäten, welche nachfolgend näher erläutert werden. Ausschlaggebend für die Wahl der beschriebenen Adaptoren sind die Funktionalitäten, welche für die vorliegende Forschungsarbeit relevant sind:

Migration Orchestration Adapter: Dieser Adapter führt die Anweisungen seines serverseitigen Gegenparts aus und kann beispielsweise Anwendungen bei Be-
Modellierung des Komponentenzustandes

Stand der Forschung und Technik

darf pausieren / fortfahren, um einen konsistenten Zustandstransfer zwischen den beteiligten Komponenten zu gewährleisten.

**State Handler Adapter:** Dieser Adapter extrahiert den Zustand der Web-Anwendung und stellt ihn für den serverseitigen *State Handler* bereit, welcher die Zustandsinformationen vom Ausgangs- zum Zielgerät transferiert.

**Abbildung 3.5: OPEN - Architektur [Pat11]**

Neben diesen zwei Adapters stellt die OPEN-Plattform weitere Module bereit, welche beispielsweise den Zeitpunkt der Migration aufgrund von Kontextinformationen festlegen, oder bestimmte QoS-Parameter (z. B. die aktuelle Netzwerkleistung) überwachen.


**Die OPEN-Dispatcher**


Um diese Herausforderung zu bewältigen wird das *Dispatcher Pattern* [Gam+95] verwendet, bei dem ein Dispatcher-Modul als alleinstehender Endpunkt für mehrere Komponenten agiert und ankommende Nachrichten entgegennimmt. Gleichzeitig verwenden alle Komponenten dieses Modul, um eigene Nachrichten nach außen zu
Modellierung des Komponentenzustandes 3 Stand der Forschung und Technik

senden. Abbildung 3.5 zeigt alle 3 verschiedenen Typen der Dispatcher, welche nachfolgend kurz beschrieben werden:


- Jeweils ein **OPEN-Adaptor Dispatcher** ist an eine Anwendung gebunden und agiert als Proxy, stellvertretend für alle Funktionalitäten der Plattform.

Nachdem dieser Abschnitt die Architektur der OPEN-Plattform überblicksweise vorgestellt hat, fokussiert sich der folgende Abschnitt auf die Erfassung, Extraktion und Injektion der Komponentenzustände.

### 3.3.2 Migration von Komponenten in OPEN


#### Ablauf der UI-Migration

Um dem Anwendungsentwickler Implementierungsarbeit zu ersparen und den Vorgang der UI-Migration zu ermöglichen, wird in OPEN eine **Reverse Engineering**-Technik angewandt, welche zunächst eine *abstrakte Beschreibung* der Benutzerschnittstelle aus dem (X)HTML-Quellcode der pausierten Anwendung generiert. „Abstrakt“ bedeutet in diesem Zusammenhang, dass das beschriebene User-Interface unabhängig von der Programmiersprache sowie von den Ein- und Ausgabemodalitäten ist. Die dabei verwendete Beschreibungssprache ist *MARIA XML* [Pat09], welche
auf der Beschreibungssprache $TERESA\ XML$ basiert und von dem Forschungsprojekt $ServFace$ übernommen und erweitert wurde. Beinhaltet die Web-Anwendung Flash-Komponenten oder Java Applets, so ist es nicht möglich deren Quellcode zu analysieren. In diesem Fall werden die betroffenen Komponenten durch alternative Komponenten ersetzt oder umgepafstet an das Zielgerät weitergeleitet.

Im nächsten Schritt wird die $abstrakte\ Beschreibung$ der Benutzerschnittstelle an die Zielplattform angepasst und kann ab diesem Zeitpunkt als eine $konkrete\ Beschreibung$ angesehen werden. Diese ist im Gegensatz zur abstrakten Beschreibung modalitätsabhängig und programmiersprachenunabhängig.

Der Zustand der zu migrierenden Web-Seite wird anschließend durch einen $State\ Mapper$ ermittelt und auf die konkrete Beschreibungen übertragen. Die aktuellen Zustandsinformationen holt sich der $State\ Mapper$ dabei aus dem DOM\(^1\)-Baum der Ausgangsseite indem er alle Eigenschaften dieser Web-Seite ausliest (siehe Abschnitt 3.3.2).

Im letzten Schritt wird von einem $Generator$ auf Grundlage der $konkreten\ Beschreibung$ die angepasste Web-Seite generiert und anschließend auf das Zielgerät übertragen. Wurde die UI-Migration erfolgreich abgeschlossen, wird die pausierte Anwendung auf dem Zielgerät fortgesetzt.

Nachdem dieser Abschnitt überblicksweise den Prozess der UI-Migration in OPEN beschrieben hat, erläutert der folgende Abschnitt wie der Zustand einer UI-Komponente erfasst und extrahiert bzw. injiziert wird.

**Erfassung sowie Extraktion und Injektion des Komponentenzustandes**


![Abbildung 3.6: OPEN - Migration eines PC-Spiels][1]

Wie in Abbildung 3.6 dargestellt, soll die Desktop-Version von „PacMan“ auf ein Smartphone migriert werden. Die Desktop-Variante des Spiels (linke Bildhälfte) bietet dem Nutzer mehr Elemente / Einstellungsmöglichkeiten auf einer Web-Seite als

\(^1\)Document Object Model

### 3.3.3 Herausforderungen bei der Serialisierung von JS-Variablen

Wie bereits im vorherigen Abschnitt beschrieben, verfügt die OPEN Migration Service Platform über Methoden, welche es ermöglichen globale JavaScript-Variablen auszulesen und in jeweils einem JSON-Objekt zu speichern. Dabei ist ein JSON-Objekt in OPEN wie folgt aufgebaut: `{Name:Wert}` Alle JSON-Objekte werden nacheinander in ein Array gepackt, welches zusammen mit dem DOM-Objekt an den Migration Server übertragen wird.


---

²JavaScript Object Notation
³https://dojotoolkit.org
Globale und BOM-Variablen


Objektreferenzen

Bei Objektreferenzen handelt es sich um zwei Variablen oder Objekte, welche auf den selben Wert verweisen, wobei der Datentyp komplex ist. Folgendes Beispiel soll diesen Sachverhalt verdeutlichen:

```
1 var x = <einObjekt>;
2 var y = x;
```

Listing 3.2: Objektreferenzen

In dieser Situation würde JSON normalerweise das Objekt <einObjekt> zwei mal serialisieren und jeweils einmal für die Variable x sowie für die Variable y in zwei separaten Objekten abspeichern. In dem Ansatz von OPEN hingegen, wird die Variable y mit einer eindeutigen Referenz auf die Variable x versehen. Somit wird eine doppelte Serialisierung verhindert und zugleich sichergestellt, dass auch nach der Migration die Variable y noch immer auf die Variable x referenziert.

Zyklische Referenzen

Zyklische Referenzen stellen ein Spezialfall von Objektreferenzen dar. Hierbei definiert beispielsweise Variable A eine andere Variable B, wobei A selbst in B definiert wird. Folgender JavaScript-Ausschnitt soll dieses Problem verdeutlichen:

```
1 var peterLustig = {
2   name : 'Peter',
3   vater : {
4     name : 'Paul',
5     sohn : peterLustig
6   }
7 }
```

Listing 3.3: Zyklische Referenzen

Bei diesem Beispiel würde JSON normalerweise in einer Endlosschleife stecken bleiben oder eine Fehlermeldung ausgeben, weil die Serialisierung der Variablenreferenz einen rekursiven Aufruf erzeugt. Um dieses Problem zu lösen, wird in OPEN, ähnlich wie im ersten Beispiel, lediglich eine Referenz auf das Objekt gespeichert und nicht

**Timer**


```
1 var timerId = setTimeout (function , ms);
2 clearTimeout (timerId);
```

Listing 3.4: Timer


**Objekte, die ein Datum enthalten**


**Dynamische Zuweisung von Werten zu Objekten**

Alle nicht-primitiven Datentypen in JavaScript, wie beispielsweise (assoziative) Arrays oder Strings, sind Objekte, welche während der Laufzeit dynamisch mit Eigen-
schaften erweitert werden können. Diese Werte werden von JSON nur dann berück-
sichtigigt und erfasst, wenn das zu serialisierende Objekt ein assoziatives Array ist. 
Das folgende Codebeispiel soll diese Problematik verdeutlichen:

```javascript
var array = [wert1, wert2];
array.dynEig = einWert;
```

Listing 3.5: Dynamische Wertzuweisung

Für die Lösung dieses Problems wurde der Serialisierer der Library dojoX dahinge-
hend erweitert, dass jedes Array in einem Objekt gekapselt wird, welches zusätzlich 
eine Liste mit allen dynamisch hinzugefügten Eigenschaften enthält. Diese Vorge-
hensweise lässt sich analog auf andere Objekte wie bspw. Strings anwenden.

Referenzen auf DOM-Knoten

Mit JavaScript ist es möglich einzelne Knoten des DOM-Baums einer Web-Seite 
zu referenzieren, um deren Werte auszulesen oder gar zu verändern. Für die Erhal-
tung des Zustandes ist es wichtig, dass auch nach der Migration der Zugri
auf den 
DOM-Baum möglich ist. Um auf die einzelnen Elemente des DOM-Baums zuzu-
greifen wird häufig die Methode `getElementById` genutzt, welche eine Referenz zu 
dem gewünschten Knoten innerhalb des DOM liefert. Unglücklicherweise kann JSON 
JavaScript-Variablen, welche eine Referenz auf einen DOM-Knoten beinhalten, nicht 
korrekt serialisieren, da nicht sichergestellt werden kann, ob das aktuell untersuchte 
Element eine ID besitzt oder nicht. Darüber hinaus ist es außerdem möglich, dass 
das Element überhaupt kein Bestandteil des DOM ist. Das folgende Code-Beispiel 
soll die Problematik etwas verdeutlichen:

```javascript
var element = getElementById("myHtmlElement");
var image = new Image("imageSource");
```

Listing 3.6: Referenzen auf DOM-Knoten

In OPEN wird bei Objekten des Typs „HtmlElement“, welche eine Referenz auf 
ein DOM-Knoten sind, geprüft, ob diese eine ID besitzen. In diesem Fall wird die 
ID für die Zustanderhaltung abgespeichert. Andernfalls wird eine ID generiert und 
dem Objekt zugewiesen. Sollte das Objekt nicht Teil des DOM-Baums sein, wie bei-
spielsweise die Variable „image“ des oberen Beispiels, so werden alle zur Verfügung 
stehenden Eigenschaften dieses Objektes mit Hilfe der JavaScript Library JsonML⁴ 
ausgelesen und abgespeichert.

3.3.4 Zusammenfassung

Zusammenfassend lässt sich sagen, dass mit Hilfe der OPEN-Plattform die Migra-
tion einer Web-Anwendung zwischen zwei Geräten realisiert werden kann und aus 
Sicht des Anwenders somit ein unterbrechungsfreies Arbeiten mit dieser Anwen-
dung, selbst bei einem Gerätewechsel, möglich ist. Ein weiteres hervorstechendes 
Merkmal dieses Forschungsprojektes ist die Tatsache, dass sich Komponentenent-
wickler keine Gedanken über die Implementierung von Methoden, welche für die 
Migration verantwortlich sind, machen müssen. Dies wird durch die Verwendung

⁴http://jsonml.org/


### 3.4 COSMOD: Generierung von Zustandsgraphen


- Zunächst wird eine Liste aller verfügbaren Methoden mit Hilfe der Schnittstellenbeschreibung generiert.

- Anschließend werden all diese Methoden $m_{\theta_u}$ aus der soeben generierten Liste, zuerst im Anfangszustand $Z_0$ und anschließend in allen ermittelten Folgezuständen ausgeführt, wobei mögliche Eingabeparameter $p_{\theta_u}$ per Zufall generiert werden. Ein Zustandswechsel wird dabei mit Hilfe der zur Verfügung stehenden Methoden bestimmt. So wird jederzeit überprüft, welche Teilmengen aller Methoden $m_{\theta_u}$ aktuell ausgeführt werden können und welche Methoden entweder eine Fehlermeldung generieren oder eine Nachricht anzeigen, dass die gewählte Methode derzeit nicht zur Verfügung steht.

- Bei Erreichung eines neuen Zustandes $Z_u$ ausgehend von $Z_0$ aufgrund der Ausführung von $m_{\theta_u}$ mit den Werten $p_{\theta_u}$ wird die Sequenz $m_{\theta_u}(p_{\theta_u})$ abgespeichert.

\[ ^5 \text{"reverse engineering COmponent State MOdel from its external behavior"} \]
• Wird ausgehend von $Z_u$ ein weiterer Zustand $Z_v$ erreicht, so wird die Sequenz erweitert und sieht für $Z_u$ wie folgt aus: $m_{0u}(p_{0u}), m_{uv}(p_{uv})$. Demzufolge besteht eine Sequenz der Zustände, mit Ausnahme der direkten Nachfolger von $Z_0$, aus zwei oder mehr Methoden.


Für den beschriebenen Algorithmus wird in [SM09] eine polynomiale Zeitkomplexität von $T = O(k^2 \cdot (alle\Zustande^2 + alle\Zustande \cdot \sum_{i=1}^{k} T(m_i)))$ angegeben, wobei $k$ die Anzahl der Methoden und $\sum_{i=1}^{k} T(m_i)$ die Summe der Zeitkomplexitäten jeder einzelnen Methode ist.

Inwiefern diese Methode nützlich für die UI-Migration in CRUISe ist, soll im Kapitel 4 geklärt werden. Der nächste Abschnitt dieses Kapitels befasst sich mit DOM-Mutation-Events, mit deren Hilfe es möglich ist, auftretende Veränderungen innerhalb des DOM-Baumes festzustellen.

### 3.5 Registrierung auftretender (DOM-Baum-)Events


Unter der Annahme, dass alle registrierten Strukturveränderungen des DOM-Baums gefiltert werden können, sodass letztendlich ausschließlich zustandsrelevante Informationen gespeichert werden, stellt diese Technik, neben der Vorgehensweise von OPEN, einen weiteren möglichen Ansatz zur Zustandsextraktion dar. In diesem Zusammenhang müsste noch geklärt werden, inwiefern sich dieser Ansatz auf andere Programmiersprachen wie bspw. Flash übertragen lässt. Aufgrund der Tat-

### 3.6 Zusammenfassung


Im darauffolgenden Abschnitt wurde das Forschungsprojekt CoBRA in Bezug auf den dynamischen Austausch von Service-Komponenten untersucht. Dabei wurden zunächst Voraussetzungen für diesen Vorgang beschrieben. Anschließend wurden die verwendeten Entwurfsmuster Protection-Proxy-Pattern und Memento, welche auch in CRUISe verwendet werden, näher untersucht. Trotz der Verwendung identischer Entwurfsmuster, existieren konzeptionelle Unterschiede in beiden Ansätzen. Beispielsweise gehen in CoBRA Nachrichten verloren, welche während des Austauschprozesses an die betroffene Komponente gesendet werden. Bei CRUISe hingegen,
werden diese Nachrichten gepuffert und nach dem Austauschprozess der Reihenfolge nach an die neue Komponente weitergeleitet.


Der darauffolgende Abschnitt hat die Mutation-Observer-Technologie vorgestellt, mit der es ermöglicht wird, alle auftretenden Strukturveränderungen eines DOM-Baums zu erfassen. In diesem Zusammenhang wurde der Kerngedanke dieses Ansatzes näher betrachtet und festgestellt, dass dieser theoretisch auch auf weitere Programmiersprachen übertragbar ist.

Inwiefern diese Ansätze für die UI-Migration in CRUISe in Frage kommen, wird im folgenden Kapitel 4 diskutiert. In diesem Zusammenhang werden zunächst Anforderungen an eine Migrations-Umgebung aufgestellt und anschließend werden die vorgestellten Konzepte miteinander verglichen und evaluiert. Des Weiteren wird darauf aufbauend das Gesamtkonzept für die UI-Migration in CRUISe vorgestellt und die wichtigen Teilaspekte Zustandserfassung und -extraktion sowie -injektion näher beschrieben.
4 Konzeption der UI-Migration


4.1 Anforderungsanalyse


4.1.1 Funktione Anforderungen

Die zu erarbeitende Konzeption soll eine (partielle) UI-Migration einer durch CRUISe bereitgestellten, kompositen Web-Anwendung ermöglichen. Dieser Prozess soll dabei den in Abschnitt 2.2 erläuterten Prinzipien genügen. Nachfolgend werden dafür funktionale Anforderungen aufgestellt.

Anforderungen an die Komponenten

Die folgenden funktionalen Anforderungen beziehen sich auf den Aufbau und die Methoden des Komponentenmodells.

1 Bereitstellung von Methoden zum Austausch der Zustandsdaten: Die UI-Komponenten müssen in der Lage sein ihren aktuellen Zustand bereitzustellen sowie erhaltene Zustandsdaten zu injizieren.

**Anforderungen an die Laufzeitumgebung**

Die folgenden Anforderungen beziehen sich auf die Ausführung des Migrationsprozesses.

**3 Erweiterung des Austauschprozesses von UI-Komponenten:** Die Laufzeitumgebung muss, im Rahmen der UI-Migration, den Austausch von UI-Komponenten durch andere UI-Komponenten unterstützen. Aufgrund der Tatsache, dass CRUSe bereits den Austausch von UI-Komponenten unterstützt, muss dieser nicht komplett neu entwickelt werden, sondern lediglich um das entwickelte Zustandsmodell erweitert werden.

**4 Übertragung der Informationen:** Die serialisierten Zustandsinformationen müssen ausgehend von dem Ausgangsgerät, über den Migration Server, hin zu dem Zielgerät von der Laufzeitumgebung übertragen werden.

**Anforderungen an das Zustandsmodell**

Die folgenden Anforderungen beziehen sich auf den Aufbau des technologie- und plattformneutralen Zustandsmodells.


**4.1.2 Nicht-funktionale Anforderungen**


**6 Plattformunabhängigkeit:** Der Migrationsprozess muss auf verschiedenen Plattformen ausführbar sein.

**7 Intuitive Benutzerinteraktionen:** Um eine reibungslose Interaktion zwischen dem Nutzer und der zu migrierenden Anwendung zu gewährleisten, muss die Auslösung einer UI-Migration unkompliziert und intuitiv sein. Dabei sollen ausschließlich vom Nutzer ausgelöste Migrationen von der Laufzeitumgebung unterstützt werden.

**8 Stabilität:** Die Abarbeitung einzelner Prozessschritte durch die Laufzeitumgebung müssen zuverlässig und möglichst fehlerfrei durchgeführt werden. Des Weiteren darf die UI-Migration im Fehlerfall nicht die Stabilität der komponenten Anwendung gefährden.
9 Effizienz: Der Migrations-Prozess soll möglichst geringen Einfluss auf das Laufzeitverhalten der gesamten Web-Anwendung ausüben. Das Konzept soll eine zeitleich effiziente und ressourcenschonende Abarbeitung der einzelnen Teilprozesse gewährleisten.

10 Reduzierung des Entwicklungsaufwandes: Der Implementierungsaufwand für die Unterstützung der UI-Migration soll auf der Seite des Komponentenentwicklers so gering wie möglich gehalten werden. Demnach soll beispielsweise verhindert werden, dass für jede Komponente ein passender Serialisierungsalgorithmus implementiert werden muss.

11 Benutzbarkeit: Nach Abschluss einer erfolgreichen UI-Migration soll der Anwender die migrierte Komponente anhand des wiederhergestellten Zustandes wiedererkennen können.

12 Kontinuität: Die Web-Anwendung muss sich nach der Migration genauso verhalten wie vor dem Migrationsprozess.

Anhand dieser aufgestellten funktionalen und nicht-funktionalen Anforderungen werden im nächsten Abschnitt Kriterien für den Vergleich der in Kapitel 3 beschriebenen Konzepte aufgestellt.

4.2 Evaluation der untersuchten Ansätze

Dieser Abschnitt soll die im vorherigen Kapitel untersuchten Ansätze bewerten und in diesem Zusammenhang verwertbare Technologien herausfiltern, welche im weiteren Verlauf dieses Kapitels in einem eigenen Konzept wiederverwendet werden können. Dazu werden zunächst Kriterien definiert, welche sich aus den funktionalen und nicht-funktionalen Anforderungen an eine Laufzeitumgebung (4.1) ergeben. Im Anschluss daran wird die CRUISe-Architektur in Bezug auf die UI-Migration bewertet. Abschließend werden die beschriebenen Konzepte von CoBRA, OPEN und COSMOD auf Grundlage der eingangs definierten Kriterien miteinander verglichen.

4.2.1 Kriterienkatalog

Die in Abschnitt 4.1 erarbeiteten funktionalen und nicht-funktionalen Anforderungen, welche eine Laufzeitumgebung besitzen muss, um die UI-Migration erfolgreich und zufriedenstellend durchzuführen, bilden die Grundlage für den Kriterienkatalog, welcher anschließend definiert und erläutert wird.


2. Entwicklungsaufwand für den Komponentenentwickler: Der Grad der Automatisierung gibt im Kontext der vorliegenden Untersuchungen an,
wie hoch der verbleibende Implementierungsaufwand des Komponentenentwicklers, in Bezug auf die Erfassung der Zustandsinformationen einer UI-Komponente, ist. Je höher der Automatisierungsgrad eines Ansatzes, desto weniger Aufwand hat der Komponentenentwickler die UI-Migration zu implementieren. Im besten Fall sollte somit die UI-Migration für den Komponentenentwickler vollkommen transparent ausgeführt werden.


4. **Entwicklungsaufwand zur Erweiterung der Thin-Server-Runtime:** Das letzte Vergleichskriterium untersucht das Verhältnis zwischen dem Implementierungs- bzw. Integrationsaufwand (in die bestehende CRUISe-Architektur) und dem daraus resultierenden Nutzen für die UI-Migration.

Nachdem dieser Abschnitt grundlegende Kriterien für eine vergleichende Betrachtung definiert hat, werden nachfolgend zunächst die Vor- und Nachteile, der in Kapitel 3 vorgestellten Technologien, aufgezeigt. Anschließend werden die präsentierten Konzepte miteinander verglichen.

### 4.2.2 Bewertung der untersuchten Konzepte


**CRUISe**


(-) **Nutzbarkeit des erfassbaren Zustandes:** In CRUISe wird derzeitig der Zustand einer Komponente über die Summe der Properties dieser Komponente bzw. deren Werte definiert. Dabei werden ausschließlich alle nicht-flüchtigen Eigenschaften (vgl. Abschnitt 3.1.3) von der zu migrierenden Komponente berücksichtigt, welche mit Hilfe der entsprechenden MCDL ermittelt werden. Demzufolge werden Variablen, welche nicht als Teil der Schnittstelle definiert wurden, oder beim Komponentenentwickler in Vergessenheit geraten sind, bei...


Nachdem CRUISe unter den zwei Aspekten Nutzbarkeit des erfassbaren Zustandes und Entwicklungsaufwand für den Komponentenentwickler untersucht wurde, beschreibt der nächste Abschnitt zusammenfassend mit welchen Methoden die Komponenten von der Ablaufumgebung während des Austauschprozesses isoliert werden.


Ein weiterer positiver Aspekt in CRUISe ist das in Abschnitt 3.1.3 bereits beschriebene Semantic Matching, mit dessen Hilfe festgestellt werden kann, welcher Eigenschaftswert der Ursprungs komponente an welchen Eigenschaftswert der Zielkomponente übergeben werden muss. Diese Zuordnung geschieht durch den Mediators,
welcher mit Hilfe der SMCDL eine korrekte Wertzuweisung zwischen zwei syntaktisch unterschiedlichen, aber semantisch gleichen Properties zweier UI-Komponenten durchführen kann.


Der nächste Abschnitt widmet sich dem Forschungsprojekt CoBRA und beschreibt zunächst dessen Vor- und Nachteile bezüglich der UI-Migration. Im Anschluss daran wird es anhand der Vergleichskriterien bewertet.

**CoBRA**

Nachdem der vorherige Abschnitt die derzeitige Architektur von CRUISe in Bezug auf verwendbare Migrationsfunktionalitäten untersucht hat, evaluiert dieser Abschnitt das Forschungsprojekt CoBRA mit Hilfe der in Abschnitt 4.2.1 definierten Vergleichskriterien. Zuvor werden allerdings noch die Besonderheiten dieses Forschungsprojektes in Bezug auf Migrationsfunktionalitäten zusammenfassend dargestellt.


Nach dieser zusammenfassenden Beurteilung der zwei Besonderheiten von CoBRA folgt anschließend die Bewertung mit Hilfe der Vergleichskriterien:

**(-) Nutzbarkeit des erfassbaren Zustandes:** Aufgrund der Tatsache, dass CoBRA lediglich ein allgemeines Framework für den Austausch von Komponenten bereitstellt, können demzufolge keine generellen Aussagen über die Granularität des erfassbaren Zustandes getroffen werden. Aus diesem Grund wird an dieser Stelle auf die Arbeit von Florian Irmert [Irm11] Bezug genommen, in welcher die OSGi-Plattform (eine serviceorientierte, komponentenbasierte Plattform für Java) als Grundlage für eine prototypische Realisierung diente. In die-
Modellierung des Komponentenzustandes

Konzeption der UI-Migration

sem Zusammenhang müssen alle zustandsrelevanten Attribute bzw. Klassen einer Komponente die \texttt{Serializable} Schnittstelle der \texttt{Java Serialization API} implementieren, um in serialisierter Form in dem \texttt{Memento-Objekt} gespeichert werden zu können. Demzufolge ist sowohl die Granularität des erfassbaren Zustandes, als auch die Möglichkeit die erfassten Zustandsinformationen zu kategorisieren, abhängig von der Implementierung des Komponentenentwicklers. Hinzu kommt, dass systemnahe Klassen wie beispielsweise \texttt{Thread} mittels der Java Serialization API nicht serialisiert werden können.

\(-\) \textbf{Entwicklungsaufwand für den Komponentenentwickler:} Aufgrund der Tatsache, dass der Komponentenentwickler entscheiden muss welche Attribute einer Komponente zustandsrelevant sind und welche nicht, ergibt sich die Konsequenz, dass der Aufwand die UI-Migration zu implementieren sehr hoch ist.


\(+) \textbf{Entwicklungsaufwand zur Erweiterung der Thin-Server-Runtime:} Der Implementierungsaufwand dieser Vorgehensweise kann als gering eingestuft werden, da der größte Teil der Arbeit auf den Komponentenentwickler übertragen wird. Aufgrund der Tatsache, dass die Serialisierung von Java die Objekte und Datenstrukturen nicht sperrt und andere Threads diese somit verändern könnten, müsste jedoch der Schreibzugriff isoliert werden, damit kein halbfertiger Datenstrom auf das Zielgerät übertragen wird.


Im folgenden Abschnitt wird analog zu diesem die OPEN Migration Service Platform auf deren Vor- und Nachteile untersucht und ebenfalls anhand der aufgestellten Vergleichskriterien bewertet.

\footnote{http://java.sun.com/developer/technicalArticles/Programming/serialization/}
OPEN


ne vollständige Zustandserfassung aller Web-Anwendungen nicht gewährleistet werden kann, ist die Granularität in diesem Ansatz dennoch als gut anzusehen.

(+ ) Entwicklungsaufwand für den Komponentenentwickler: Aufgrund der Tatsache, dass die OPEN-Plattform diverse OPEN-Adaptoren, welche unter anderem Migrationsfunktionalitäten zur Verfügung stellen, für den Komponentenentwickler bereitstellt, reduziert sich der Entwicklungsaufwand für den Komponentenentwickler enorm. Diese Feststellung resultiert aus der Tatsache, dass eine beliebige Komponente die erwähnten Adaptoren lediglich implementieren muss, um migrierbar zu sein.


Bevor in den folgenden Abschnitten das Gesamtkonzept vorgestellt wird, folgt eine weitere Evaluierung des Zustandsmodellierungsalgorithmus COSMOD.

COSMOD

Der in dem Forschungsprojekt COSMOD entwickelte Algorithmus ermöglicht die Erstellung eines Zustandsgraphen mit Hilfe der Beobachtung von Methodenaufrufen. Dabei kommen endliche Automaten, bestehend aus Zuständen, Transitionen / Zustandsübergängen und Aktionen (Berechnungen innerhalb eines Zustandes), zum Einsatz. Das vom Algorithmus gelieferte Ergebnis ist eine Sequenz von Methodenaufrufen, wobei jedes Element den Namen der Methode sowie die per Zufall generierten Eingabeparameter beinhaltet.

Analog zu den bereits untersuchten Forschungsprojekten, wird auch COSMOD im folgenden Abschnitt mit Hilfe der Vergleichskriterien bewertet.

Copyright TU Dresden, Christoph Pohl 52


ponente durch ein Testsystem laufen lassen, welches das Zustandsmodell gene-
eriert. Theoretisch müsste dieser Vorgang nach einer Änderung am Quellcode erneut ausgeführt werden, um die Konsistenz des Zustandsmodells zu wahren. Dies impliziert, dass der Ansatz in dieser vorliegenden Form nicht für die UI-Migration während der Laufzeit geeignet ist.


Nach dieser Evaluierung kann geschlussfolgert werden, dass der untersuchte Ansatz zur Modellierung des Zustandsmodells mit Hilfe von Methodenaufträffen nicht für den Einsatz in Bezug auf die UI-Migration im Forschungsprojekt CRUISe geeignet ist. Dies resultiert zum einen aus der sehr hohen Zeitkomplexität sowie aus der Tatsache, dass das Problem möglichst alle Zustandsdaten einer Komponente zu erfassen, nicht zufriedenstellend gelöst wurde.

**Verwendung von Event-Observern**

Bei diesem Ansatz werden alle auftretenden (UI-)Events registriert und müssen anschließend gefiltert werden, um als Ergebnis eine verwertbare Menge von zustandsrelevanten Informationen zu erhalten. Die Herausforderung liegt bei dieser Herangehensweise in der Filterung der extrahierten Informationsmenge. Analog zu den vorangegangenen Forschungsprojekten wird dieser Ansatz im nachfolgenden Abschnitt anhand des Kriterienkataloges bewertet.

(+ Nutzbarkeit des erfasbbaren Zustandes: Aufgrund der Tatsache, dass alle UI-Elemente sowie deren aktuelle Zustandswerte, welche durch den Anwen-
der modifiziert worden sind, erfasst werden, kann davon ausgegangen werden, dass alle zustandsrelevanten Informationen in dem resultierenden Zustands-
modell enthalten sind. Bei dieser Herangehensweise können die Zustandsinformationen in verschiede-
denen Formen abgespeichert werden. Wichtig für die Wiederherstellung des Komponentenzustandes sind diesbezüglich vor allem der Name bzw. die ID des jeweiligen UI-Elementes sowie dessen aktueller Wert. In diesem Zusammen-
hang wird bspw. der Inhalt eines Textfeldes oder der Zustand einer Checkbox als Wert verstanden.

(+ Entwicklungsaufwand für den Komponentenentwickler: Unter der Annahme, dass für jene Programmiersprache, welche der Komponentenentwickler für seine Implementierung gewählt hat, eine Basis-Klasse verfügbar ist, welche alle notwendigen Funktionen zur Erfassung, Extraktion sowie zur Injektion der Zustandsinformationen bereitstellt, kann der Entwicklungsaufwand für den Komponentenentwickler als gering eingestuft werden. Dies resultiert aus der Tatsache, dass der Komponentenentwickler lediglich von der abstrakten Klasse
erben und gegebenenfalls vorgegebene „Coding-Conventions“ einhalten muss. Dies könnte beispielsweise darauf hinaus laufen, dass jede ausgeführte Funktion einer Anwendung eine weitere Funktion aufrufen muss, um dieser ihren eigenen aktuellen Zustand mitzuteilen.


Diese letzte Evaluation hat gezeigt, dass die Überwachung und Registrierung von auftretenden UI-Events eine gute Basis für den Zustandserhalt von UI-Komponenten darstellt. Die Tatsache, dass sich das Kernkonzept auf andere Programmiersprachen übertragen lässt, macht diesen Ansatz sogar noch interessanter als die verwendete Technologie des Forschungsprojektes OPEN. Nachdem an dieser Stelle alle untersuchten Forschungsansätze und Technologien mit Hilfe des Kriterienkataloges evaluiert wurden, erfolgt im nächsten Abschnitt eine zusammenfassende Beschreibung der gewonnenen Erkenntnisse und die daraus resultierenden Schlussfolgerungen für die eigene Konzeption.

4.2.3 Zusammenfassung und Schlussfolgerung

Dieser Abschnitt fasst abschließend die Ergebnisse der Evaluation zusammen und erläutert im Anschluss daran die daraus resultierenden Schlussfolgerungen für das Konzept. In diesem Zusammenhang wird noch einmal auf Tabelle A.1 verwiesen, welche die positiven und negativen Eigenschaften der untersuchten Forschungsprojekte und Technologien in Bezug auf die UI-Migration im Überblick darstellt. Neben der Granularität des erfassbaren Zustandes, dem Entwicklungsaufwand für den Komponentenentwickler sowie dem Aufwand zur Erweiterung der Thin-Server-Runtime, wurden die ausgewählten Forschungsprojekte in Bezug auf deren Effizienz näher untersucht. Wie in Tabelle A.1 dargestellt, hat die Bewertung der Forschungsprojekte anhand der Vergleichskriterien ergeben, dass sowohl die Methoden der OPEN Migration Service Platform, als auch die Technologie zur Überwachung auftretender Events, für das zu entwickelnde Konzept der UI-Migration am ehesten geeignet sind. Besonders hervorzuheben sind in diesem Zusammenhang beispielsweise der Algorithmus für die Erfassung der Zustandsinformationen sowie das verwendete Zustandsmodell von OPEN. Nichtsdestotrotz muss an dieser Stelle
noch einmal erwähnt werden, dass weder Java Applets, noch Flash-Komponenten mit Hilfe der OPEN Middleware analysiert werden können.

Das Forschungsprojekt CoBRA liefert, trotz einer unzureichenden Effizienz und der Tatsache, dass die Granularität des erfassbaren Zustandes ausschließlich von dem Komponentenentwickler abhängig ist, zwei wiederverwendbare Entwurfsmuster: **Memento** und **Proxy**. Letztgenanntes Entwurfsmuster ist in CRUISe bereits implementiert, wobei im direkten Vergleich die Implementierung bzw. die Umsetzung dieses Entwurfsmusters in CRUISe besser gelungen ist, da eingehende Nachrichten während eines atomaren Prozesses nicht nur geblockt (CoBRA), sondern in einem Puffer abgespeichert werden (CRUISe), sodass diese nicht verloren gehen. Des Weiteren kommt bei dem Austausch von Komponenten in CRUISe ein 2-Phasen-Commit-Protokoll zum Einsatz, welches eine gute Grundlage für die zu konzipierende UI-Migration darstellt.

Die Zustandsmodellierung mit Hilfe der Beobachtung von Methodenaufrufen, wie sie in COSMOD durchgeführt wird, wurde aufgrund der zugrundeliegenden polynomialen Zeitkomplexität sowie dem erhöhten Entwicklungsaufwand (Implementierung eines eigenständigen Testsystems für Komponenten) schlecht bewertet und ist nicht für das Konzept dieser Forschungsarbeit relevant.

Die Registrierung aller auftretenden Events, welche in JavaScript-basierten Web-Anwendungen mit Hilfe von *Mutation Observern* realisiert werden kann, wurde ähnlich positiv bewertet wie der Ansatz von OPEN. Die Tatsache, dass dieser Ansatz auch auf andere Programmiersprachen, wie bspw. Flash, übertragen werden kann, macht diese Technologie am interessantesten für das eigene Konzept. Auf Grundlage dieser Erkenntnisse wurden folgende Konsequenzen für den weiteren Verlauf der vorliegenden Arbeit gezogen:

- **Proxy-Entwurfsmuster und 2-Phasen-Commit-Protokoll:** Sowohl das Entwurfsmuster *Proxy* als auch das *2-Phasen-Commit-Protokoll* sollen für das eigene Konzept der UI-Migration in CRUISe angewendet werden, um die involvierten Komponenten zu isolieren und sie somit vor dem Erreichen eines inkonsistenten Zustandes zu schützen. Dabei wird die Implementierung des Proxy-Patterns von CRUISe der Implementierung von CoBRA vorgezogen. Diese Entscheidung wurde aufgrund der Tatsache getroffen, dass eingehende Nachrichten, welche während des Austauschprozesses an die Komponente gesendet wurden, von dem Proxy gepuffert werden und somit nicht verloren gehen.


Auf Grundlage dieser Ansätze wird ein Konzept für CRUISe entwickelt, bei welchem die Komponentenentwicklung, in Hinblick auf die UI-Migration, vereinfacht werden soll. Ziel ist eine möglichst vollständige und automatische Erfassung aller
zustandsrelevanten Informationen, wobei der Komponentenentwickler in diesem Zusammenhang so gut wie möglich entlastet wird. Nach dieser abschließenden Zusammenfassung der Evaluationsergebnisse sowie der Beschreibung der daraus resultierenden Schlussfolgerungen, wird im nächsten Abschnitt das entwickelte Gesamtkonzept im Überblick beschrieben.

4.3 Überblick des Gesamtkonzeptes


4.3.1 Anwendungsszenario

4.3.2 Abgrenzung der Verantwortlichkeiten

Das in Abbildung 4.1 dargestellte Grobkonzept gewährt einen architektonischen Überblick und wird in Verbindung mit dem eben beschriebenen Anwendungsszenario näher erläutert. In diesem Zusammenhang werden den einzelnen Bestandteilen klar voneinander abgrenzbare Kernfunktionen zugeordnet. Eine detailliertere Beschreibung der einzelnen Teilkonzept erfolgt in Abschnitt 4.3.3.

Abbildung 4.1: Grobkonzept der UI-Migration

Wie in Abbildung 4.1 dargestellt, besteht die gesamte Migrationsinfrastruktur aus drei Teilen:

- Ausgangsgerät (Source Device)
- Server (Server Environment)
- Zielgerät (Target Device)


In Anbetracht der Tatsache, dass das im Anwendungsszenario erwähnte Smartphone keine Flash-Unterstützung bietet, befindet sich auf Serverseite (Server Environment) der Migration Manager, welcher diesbezüglich unter Zuhilfenahme des Context Service eine funktional äquivalente JavaScript-Komponente aus dem CoRe heraussucht, um die zu migrierende Flash-Komponente durch diese zu ersetzen.

Auf dem Zielgerät (Target Device) befindet sich, analog zum Ausgangsgerät, ebenfalls eine Webanwendung (Mashup Component), welche allerdings in JavaScript programmiert wurde, sowie ein State Handler. Im Gegensatz zum Ausgangsgerät, ist der State Handler des Zielgerätes für die Injektion der erhaltenen Zustandsinformationen verantwortlich. Somit ist die Konsistenz zwischen der Anwendung des Ausgangsgerätes sowie der Anwendung des Zielgerätes gewährleistet und der Anwender kann an der Stelle weiterarbeiten, an der er aufgrund der UI-Migration aufgehört
hatte. Nachdem dieser Abschnitt das Grobkonzept der Migrationsumgebung überblicksweise dargestellt hat, beschreibt der folgende Abschnitt die Funktionsweisen der einzelnen Bestandteile genauer.

### 4.3.3 Übersicht aller Teilkonzepte der UI-Migration

Dieser Abschnitt dient dazu eine Gesamtübersicht aller beteiligten Bestandteile zu geben, welche die UI-Migration auf Grundlage der CRUISe TSR ermöglichen sollen. Dabei werden die Aufgaben des entwickelten Migration Managers, des State Handlers sowie des Communication Managers, welche nachfolgend in Abbildung 4.2 dargestellt werden, überblicksweise beschrieben.

![Abbildung 4.2: Detailansicht der beteiligten Komponenten](image)

Analog zur Abbildung werden, beginnend mit dem Ausgangsgerät, die einzelnen Bestandteile der Migrationsinfrastruktur sowie deren Funktionen nacheinander benannt. Hierbei ist zu beachten, dass Teilkonzepte wie der Migration Manager sowohl server- als auch clientseitig auftauchen. Dieser Aspekt der „Verteilung“ wurde bewusst aus dem Grobkonzept herausgelassen, um die grundlegende Migrationsinfrastruktur so übersichtlich wie möglich zu halten. Im Anschluss an diese einführende Funktionsbeschreibung werden grundlegende Vorbetrachtungen diskutiert, welche die genannten Funktionen genauer beschreiben. Anschließend folgt eine detaillierte Betrachtung der entwickelten Kernkonzepte der UI-Migration in CRUISe.

### Ausgangsgerät (Source Device - Runtime)

- Auf dem Ausgangsgerät repräsentiert der Migration Manager die Benutzerschnittstelle der UI-Migration. Diese ermöglicht dem Anwender ein von ihm gewähltes Migrationssubjekt (eine UI-Komponente) sowie das Migrationsziel(gerät) auszuwählen und die UI-Migration auszulösen.

- Ein weiterer Bestandteil des clientseitigen Migration Managers ist der State Handler. Dieser ist im Kontext des Ausgangsgerätes für die Zustandsextraktion (State Extraction) sowie für die Serialisierung dieser Informationen (State Serialization) verantwortlich.
• Der clientseitige **Communication Manager** bildet die Schnittstelle zum **Migration Manager** und ist zusammen mit seinem serverseitigen Pendant für den Informations- und Datenaustausch zwischen den beteiligten Geräten verantwortlich.

**Migrationsserver (Server Environment)**

• Die Zwischenspeicherung der vom Ausgangsgerät erhaltenen Zustandsdaten ist eine Aufgabe des serverseitigen **Migration Managers (Intermediate Storage Of State)**. Diese Sicherheitskopie wird erstellt, um beispielsweise eine wiederholte Zustandsextraktion, aufgrund eines aufgetretenen Verbindungsabbruches, zu verhindern. Dies spart in dem genannten Ausnahmefall sowohl Zeit, als auch Netzwerkressourcen.

• Wie bereits im vorherigen Abschnitt beschrieben, ist der **Communication Manager** unter anderem Bestandteil des Migrationsservers. Auf Serverseite ist er für die allgemeine Kommunikationskoordinierung unter den beteiligten Geräten verantwortlich und verwaltet alle zur Verfügung stehenden Geräte (**Device Management**). Eine weitere Aufgabe des serverseitigen Communication Managers ist das **Trigger Management**, welches für den Ausführungszeitpunkt der UI-Migration verantwortlich ist.

**Zielgerät (Target Device - Runtime)**

• Auf dem Zielgerät findet man analog zum Ausgangsgerät ebenfalls einen **State Handler**, welcher allerdings die erhaltenen Zustandsinformationen deserialisiert (**State De-Serialization**) und anschließend in die Zielkomponente injiziert (**State Injection**). Zuvor bildet der **Migration Manager** jedoch die Zustandsvariablen der Ausgangskomponente auf die Variablen der Zielkomponente ab (**Mapping State Information** - siehe Abschnitt 3.1.3).

• Des Weiteren werden auf dem Zielgerät UI-Elemente von dem Migration Manager dargestellt, welche eine bevorstehende UI-Migration ankündigen und den Anwender so auf den bevorstehenden Migrationsprozess vorbereiten.

4.3.4 Vorbetrachtungen zur UI-Migration


Die Grundlage für die nachfolgenden Vorbetrachtungen bildet das in Abschnitt 4.3.1 beschriebene Anwendungsszenario, bei welchem eine Reise mit Hilfe einer Webanwendung auf einem stationären Desktop-PC zunächst geplant wird und die Webanwendung zum Zeitpunkt des Reiseantritts auf ein Smartphone migriert werden soll. Die Reihenfolge der einzelnen Teilprozesse orientiert sich an dem in Abschnitt 3.1.2 beschriebenen 2-Phasen-Commit-Protokoll von CRUISe.

1. Registrierung der beteiligten Endgeräte

Der erste Schritt der UI-Migration ist die Erfassung aller zur Verfügung stehenden Geräte, welche für den Migrationsprozess in Frage kommen können. In diesem Zusammenhang werden verschiedene Informationen jedes Gerätes, welche für den weiteren Verlauf der UI-Migration benötigt werden, wie folgt abgespeichert:

1. IP-Adresse des Endgerätes
2. Typ des Endgerätes
3. Eine von dem Communication Manager zugewiesene ID bzw. Bezeichnung des Endgerätes


a) Automatische Registrierung aller Endgeräte: Bei diesem Verfahren müsste jedes Gerät bereits bei der Anmeldung in einem Netzwerk auf die Migrationsfähigkeit untersucht werden. Aufgrund der Tatsache, dass mit Hilfe konventioneller Kommunikationsprotokolle wie TCP², UDP³ oder ARP⁴ derzeitig lediglich Informationen wie die MAC-Adresse, der Name des Endgerätes sowie der Name des Herstellers ausgelesen werden können, setzt dieses Vorgehen voraus,

---

²Transmission Control Protocol
³User Datagram Protocol
⁴Address Resolution Protocol

**b) Manuelle Registrierung jedes Gerätes bei dem CM:** Bei einer manuellen Anmeldung muss sich jedes Gerät, welches für die UI-Migration in Frage kommt, selbst bei dem Communication Manager registrieren. Demzufolge ist bei dieser Variante der Geräte- er- registrierung eine Benutzerinteraktion erforderlich, bei der zum Beispiel eine spezielle Web-Seite aufgerufen wird, welche die Metadaten des verwendeten Browsers ausliest, um Rückschlüsse auf das verwendete Endgerät zu ziehen. Aufgrund der Tatsache, dass diese UserAgent-Informationen wenig über die Spezifikationen eines Gerätes aussagen, soll in diesem Zusammenhang auf das Framework WURFL\(^7\) verwiesen werden, mit dessen Hilfe es möglich ist aus dem Header eines HTTP-Requests auf ein detaillierteres Profile des Clients zu schließen - vorausgesetzt das Gerät ist in der WURFL-Datenbank vorhanden.

Diese Untersuchungen haben gezeigt, dass sowohl die automatische, als auch die manuelle Registrierung der Endgeräte verschiedene Herausforderungen mit sich bringen. Aufgrund des Fokus der Arbeit auf die Erfassung und Modellierung des Komponentenzustandes wird zur Vereinfachung angenommen, dass lediglich zwei Endgeräte für die UI-Migration zur Verfügung stehen, welche bereits serverseitig registriert sind.


### 2. Auslösungsstrategien der UI-Migration und notwendige UI-Elemente

Wie bereits im Grundlagenkapitel 2.2.2 beschrieben, existieren zwei Möglichkeiten den Ausführungszeitpunkt der UI-Migration festzulegen, welche in diesem Teil der Vorbetrachtungen diskutiert werden:

**a) Die automatische, kontextbasierte UI-Migration:** Bei einer von dem serverseitigen Communication Manager automatisch ausgelösten UI-Migration werden externe Kontextdienste (Context Service) genutzt, um z.B. einen Ortwechsel des Nutzers festzustellen. Verlässt ein Anwender bspw. seinen Arbeitsplatz, kann diese Bewegung mit Hilfe seines Smartphones festgestellt werden.

---

\(^5\)Universal Plug and Play (http://www.upnp.org)
\(^6\)https://developer.apple.com/bonjour/
\(^7\)Wireless Universal Resource FiLe (http://wurfl.sourceforge.net)

b) Die On-Demand-UI-Migration:


3. Pausieren der UI-Komponenten

Unter der Annahme, dass mehrere UI-Komponenten auf einem Endgerät aktiv sind, ist es notwendig, diese kurz vor Beginn der UI-Migration zu pausieren. Der Grund für die Notwendigkeit dieses Schrittes wurde zwar in Abschnitt 3.1.1 bereits erläutert, soll an dieser Stelle aber noch einmal kurz aufgegriffen und anhand eines kleinen Beispiels verdeutlicht werden:

Die Grundlage dieses Beispiels bilden zwei voneinander abhängige UI-Komponenten. Dies könnte bspw. die Kartenkomponente des Ausgangsszenarios, sowie eine beliebige weitere Komponente sein. Zu einem unbestimmten Zeitpunkt leitet der Anwender die UI-Migration der gesamten Webanwendung ein und die beiden Komponenten werden *nicht* pausiert. Während der Migrationsprozess gestartet wird, arbeitet

Dieses einfache Beispiel hat gezeigt, dass das Pausieren von UI-Komponenten ein nicht zu vernachlässigender Teilaspekt der UI-Migration ist. Im weiteren Verlauf der Arbeit wird deshalb angenommen, dass die zu migrierende UI-Komponente zum Zeitpunkt des Migrationsprozesses vollständig pausiert wird und somit zum einen keine Nutzereingaben möglich sind und zum anderen alle eintreffenden Nachrichten blockiert werden. Diesbezüglich wird ein weiteres Mal auf das in CRUISe bereits implementierte 2-Phasen-Commit-Protokoll, welches in Abschnitt 3.1.2 detailliert beschrieben wurde, verwiesen.

Aufgrund der Tatsache, dass die Teilprozesse 4, 5 und 6 in den Abschnitten 4.6.1, 4.6.2 und 4.7 detailliert erörtert werden, lässt der nächste Abschnitt diese Punkte bewusst aus und beschreibt stattdessen die serverseitige Zwischenspeicherung der Zustandsinformationen.

7. Zwischenspeicherung der Informationen


Der Grund für die Zwischenspeicherung der Daten ist die Steigerung der Effizienz des Migrationsvorganges und wurde in Abschnitt 4.3.3 bereits erläutert. Der Grundgedanke besteht darin, bei einem unvorhergesehenen Verbindungsabbruch zu verhindern, dass der Migrationsprozess komplett von Beginn an noch einmal durchgeführt werden muss. Dies nimmt in dem genannten Ausnahmefall sowohl Zeit, als auch weitere Netzwerkressourcen in Anspruch. In Anlehnung an das beschriebene Anwendungsszenario scheinen die eben erwähnten Begründungen vernachlässigbar zu sein, aber bei steigender Komplexität der zu migrierenden Webanwendungen zum Beispiel nimmt dieser Teilaspekt eine entscheidendere Rolle ein.

Nach diesem Prozessschritt, muss eine UI-Komponente gefunden werden, welche den Spezifikationen des Zielgerätes genügt. In diesem Zusammenhang wird der folgende Abschnitt auf das Finden einer passenden UI-Komponente eingehen.

8. Finden einer passenden UI-Komponente

Nachdem der serverseitige MM die extrahierten Zustandsinformationen der Ausgangskomponente zwischengespeichert hat, muss zu diesem Zeitpunkt eine funktional äquivalente UI-Komponente mit Hilfe des CoRe gefunden werden. Wichtig dabei sind, neben dem Funktionsumfang der Ausgangskomponente, die Gerätespezifikatio-

Der folgende Abschnitt befasst sich mit dem Teilaspekt der Integration der UI-Komponente auf dem Zielgerät und diskutiert dabei zwei unterschiedliche Vorgehensweisen.

9. Integrieren der UI-Komponente auf dem Zielgerät

Das letzte in den Vorbetrachtungen diskutierte Teilkonzept legt den Fokus auf die Integration der zu migrierenden UI-Komponente auf dem Zielgerät. Grundsätzlich kann dabei zwischen zwei unterschiedlichen Herangehensweisen unterschieden werden, welche nachfolgend diskutiert werden sollen.

Eine der größten Herausforderung der UI-Migration besteht im Wesentlichen darin, den Zustand der Ausgangskomponente auf die Zielkomponente zu übertragen. Unter der Annahme, das die Teilprozesse 1 - 8 aus Abbildung A.2 erfolgreich durchgeführt wurden, muss an dieser Stelle entschieden werden, ob die extrahierten Zustandsinformationen bereits auf Serverseite in die Zielkomponente injiziert werden sollen oder ob die Zielkomponente zunächst auf dem Zielgerät integriert wird, um im Anschluss daran mit den entsprechenden Zustandsdaten der Ausgangskomponente aktualisiert zu werden. Wie in der Abbildung A.2 ersichtlich, wurde in der vorliegenden Arbeit der letztgenannte Ansatz gewählt. Diese Entscheidung wurde aufgrund folgender Annahmen getroffen:

- Sollte während der Injektion der Zustandsdaten ein unvorhergesehener Fehler auftreten, so kann die UI-Komponente auf dessen Standardwerte zurückgesetzt und vom Anwender trotz fehlerhafter Zustandsinjektion benutzt werden. Würde man die Zustandsdaten serverseitig in die Zielkomponente injizieren, so wäre der gesamte Migrationsprozess für den Anwender weniger transparent, da ihm keine Einblicke gegeben werden, in welchem Zustand sich die Zielkomponente befindet.

- Ein weiterer Vorteil der clientseitigen Zustandsinjektion ist die Unabhängigkeit von dem Migration Server. Denn wurden die zustandsrelevanten Daten erfolgreich auf das Zielgerät übertragen, so wird praktisch keine unbedingt notwendige Verbindung mehr zum Server benötigt.
Ein Argument was theoretisch für eine serverseitige Zustandsinjektion sprechen würde, ist die Tatsache, dass bei einer fehlerhaften Zustandsinjektion automatisch nach einer anderen UI-Komponente gesucht werden könnte, um den fehlgeschlagenen Injektionsprozess ein weiteres mal durchzuführen. Trotzdem wird im weiteren Verlauf der Arbeit an dieser Stelle festgelegt, dass bei der UI-Migration in CRUISe die ausgewählte UI-Komponente zunächst auf dem Zielgerät initialisiert und anschließend mit den extrahierten Zustandsdaten der Ausgangskomponente aktualisiert wird. An dieser Stelle wird ein weiteres Mal auf Abschnitt 4.8 verwiesen, in welchem der nächste und letzte Prozessschritt der UI-Migration - die Injektion der erfassten Daten im Kontext des Zielgerätes - ausführlich beschrieben wird.

Nachdem in den vorangegangenen Abschnitten diverse Teilkonzepte überblicksweise beschrieben wurden, fasst der nächste Abschnitt noch einmal die für die vorliegende Arbeit relevanten Entscheidungen zusammen und beschreibt somit die Ausgangslage für den weiteren Verlauf der Konzeption.

4.3.5 Ausgangslage für den weiteren Verlauf der Konzeption

Die einleitenden Vorbetrachtungen, in welchen unterschiedliche Ansätze der einzelnen Prozessschritte diskutiert wurden, sollen an dieser Stelle noch einmal kompakt zusammengefasst werden, um so die Ausgangssituation für alle nachfolgenden Teilaspekte der UI-Migration zu definieren:

1. Alle für die UI-Migration in Frage kommenden Endgeräte sind bei dem serverseitigen Communication Manager registriert.


3. Der serverseitige Migration Manager hat mit Hilfe der übermittelten Informationen eine UI-Komponente im CoRe gefunden, welche eine äquivalente Funktionalität bezüglich der Ausgangskomponente bietet und auf dem Zielgerät ausgeführt werden kann.

4. Die für die Integration notwendigen Informationen (u. a. der Komponentendescriptor) der passenden UI-Komponente sowie die extrahierten Zustandsdaten der Ausgangskomponente wurden auf das Zielgerät übertragen und der clientseitige Component Manager der TSR konnte die Komponente erfolgreich im Kontext des Zielgerätes integrieren.

4.4 Aufbau des verwendeten Zustandsmodell des Modellierungs- und Konzeptionsservices


4.4.1 Kategorisierung von Zustandsinformationen

Die Erhaltung des Komponentenzustandes ist einer der wichtigsten Aspekte der UI-Migration (vgl. Abschnitt 2.2.1). In Kapitel 3 (Stand der Forschung und Technik) wurden in diesem Zusammenhang mehrere Ansätze zur Erhaltung der Zustandsinformationen vorgestellt. Die Evaluation dieser Forschungsprojekte in Abschnitt 4.2 hat gezeigt, dass die Serialisierung des DOM-Baums, kombiniert mit der Erfassung vorkommender JavaScript-Variablen, die effektivste Methode zur Erfassung von Zustandsinformationen ist. Dieser Ansatz der OPEN Migration Service Platform wurde für das eigene Konzept aufgegriffen und um den Aspekt der Zustandsdatenkategorisierung erweitert. Die Intention hinter dieser Kategorisierung ist die ausschließliche Erfassung aller relevanten Zustandsinformationen. Was damit gemeint ist wird deutlich, wenn man sich das in 4.3.1 beschriebene Anwendungsszenario noch einmal ins Gedächtnis ruft. In diesem Zusammenhang müsste die Frage geklärt werden, ob eine vollständige Serialisierung des gesamten DOM-Baums wirklich notwendig ist, um den Zustand der Ausgangskomponente des Desktop-PCs auf die Zielkomponente des Smartphones zu übertragen, oder ob es genügt nur einen Teil aller Zustandsdaten für die UI-Migration zu erfassen. Bevor die Antwort auf diese Frage erörtert wird, definiert die folgende Übersicht zunächst mögliche Zustandskategorien, um einen Überblick zu schaffen, welche verschiedenartigen Zustandsdaten existieren und berücksichtigt werden müssen:


Verhaltenszustand Der Verhaltenszustand ist gekoppelt mit dem UI-Zustand und beinhaltet bspw. den Zustand beziehungsweise den Wert eines laufenden Timers, welcher durch die Bestätigung eines Buttons aktiviert wurde.

Style Die Styleinformationen beschreiben das Erscheinungsbild einer UI-Komponente. Dazu zählen grundlegende Informationen wie Höhe, Breite und Farbgebung der Komponente.

Layout Layoutinformationen beschreiben die Position der Komponente in Bezug auf die Web-Anwendung sowie die Anordnung einzelner Elemente innerhalb der Komponente.

Abbildung 4.3: Informationen über die zu übertragenden Zustandsinformationen je Anwendungsszenario

Wie in Tabelle 4.3 dargestellt, werden in Migrationsszenarien, bei denen ausschließlich *homogene* Endgeräte beteiligt sind, alle *Zustandsinformationen* migriert. In diesem Zusammenhang bedeutet „homogen“, dass sowohl der Gerätetyp, als auch das verwendete Betriebssystem bzw. die ausgeführte Laufzeitumgebung identisch sein muss, um eine fehlerfreie UI-Migration zu ermöglichen. Der Vorteil dieser Art von UI-Migration ist der Aspekt der Wiedererkennung, da die UI-Komponente auf dem Zielgerät visuell identisch zu der Ausgangskomponente erscheint.

Im Gegensatz dazu, werden bei der „heterogenen“ Migration (Desktop-PC → Smartphone) ausschließlich der *Zustand der Wertemenge* sowie der *Verhaltenszustand* übernommen. Dies resultiert aus der Tatsache, dass Anwendungen auf Smartphones, aufgrund des kleineren Displays zum Beispiel, in den häufigsten Fällen eine vollkommen andere Benutzerschnittstelle bereitstellen, als deren Pendants, welche auf Laptops oder PCs ausgeführt werden. Diese Tatsache lässt sich auch auf die anderen Migrationsszenarien übertragen, bei denen *heterogene* Endgeräte beteiligt sind.


Dieser Abschnitt hat überblicksweise beschrieben, in welche Kategorien sich die Zustandsdaten von UI-Komponenten einteilen lassen. Dabei wurde die Funktionalität jeder einzelnen Kategorie definiert und im Anschluss daran wurde erörtert, warum es oft nicht notwendig ist alle theoretisch erfassbaren Zustandsinformationen zu migrieren. Bevor in Abschnitt 4.6 die konzeptionierte Technik zur *Extraktion der Zustandsdaten* erläutert wird, beschreibt der folgende Abschnitt überblicksweisen...
se die Zusammenhänge zwischen bestehenden und neu konzeptionierten CRUISe-Komponenten.

4.5 Aufgabenverteilung der beteiligten Komponenten

Nachdem im vorherigen Abschnitt Zustandskategorien definiert wurden, mit denen eine Unterscheidung aller möglichen Komponentenzustände ermöglicht wird, präsentiert dieser Abschnitt die Zusammenhänge zwischen bestehenden und neu konzeptionierten CRUISe-Komponenten. Mit Hilfe der Abbildung 4.4 werden nachfolgend die Aufgaben jeder Teilkomponente zunächst überblicksweise beschrieben und im Anschluss daran folgt eine detaillierte Betrachtung jedes Teilspecktes.

![Diagramm der CRUISe-Komponenten](image.png)

Abbildung 4.4: Zusammenhänge der beteiligten CRUISe-Komponenten

bzw. Plattform einmalig implementiert werden. Das nachfolgende Kapitel 5 präsentiert in diesem Zusammenhang unter anderem die Umsetzung der beiden erwähnten Teilkomponenten in der Programmiersprache JavaScript.


Dieser Abschnitt hat überblicksweise die Zusammenhänge bestehender und neu konzeptionierter CRUISe-Komponenten gegeben. Darauf aufbauend wird in Abschnitt 4.6.2 die konzeptionierte Technik zur Extraktion der Zustandsdaten detailliert präsentiert. Dabei wird zunächst noch einmal kurz die Designentscheidung erläutert und daran anschließend werden die einzelnen Kernaspekte näher betrachtet.

### 4.6 Extraktion der Zustandsinformationen

Nachdem im vorherigen Abschnitt die Zusammenhänge der beteiligten CRUISe-Komponenten überblicksweise beschrieben wurden, fokussiert sich dieser Abschnitt auf die clientseitige Extraktion der Zustandsinformationen. In diesem Zusammenhang wird einleitend noch einmal die getroffene Designentscheidung erläutert und im Anschluss daran folgt die Präsentation der Kernaspekte des konzeptionierten Ansatzes.

#### 4.6.1 Wahl einer Zustandsextraktnotionsmethode

Nach der Vorstellung mehrerer Forschungsprojekte und aktueller Techniken zur Zustandsextraktion in Kapitel 3, folgte zu Beginn dieses Kapitels die Evaluation dieser Ansätze auf Grundlage des zuvor definierten Kriterienkataloges. Dabei stellte sich heraus, dass sowohl die Herangehensweise der OPEN Migration Service Platform, als auch die Technologie der Event-Überwachung für die UI-Migration im Forschungsprojekt CRUISe nützlich sind.

- Bei OPEN wird der komplette DOM-Baum serialisiert, um somit zu gewährleisten, dass alle Zustandsdaten (inklusiv Style- und Layoutinformationen) migriert werden. Aufgrund der Tatsache, dass häufig nicht alle Zustandsinformationen genutzt werden können (siehe Abschnitt 4.4.1), besteht eine Herausforderung dieser Herangehensweise demzufolge darin, die extrahierten Daten je nach Migrationsszenario entsprechend zu filtern. Wie in dem Beispielszenario aus Abschnitt 4.3.1 beschrieben, werden die Style- und Layoutinformationen bei der UI-Migration von einem Desktop-PC auf ein Smartphone beispielsweise nicht mit übertragen.
Ein weiteres Defizit dieses Ansatzes, welches bereits in Abschnitt 4.2.2 beschrieben und begründet wurde, ist die fehlende Unterstützung von Flash-basierten UI-Komponenten oder Java Applets, welche nicht analysiert werden und im Zielkontext entweder ausgetauscht oder weggelassen werden.


Die Tatsache, dass in Anbetracht des Einsatz des Black-Box-Paradigma nicht davon ausgegangen werden kann, dass jede UI-Komponente eine DOM-Baum-ähnliche Struktur aufweist und sich die zweite Methode auf weitere Programmiersprachen übertragen lässt, führte zu der Entscheidung das nachfolgend vorgestellte Konzept auf Basis dieses Ansatzes zu entwickeln. Diesbezüglich wird im nachfolgenden Abschnitt die prinzipielle Vorgehensweise der Zustandsextraktion beschrieben. Im Anschluss daran wird in den Abschnitten 4.7 und 4.8 der Transport der extrahierten Zustandsdaten zum Zielkontext sowie die anschließende Injektion der Informationen präsentiert.

4.6.2 Mutation-Event-Detection mit Hilfe des State Handlers


Um einen Einblick zu bekommen, welche Teilprozesse für die Extraktion der zustandsrelevanten Informationen notwendig sind, wurden die Prozessschritte in einer Grafik (Abbildung A.3) visualisiert, welche im Folgenden detailliert erläutert wird:

1. **Instanziierung eines **\textit{UIStateObservers} **Bevor die erste Zustandsänderung erfasst werden kann, instanziert der Component Manager, während der Initialisierung der Web-Anwendung \((t_0 \text{ bis } t_1)\), für jede UI-Komponente einen UIStateObserver. Wichtig dabei ist, dass dieser Prozess vor der ersten Interaktion des Nutzers abgeschlossen ist, sodass keine Zustandsinformationen verloren gehen. Ein weiterer Bestandteil der Migrationsinfrastruktur ist der State Handler, welcher ebenfalls während der Initialisierung der Web-Anwendung von der Laufzeitumgebung instanziert wird. Dabei ist zu beachten, dass der eben erwähnte State Handler Teil der TSR ist und der UIStateObserver innerhalb der Komponente agiert. Warum diese Unterscheidung so wichtig ist, wird im nächsten Teilabschnitt verdeutlicht.


3. **Serialisierung und Pufferung aller Mutation-Events** Nachdem der UIStateObserver ein UI-Event oder zyklischen Funktionsaufruf erfasst, kategorisiert und als zustandsrelevanten Mutation-Event definiert hat, muss dieses Event in dem UIStateBuffer zwischengespeichert werden. Wie in Abbildung
A.3 dargestellt, übernimmt diese Aufgabe der State Handler. In diesem Zusammenhang wird das Mutation-Event zunächst mit einem technologie- und plattformunabhängigen Datenaustauschformat innerhalb der UI-Komponente serialisiert. Das Ergebnis dieser Serialisierung enthält dabei die folgenden Informationen:

a) **modifiziertes UI-Element**
   - Name des Objektes
   - aktueller Wert

b) **zyklische Funktionen**
   - Funktionsaufruf
   - mögliche Parameter des Methodenaufrufes
   - Länge des Zeitintervalls eines Timers


Im Anschluss an die Serialisierung werden die Daten an den State Handler übergeben, welcher für die Organisation des im Browser-Cache befindlichen Puffers verantwortlich ist. Alle eintreffenden Informationen werden mit den bereits gepufferten Daten abgeglichen und bei Vorhandensein einer älteren Version wird diese verworfen und der aktuelle Datensatz zwischengespeichert. Demzufolge können im Zielfall eventuell implementierte „UNDO-Funktionalitäten“ direkt nach der UI-Migration nicht genutzt werden.


Nachdem dieser Abschnitt ausführlich beschrieben hat, wie der Zustand einer UI-Komponente erfasst und für die anstehende UI-Migration extrahiert wird, befasst sich der nachfolgende Abschnitt 4.7 mit der Übertragung der Zustandsdaten zum Zielgerät. In diesem Zusammenhang soll vor allem der **Aufbau des verwendeten Kommunikationsprotokolls** beschrieben werden und wie die eigentliche Kommunikation zwischen den einzelnen Komponenten realisiert wird.

4.7 Transfer der Zustandsdaten


4.7.1 Einleitung der Transferphase

Bevor die erfassten Daten an das Zielgerät gesendet werden können, muss der Transfer der Zustandsinformationen von dem Communication Manager entsprechend vorbereitet werden. In diesem Zusammenhang werden die in Abschnitt 4.6.2 Punkt 3 (Serialisierung und Pufferung) genannten Daten um zusätzliche, migrationsspezifische Informationen erweitert. Die nachfolgende Übersicht gibt diesbezüglich einen Einblick über alle Informationen, die an den serverseitigen Migration Manager gesendet werden:
• Informationen der Ausgangskomponente
  – ID der UI-Komponente

• Informationen zum Zielgerät
  – ID des Zielgerätes

• Paket mit Zustandsdaten
  – modifizierte UI-Elemente: Bezeichnung und letzter, aktueller Wert
  – Zustandsdaten aktiver Intervall-Timer


4.7.2 Ausgangsgerät → Migration Server → Zielgerät


An dieser Stelle wird noch einmal hervorgehoben, dass ausschließlich IDs einzelner

Copyright TU Dresden, Christoph Pohl 75

Wurde eine passende UI-Komponente gefunden, kann die Übertragung der Zustandsdaten sowie aller benötigten Informationen zur Integration der Komponente im Kontext des Zielgerätes beginnen. Analog zur ersten Phase der Datenübertragung, übernimmt der Communication Manager diese Aufgabe, welcher zunächst die konkrete Netzwerkadresse des Ziels, anhand der mitgesendeten Geräte-ID, ermittelt.

Im Anschluss daran beginnt die Übertragung des Datenstroms, welcher in diesem Zusammenhang die folgenden Informationen beinhaltet:

- **Komponentendescriptor** und **ID** der zu integrierenden UI-Komponente
- **Schlüsselwort UI-Migration**
- **Paket der Zustandsdaten**

Nachdem der serverseitige Migration Manager eine funktional äquivalente UI-Komponente im CoRe gefunden hat, übergibt er deren Komponentendescriptor, die ID der Zielkomponente sowie die bereits zwischengespeicherten Zustandsinformationen wieder dem (serverseitigen) Communication Manager. Zusätzlich wird ein Schlüsselwort hinzugefügt, um dem clientseitigen Component Manager zu signalisieren, dass es sich um eine UI-Migration handelt, sodass nach Abschluss der Integration die UI-Komponente sofort blockiert wird, um die Zustandsinjektion zu ermöglichen (siehe Abschnitt 4.8). Der zu versendende Datenstrom wird vom serverseitigen CM über sein clientseitiges Pendant hin zum clientseitigen Migration Manager übertragen, welcher den weiteren Verlauf des UI-Migrationsprozesses koordiniert.


Einzelheiten dieser Vorgehensweise werden von dem nachfolgenden Abschnitt 4.8 präsentiert.

### 4.8 Wiederherstellung des UI-Komponentenzustandes

Im letzten Prozessschritt der UI-Migration müssen die Zustandsdaten, welche zuerst aus der Ausgangskomponente extrahiert, anschließend serialisiert und danach über den serverseitigen Migration Manager bis hin zum Endgerät des Zielkontextes transferiert wurden, in die Zielkomponente injiziert werden. Mit dieser Herausforderung beschäftigt sich der aktuelle Abschnitt. In diesem Zusammenhang werden einleitend noch einmal die zur Verfügung stehenden Zustandsinformationen präsentiert.
und im Anschluss daran wird der eigentliche Ablauf der Zustandsinjektion detailliert erläutert.

4.8.1 Die verfügbaren Zustandsinformationen

Bevor der eigentliche Ansatz präsentiert wird, sollen noch einmal überblicksweise die zur Verfügung stehenden Zustandsinformationen zusammengetragen werden, um ein Verständnis für die Voraussetzungen der anstehenden Zustandsinjektion zu vermitteln:

- ID der zu aktualisierenden UI-Komponente
- Zustand der Wertemenge
  - Name des modifizierten UI-Elementes
  - aktueller Wert dieses Objektes
- Intervall-Timer
  - Methodenauftrag
  - Parameter der aufzurufenden Funktion
  - Länge der Zeitspanne des Intervall-Timers

Wie bereits im vorherigen Abschnitt kurz angedeutet, übergibt der clientseitige MM die ID der zu aktualisierenden UI-Komponente dem State Handler, welcher für die eigentliche Zustandsinjektion verantwortlich ist. Somit ist gewährleistet, dass die Zustandsdaten an der richtigen Stelle injiziert werden, sobald die Web-Anwendung aus zwei oder mehr UI-Komponenten besteht.

4.8.2 Ablauf der Zustandsinjektion

Analog zum Abschnitt 4.6.2 werden die nachfolgenden Ausführungen auf Grundlage einer graphischen Darstellung (Abbildung A.5) präsentiert und erörtert:


2. **Vorbereitung der Zustandsinjektion** Ein essentieller Bestandteil für eine erfolgreiche Zustandswiederherstellung ist der UISateInjector. Dieser wird, ähnlich wie der UISateObserver, von dem Component Manager für jede UI-Komponente initialisiert und übernimmt alle kommenden Aufgaben bezüglich der Zustandsinjektion. Nachdem die Benutzeroberfläche gesperrt und die
**Modellierung des Komponentenzustandes** 4 Konzeption der UI-Migration


3a. **Injektion von Zustandsdaten in UI-Elemente**
Hat der **UIStateInjector** den vom **State Handler** übergebenen Puffereintrag als ein Wertezustand identifiziert, wird anhand des Namens ein UI-Element in der Komponente mit über- einstimmender Bezeichnung gesucht und eine der folgenden Anweisungen ausgeführt:


- **b)** Wird kein entsprechendes UI-Element in der Zielkomponente gefunden, melde diese Information dem **State Handler** und warte auf den nächsten Eintrag aus dem **UIStateBuffer**.

Sollte kein entsprechendes UI-Element in dem Kontext der Zielkomponente gefunden werden, wird dieser Eintrag im **UIStateBuffer** markiert, um dem Anwender am Ende der UI-Migration mitteilen zu können, welche Zustandsdaten nicht migriert werden konnten. Andernfalls wird der Eintrag vom **State Handler** aus dem Puffer gelöscht und die nächsten Zustandsinformationen werden an den **UIStateInjector** übergeben. Sobald der letzte Eintrag in der Zielkomponente verarbeitet wurde, geht der **State Handler** zu Prozessschritt 4 über.

3b. **Wiederherstellung des Zustandes eines Intervall-Timers**
Eine detaillierte Beschreibung der genauen Umsetzung dieser Teilkonzepte folgt im folgenden Kapitel 5.


4.9 Zusammenfassung

Dieses Kapitel beinhaltete die Konzeption einer Migrationsinfrastruktur für das Forschungsprojekt CRUISe. Dazu wurde zunächst eine Anforderungsanalyse durchgeführt, bei welcher funktionale sowie nicht-funktionale Anforderungen aufgestellt wurden. Im Anschluss daran wurden die in Kapitel 3 untersuchten Ansätze bewertet, um verwertbare Technologien herauszufiltern, welche in dem eigenen Konzept wiederverwendet werden können. Ein Kriterienkatalog, welcher auf Grundlage der eingangs definierten Anforderungen spezifiziert wurde, bildete dabei die Basis dieser Evaluation.

In Abschnitt 4.3 wurde ein Überblick über die konzeptionierte Migrationsarchitektur gegeben. Die Beschreibung eines Anwendungsszenarios sowie die anschließende Vorstellung der einzelnen Teilkonzepte präsentierte einen architektonischen Einblick in die Migrationsumgebung. Abschnitt 4.3.4 grenze den Fokus der vorliegenden Arbeit weiter ein und beschrieb in diesem Zusammenhang wichtige Voraussetzungen für den UI-Migrationsprozess, wie beispielsweise die Erfassung der beteiligten Endgeräte. Nach dieser Eingrenzung und Beschreibung wichtiger Teilaspekte, wurden die Ergebnisse in 4.3.5 noch einmal zusammengefasst und dabei die Ausgangslage für den weiteren Verlauf der Konzeption definiert.

In Abschnitt 4.4 wurde der Aufbau des verwendeten Zustandsmodells beschrieben und dabei auf den Aspekt der Kategorisierung von Zustandsinformatio-
eingegangen. Es wurde festgelegt, dass im Gegensatz zu den Style- und Layout-
informationen einer Komponente, ausschließlich der Zustand der Wertemenge sowie
Zustände aktiver Intervall-Timer für das Konzept dieser Arbeit von Bedeutung sind.
Im Fokus des darauffolgenden Abschnittes stand die Extraktion der Zustandsin-
formationen. In diesem Zusammenhang wurde das Mutation-Event-Detection-
Verfahren vorgestellt, welches das Ergebnis einer vorangegangenen Diskussion, be-
züglich der Wahl einer geeignete Extraktionsmethode, darstellte.
Der nächste Abschnitt erläuterte den Transfer der extrahierten Zustandsda-
ten. Es wurde beschrieben, über welchen Weg die Zustandsinformationen von dem
Ausgangsgerät zu dem Zielgerät gelangen und wie die Kommunikation zwischen
den beteiligten UI-Migration-Managern funktioniert.
Abschnitt 4.8 fokussierte den letzten Prozessschritt der UI-Migration - die Zustand-
sinjektion - wobei ähnlich wie in dem Abschnitt der Zustandsextraktion, zunächst
noch einmal die gegebenen Voraussetzungen beschrieben wurden, bevor unter Zu-
hilfenahme einer graphischen Prozessvisualisierung die eigentliche Vorgehensweise
detailliert erörtert wurde.
Im nächsten Kapitel folgt die Vorstellung der prototypischen Implementierung. In
diesem Zusammenhang stehen sowohl Sender- als auch Empfängerlogik im Fokus
der Betrachtungen. Des Weiteren wird anhand einer Beispielanwendung der Proto-
typ und somit das Konzept validiert.
5 Implementation


5.1 Das verwendete Implementierungsframework


5.2 Erweiterung der CRUISe-Thin-Server/Runtime

Dieser Abschnitt gibt einen Überblick über die implementierten Erweiterungen in der Laufzeitumgebung von CRUISe. Dabei wird vorausgesetzt, dass die folgenden Quelldateien während der Initialisierung einer Web-Anwendung mit einbezogen werden:

- StateHandler.js
- CommunicationManager.js
- MigrationManager.js

Abbildung 5.1 visualisiert die Zusammenhänge zwischen allen CRUISe-Manager, wobei aufgrund der Übersichtlichkeit nicht alle Verbindungen zwischen den einzelnen Teilkomponenten der TSR eingezeichnet wurden [Pie12]. Die hervorgehobenen Objekte stellen in diesem Zusammenhang die konzeptionierten Erweiterungen der TSR dar, welche nachfolgend näher beschrieben werden.

Wie in der Grafik dargestellt, besteht eine Verbindung zwischen dem Migration Manager und der kompositen Web-Anwendung. Dies resultiert aus der Tatsache, dass der MM Benutzerschnittstellen zur Konfiguration der UI-Migration bereitstellt, welche unabhängig von jeder UI-Komponente existieren. Im Gegensatz dazu besitzen sowohl der UIStateObserver als auch der UIStateInjector eine direkte Verbindung

\[1\text{http://www.sencha.com/products/extjs}\]

5.2.1 Anpassungen innerhalb des Application Managers

Nachdem im vorherigen Abschnitt überblicksweise beschrieben wurde, wie die Infrastruktur von CRUISe erweitert wurde, gibt dieser Abschnitt einen Einblick in die vorgenommenen Anpassungen innerhalb des Application Managers:

Erweiterung des Konstruktors Um die UI-Migration einer Web-Anwendung ohne große Änderungen am Quellcode zu aktivieren bzw. zu deaktivieren wurde der Konstruktor des Application Managers um folgenden Code erweitert:

```
constructor: function(config){
Ext.apply(this, config, {

  useMigration: false
});
```

Listing 5.1: Erweiterung des Konstruktors von dem Application Manager
Wie in dem Codebeispiel dargestellt ist die UI-Migration standardmäßig deaktiviert, kann aber durch durch `useMigration: true` in der `index.html` aktiviert werden.

**Instantiierung der notwendigen Migrations-Komponenten** Die nachfolgenden Codezeilen instantiieren den State Handler, den Communication Manager sowie den Migration Manager:

```javascript
if (this.useMigration) {
    this.stateHandler = new Ext.cruise.client.StateHandler(this.eventHandler, this.log, this.applicationContext);
    this.communicationManager = new Ext.cruise.client.CommunicationManager(this.eventHandler, this.log, this.applicationContext);
    this.migrationManager = new Ext.cruise.client.MigrationManager(
        this.eventBroker, this.log, this.applicationContext, this.stateHandler, this.communicationManager);
}
```

Listing 5.2: Instantiierung der notwendigen Migrations-Komponenten

**Registrierung eines System-Kanals** Um alle auftretenden Zustandsübergänge der Anwendungskomponente, welche durch Events signalisiert werden, zu erfassen, wird bei der Laufzeitumgebung ein System-Kanal registriert:

```javascript
// register the specified system channels
this.registerSystemChannels([{
    id: 'errorChannel', type: 'object'},
{id: 'codeReceivedChannel', type: 'object'},
{id: 'stateChangeReceivedChannel', type: 'object'},
{id: 'runtimeChannel', type: 'object'},
{id: 'componentLCChannel', type: 'object'},
{id: 'adaptabilityChannel', type: 'string'}]);
```

Listing 5.3: Registrierung eines System-Kanals

**Hinzufügen eines Events zum Kanal** Diese Erweiterung dient dazu alle auftretenden Events mit der Bezeichnung „stateEventInjJSONformat“ dem zuvor definierten Kanal zuzuordnen:

```javascript
if (this.useMigration) this.eventBroker.addEventToChannel('stateEventInjJSONformat', 'object', 'stateChangeReceivedChannel');
```

Listing 5.4: Hinzufügen eines Events zum Kanal

**Registrierung eines Event-Handlers** Jedem registrierten Kanal wird ein Event-Handler zur Verfügung gestellt, welcher die Verarbeitung aller auftretenden Events (dieses Kanals) übernimmt:

5.3 Senderlogik: Zustandserfassung und -extraktion

Wie bereits in Kapitel 4, Abschnitt 4.6.2 angedeutet, übernimmt der UIStateObserver die Erfassung, Kategorisierung sowie Extraktion aller zustandsrelevanten Events. Eine Anforderung, welche zu Beginn des Konzeptions-Kapitels definiert wurde, war die Reduzierung des Entwicklungsaufwandes bezüglich der UI-Migration. Um dieser Anforderung gerecht zu werden, wurde die Entscheidung getroffen alle Migrationsfunktionalitäten über programmiersprachenspezifische Basisklassen bereitzustellen. In diesem Zusammenhang wird nachfolgend das Klassendiagramm (Abbildung 5.2) der Basisklasse von JavaScript-basierten UI-Komponenten präsentiert und anschließend erläutert:

Abbildung 5.2: Klassendiagramm der JavaScript-basierten Basisklasse

Um jegliche Migrationsfunktionalitäten zu erhalten, müssen alle UI-Komponenten
5.3.1 DOM-Mutation-Observer in JavaScript

Das nachfolgende Codebeispiel dient als Grundlage der Dokumentationen dieses Abschnitts, wobei in diesem Zusammenhang vereinzelt auf bestimmte Zeilen des Quellcodes Bezug genommen, indem am Ende des Satzes in Klammern die Zeilennummer des zugehörigen Quellcode-Ausschnittes notiert wird:

```javascript
// create an UIStateObserver instance
var UIStateObserver = new MutationObserver(function(mutations) {
    mutations.forEach(function(mutationEvent) {
        if (mutationEvent.target.type) {
            // create MutationEventObject
            var stateEventObject = createMutationEvent(mutationEvent);

            // transform object to JSONString
            var JSONstateEvent = JSON.stringify(stateEventObject);

            // publish StateEvent to runtime
            publishStateChangeEvent(JSONstateEvent);
        }
    });
});

// select the target node
var target = document;

// configuration of the observer:
var config = {
    subtree: true,
    attributes: true,
    characterData: true,
};

// pass in the target node, as well as the observer options
UIStateObserver.observe(target, config);
```

Listing 5.6: Instantiierung eines DOM-Mutation-Observers in JavaScript

Um Zugriff auf den DOM-Baum zu erhalten und alle auftretenden Strukturveränderungen innerhalb des DOM-Baums registrieren zu können, wird im ersten Schritt eine Instanz eines MutationObservers generiert (2). Anschließend wird der Zielknoten festgelegt (20), welcher bezüglich der eben beschriebenen Events überwacht werden soll. Des Weiteren müssen diverse Konfigurationsparameter (23 bis 27), wie bspw. die Einbeziehung aller Kindknoten (24), definiert werden. Im letzten Schritt wird
eine Methode des instantiierten MutationObservers aufgerufen, welche den Zielknoten sowie die Konfigurationparameter beinhaltet und die Überwachung auftretender Strukturveränderungen startet (30). In den Zeilen 7 bis 13 werden zusätzliche Funktionen aufgerufen, um bspw. zustandsrelevante Informationen aus den Mutation-Events zu extrahieren oder die Daten zu serialisieren. Diese Methoden stehen im Fokus der nachfolgenden Abschnitte.

Erstellung eines stateObjects sowie anschließende Serialisierung

Nachdem eine Instanz des UIStateObservers erstellt wurde, müssen alle signalisierten Zustandsveränderungen auf deren Relevanz untersucht und entsprechend aus sortiert werden. Dies wird durch Aufruf der Methode createMutationEvent(mutation) (7) realisiert:

```javascript
function createMutationEvent(mutationEvent) {
  // create an empty object
  var stateObject = new Object();

  // fill object
  stateObject['type'] = mutationEvent.target.type;
  stateObject['name'] = mutationEvent.target.name;
  stateObject['id'] = mutationEvent.target.id;
  stateObject['value'] = mutationEvent.target.value;

  // return object
  return stateObject;
}
```

Listing 5.7: Kategorisierung der signalisierten Zustandsveränderungen

Die Funktion bekommt als Eingabe ein mutationEvent und wandelt dieses zunächst in ein stateEventObject um (3). Anschließend extrahiert die Methode die Attribute „type“ (6), „name“ (7), „id“ (8) und „value“ (9), welche essentiell für den Migrationsprozess sind, da mit Hilfe dieser Informationen eine eindeutige Zuordnung auf Empfängerseite sichergestellt werden kann. Das Ergebnis dieses Methodenaufrufes ist demzufolge ein Objekt, welches die eben beschriebenen Attribute sowie die dazugehörigen Werte beinhaltet (12).

Nach Erstellung des stateEventObjects wird dieses durch Aufrufen der Methode JSON.stringify(stateEventObject) mit Hilfe des plattform- und technologieneutralen Datenaustauschformates JSON serialisiert.

Bekanntmachung einer Zustandsveränderung

Aufgrund der Tatsache, dass in CRUISe das Black-Box-Paradigma Anwendung findet, wird es der Laufzeitumgebung nicht ermöglicht in eine UI-Komponente hineinzuschauen, um zum Beispiel ihren Zustand zu extrahieren. Aus diesem Grund wurde entschieden, dass die Komponenten selbst ein Event publizieren, sobald sich deren Zustand verändert. Dies wird durch die Methode publishStateChangeEvent(jSONstateEvent) (13) realisiert:

```javascript
function publishStateChangeEvent(stateEvent_jSON) {
  var message = new Ext.cruise.client.Message();
  message.setName('stateEventInjSONformat');
}
```
Wie in dem Codebeispiel dargestellt wird zunächst ein `CRUISe-Message-Objekt` erzeugt (2) und mit einer Bezeichnung versehen (3). Im Anschluss daran wird der zuvor erzeugte jSON-String in die Nachricht gepackt (4) und abschließend an den `Event Broker` (siehe Abbildung 5.1) übergeben.

Dieser Abschnitt hat einen Einblick gegeben, wie der in Kapitel 4 konzeptionierte Ansatz der Zustandserfassung und -extraktion programmierotechnisch umgesetzt wurde. Analog dazu wird im nächsten Abschnitt 5.4 die Injektion der Zustandsdaten bei JavaScript-basierten UI-Komponenten erläutert.

### 5.4 Empfängerlogik: Zustandswiederherstellung

Nachdem der vorherige Abschnitt die Umsetzung des Konzeptes der Zustandserfassung und -extraktion beschrieben hat, steht in diesem Abschnitt die Injektion der Zustandsdaten im Fokus der Betrachtungen. An dieser Stelle wird noch einmal auf Abbildung 5.2 verwiesen, da in dem bereits vorgestellten Klassendiagramm des `UIStateObservers` die zuletzt aufgeführte Methode `injectState(stateBuffer)` für die Injektion der Zustandsdaten verantwortlich ist.

#### 5.4.1 Erweiterung der Komponentenschnittstelle


```javascript
injectStateToComponent: function (stateBuffer) {
    for (var i=0; i<stateBuffer.length; i++) {
        if (stateBuffer[i]["id"]) {
            var id = stateBuffer[i]["id"];
            var value = stateBuffer[i]["value"];
            var selectedDOMelement = document.getElementById(id);
            selectedDOMelement.value = value;
        } else { alert("Unable to locate position in DOM – StateBufferEntry doesn’t have an ID.") }
    }
}
```

Listing 5.9: Funktions zur Injektion des Komponentenzustandes
Für jeden Eintrag im Zustandsdatenpuffer mache folgendes:

Überprüfe, ob der aktuelle Eintrag das Attribut „id“ enthält.

a) Wenn ja, dann mache folgendes:
   1. Erstelle eine Variable „id“ und speichere darin den Wert des Attributes „id“ aus dem aktuellen Puffereintrag.
   2. Erstelle eine Variable „value“ und speichere darin den Wert des Attributes „value“ aus dem aktuellen Puffereintrag.
   3. Durchsuche den kompletten DOM-Baum der Web-Anwendung nach einem Element mit der abgespeicherten „id“.
   4. Ersetze den Wert des gefundenen Elementes mit dem Wert der Variable „value“.

b) Ist das Attribut „id“ nicht existent, dann gebe eine Fehlermeldung aus.


5.5 Beispielanwendung

Nachdem die Funktionalität des umgesetzten Konzeptes erörtert wurde, folgt in diesem Abschnitt die Validierung des entwickelten Konzeptes. In diesem Zusammenhang wurde in Anlehnung an das erdachte Anwendungsszenario des Konzeptionskapitels (Abschnitt 4.3.1 eine JavaScript-basierte Beispielanwendung entworfen. Dabei handelt es sich um einen Reiseplaner, bei dem sich der Anwender eine Reiseroute generieren lassen kann. Des Weiteren kann der Nutzer bei jeder Station seiner Reise Notizen vermerken. Grundlage für die verwendeten Kartendaten ist die Google Maps JavaScript API Version 3\(^2\).

Abbildung 5.3 zeigt die eben beschriebene Beispielanwendung sowie das Layout des Migration Managers. Wie in dem Screenshot zu erkennen ist, lässt sich das User-Interface des Migration Managers je nach Bedarf ein- bzw. ausblenden. Neben einer kurzen Funktionsbeschreibung sind drei Buttons sowie zwei Auswahlmenüs zu finden, welche nachfolgend erläutert werden:

- **Show state information:** Dieser Button gibt nach Benutzung den derzeitigen Inhalt des UIStateBuffers in der Konsole des Browser aus.

- **Select a component:** Das erste Auswahlmenü ermöglicht dem Anwender eine UI-Komponente zu selektieren, welche migriert werden soll.

---

\(^2\)https://developers.google.com/maps/documentation/javascript/?hl=de
• **Select a target device:** Analog zur vorherigen Beschreibung, kann der Nutzer mit Hilfe dieses Auswahlmenüs das gewünschte Zielgerät wählen.

• **Start UI-Migration:** Dieser Button leitet den Migrationsprozess mit den entsprechenden Konfigurationsparametern (UI-Komponente, Zielgerät) ein.

• **Inject state information:** Bei Benutzung dieses Buttons werden fest programmierte Zustandsdaten in die angezeigte UI-Komponente injiziert.


Abbildung 5.3: Screenshot der Beispielanwendung

Zum Abschluss dieses Kapitels werden nachfolgend noch einmal alle Kernaspekte zusammengefasst, um somit einen umfassenden Überblick über die Implementation der vorliegenden Arbeit zu geben.
5.6 Zusammenfassung

Dieses Kapitel behandelte die prototypische Umsetzung des entwickelten Konzeptes. In diesem Zusammenhang wurden zu Beginn die verwendeten Technologien in Form vorhandener Implementierungen und Frameworks vorgestellt. Anschließend wurden die Erweiterungen, welche an der CRUISe-Thin-Server-Runtime vorgenommen wurden, beschrieben. Dabei präsentierte Abschnitt 5.2.1 die Anpassungen innerhalb des Application Managers, wie bspw. die Erweiterung des Konstruktors oder die Registrierung eines System-Kanals.

In Abschnitt 5.3 wurde die Senderlogik vorgestellt. Zunächst wurde dabei die entworfene Basisklasse, welche die Grundfunktionalitäten der UI-Migration für andere UI-Komponenten bereitstellt, präsentiert und daran anschließend erläuterte Abschnitt 5.3.1 die Umsetzung der Zustandserfassung bei JavaScript-basierten Komponenten. Im darauffolgenden Abschnitt 5.4 wurde die Wiederherstellung des Komponentenzustandes auf Programmebene erörtert. Analog zu den vorherigen Abschnitten bildete ein Codebeispiel dabei die Grundlage der Beschreibung. In diesem Zusammenhang stand die Erweiterung der Komponentenschnittstelle von CRUISe-Komponenten im Fokus der Betrachtungen. Abschließend wurde in 5.5 die entwickelte Beispielanwendung überblicksweise vorgestellt.

Im letzten Kapitel der vorliegenden Arbeit werden noch einmal alle Erkenntnisse sowie die erreichten Ergebnisse beschrieben und bewertet.
6 Zusammenfassung


6.1 Ergebnisse


In Kapitel 3 stand die Betrachtung von Lösungsansätzen für die Zustandserfassung, -extraktion, -serialisierung und -injektion im Mittelpunkt. In diesem Zusammenhang wurde der Austausch von Komponenten in CRUISe, die Atomarität und Zustandserhaltung in CoBRA, die OPEN Migration Service Platform, die Generierung von Zustandsgraphen in COSMOD sowie die (DOM-Baum-)Event-Registrierung untersucht.


Die prototypische Umsetzung des Konzepts behandelte Kapitel 5. Mit der erfolgreichen Implementierung eines Großteils des Konzeptes und der darauf aufset-

6.2 Ausblick

Nachdem im vorherigen Abschnitt die Ergebnisse der vorliegenden Arbeit präsentiert wurden, folgt an dieser Stelle die Beschreibung identifizierter Problemstellungen und Ansätze für weiterführende Arbeiten zum Thema UI-Migration.

Unterstützung heterogener UI-Komponenten Während den Arbeiten an der vorliegenden Forschungsarbeit musste festgestellt werden, dass die Unterstützung heterogener UI-Komponenten, welche zwar funktional äquivalent sind, sich aber sowohl im internen Aufbau, als auch in der Programmiersprache unterscheiden, nicht gewährleistet werden kann. Für diese Herausforderung wurden allerdings zwei mögliche Lösungsansätze erwähnt, welche nachfolgend noch einmal überblicksweise ins Gedächtnis gerufen werden sollen.


2. Transformation der extrahierten Zustandsinformationen Eine weitere Möglichkeit heterogene UI-Komponenten für die UI-Migration in CRUISe zu unterstützen, wäre die Transformation der erfassten Zustandsdaten. In diesem Zusammenhang wäre es denkbar, das Zustandsmodell der Ausgangskomponente serverseitig an das zugrundeliegende Zustandsmodell der Zielkomponente anzupassen.
Implementierung des **UIStateObservers** in anderen Programmiersprachen


Berücksichtigung von **Sicherheitsaspekten** Eine weitere Problemstellung, welche sowohl derzeitig als auch zukünftig immer mehr an Bedeutung gewinnt, ist das Thema **Sicherheit**. Aufgrund der Tatsache, dass die Berücksichtigung sicherheitsspezifischer Aspekte sehr komplex und umfassend ist, wurden diese im Rahmen der Arbeit allerdings nicht betrachtet und bieten somit ein sehr breites Aufgabenspektrum für zukünftige Forschungsarbeiten auf dem Gebiet der UI-Migration.

Anbindung eines **Kontext-Services** In den Vorbetrachtungen zu der UI-Migration in CRUISe wurde in dem Abschnitt 4.3.4 mehrfach die Möglichkeit zur Nutzung eines externen Kontext-Services beschrieben. In diesem Zusammenhang wäre die Verwendung dieses Services z.B. bei der **automatischen Auslösung der UI-Migration** oder beim **Auffinden einer passenden UI-Komponente**, welche im Kontext des Zielgerätes integriert werden soll, durchaus denkbar und hilfreich.

Automatische Registrierung aller Endgeräte im Netzwerk Ein weiterer Aspekt, welcher ebenfalls im Abschnitt 4.3.4 diskutiert wurde, ist die **automatische Registrierung** aller für die UI-Migration in Frage kommenden Endgeräte eines Netzwerks. In diesem Zusammenhang ist die Nutzung von etablierten Technologien wie UPnP oder Bonjour denkbar.

Unterstützung **mehrerer, parallellaufender UI-Komponenten** Die letzte Problemstellung, welche aufgrund hoher Komplexität im Rahmen der vorliegenden Arbeit nicht betrachtet werden konnte, ist die **Migration mehrerer, miteinander verknüpfter und parallellaufender UI-Komponenten**. Die Herausforderung besteht darin, diese Komponenten zu pausieren und voneinander abzukoppeln ohne dabei die Konsistenz des Zustandes jeder einzelnen Komponente zu beeinträchtigen. In diesem Zusammenhang muss bspw. die Auswirkung der Pausierung einer Komponente auf alle anderen UI-Komponenten, welche sich in einem kollaborativen Umfeld befinden, untersucht werden.
### Abbildung A.1: Evaluation der untersuchten Forschungsprojekte
Abbildung A.2: Grundlegende Prozessschritte der UI-Migration
Abbildung A.3: Prozessschritte der konzeptionierten Zustandserfassung
Abbildung A.4: Aufbau des Kommunikationsprotokolls
Abbildung A.5: Prozessschritte der konzeptionierten Zustandsinjektion
Literaturverzeichnis


[Böhm09] Thomas Böhm. »Grundlagen komponentenorientierter Softwareentwicklung«. In: (Mai 2009), Seiten 1–35 (siehe Seite 13).


[GW12] Rainer Gimmich und Andreas Winter. »Workflows der Software-Migration«. In: (Mai 2012), Seiten 1–3 (siehe Seite 5).


[Pat09] Fabio Paterno. »Logical User Interface Descriptions«. In: (Mai 2009), Seiten 1–35 (siehe Seite 34).


[Rad10] Carsten Radeck. »Kontextmodellierungs- und Adaptionsmechanismen für komposite Webanwendungen«. In: (Mai 2010), Seiten 1–126 (siehe Seite 22).


[SS06] Fabio Paternò Silvia Berti und Carmen Santoro. »A Taxonomy for Migratory User Interfaces«. In: (März 2006), Seiten 1–12 (siehe Seiten 6–10).
