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

Erklärung

Hiermit erkläre ich, Christoph Pohl, den vorliegenden Großen Beleg zum Thema

Modellierung des Komponentenzustandes von JavaScript-basierten Mashup-Komponenten

selbstständig und ausschließlich unter Verwendung der im Literaturverzeichnis angegebenen Quellen und Hilfsmittel verfasst zu haben.

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

Abbildungsverzeichnis

Listings

1 Einleitung

1.1 Motivation

Neben stationären Desktop-PCs und mobilen Notebooks etablieren sich seit wenigen Jahren Smartphones und Tablets zu Geräten, die täglich von vielen Menschen weltweit genutzt werden [Lev12]. Aufgrund der Tatsache, dass diese *ultra-mobilen Endgeräte* heutzutage mit einer Notebook-ähnlichen Performance aufwarten können, werden sie immer beliebter im Alltag und vor allem auch im Geschäftsumfeld. Neben der Informationsbeschaffung aus dem Internet, gehören kollaborative Anwendungsszenarien, in denen mehrere Anwender gemeinsam interagieren, zu den Hauptaufgaben solcher mobilen Endgeräte. Das *World Wide Web* spielt dabei eine maßgebliche Rolle, da es ein plattformunabhängiges und länderübergreifendes Zusammenarbeiten ermöglicht. Mit dem Web 2.0 entwickelte sich das Internet von einer Informationsplattform hin zu einer Kollaborationsplattform, in der es Nutzern unter anderem ermöglicht wird, ohne außergewöhnliche Programmierkenntnisse Mashups[1] mit wenigen Mausklicks zu erstellen, um neue Informationen zu erhalten oder sie für andere bereitzustellen. Um solche *serviceorientierten Architekturen* möglichst einfach umzusetzen, stellen unter anderem die Forschungsprojekte **CRUISe**[2][Pie09] und **DoCUMA**[3][Mul12] entsprechende Werkzeuge dafür bereit. Sie stellen sich der Herausforderung eine komplexe und dynamische Komposition von wiederverwendbaren, webbasierten User-Interface-Komponenten zu ermöglichen. CRUISe widmet sich dabei der modellgetriebenen Entwicklung und Bereitstellung serviceorientierter Webanwendungen, um das Web-Service-Prinzip auf die Präsentationsebene zu übertragen und somit verschiedenste Dienste zusammenzustellen, die sowohl Daten, Geschäftslogik als auch UI-Bestandteile bereitstellen können. Der Auswahl-, Konfigurations- und Anpassungsvorgang dieser Komposition zur Laufzeit geschieht dabei kontextabhängig, um möglichst adaptive und passgenaue Webanwendungen entstehen zu lassen. DoCUMA baut auf dieses Konzept auf und fokussiert sich dabei auf die Möglichkeiten der dynamischen Verteilung aller Anwendungsbestandteile.
Um die zu Beginn erwähnten kollaborativen Anwendungsszenarien genauer zu definieren und zu veranschaulichen soll folgendes Beispiel dienen:
Christoph ist auf dem Weg zu einer Konferenz und schaut sich seine Präsentationsfolien auf einem Tablet an. Neben zahlreichen Slides mit Text und Bildern hat er auch ein Video zur Veranschaulichung eingebettet. Bevor die eigentliche Konferenz stattfindet berät er sich mit einem Kollegen über einige Folien in der Präsentation. Damit beide nicht gezwungen sind auf das kleine Display des Tablets zu schauen, wird die Präsentation auf den Smart-TV im Konferenzraum übertragen. Das Tablet

[1] Web-Applikationen die durch Kombination bestehender Web-Services neue Inhalte generieren können
[2] Composition of Rich User Interface Services
[3] Development of Composite and Ubiquitous Mashup Applications

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.
Dieses Anwendungsszenario zeigt wie essentiell **Zustandsinformationen** sein können, um beispielsweise komplette User-Interface-Komponenten von einem Gerät (Tablet) zu einem anderen (Smart-TV) zu übertragen. Da in solchen Multi-User- bzw. Multi-Device-Szenarien heterogene Endgeräte und in Mashup-Anwendungen unterschiedliche Anwendungskomponenten zum Einsatz kommen, erfordert die Migration dieser UI-Komponenten die Ausführung eines Austauschprozesses. Hierbei müssen die **Zustandsdaten** der zu ersetzenden Komponente extrahiert und in die neue Komponente injiziert werden. Dabei ist es notwendig, die Zustandsinformationen auf Basis eines **technologie- und plattformneutralen Zustandsmodells** auszutauschen, um eine möglichst hohe Kompatibilität zwischen unterschiedlichen Endgeräten zu gewährleisten. Im Rahmen des Forschungsprojektes DoCUMA, wird in dieser Arbeit das Augenmerk auf den **Transfer von Zustandsinformationen** zwischen unterschiedlich implementierten Anwendungskomponenten gelegt. Neben der Frage wie der Zustand einer Anwendungskomponente modelliert werden kann, wird der damit verbundene Extraktions- und Injektionsprozess dieser Daten näher betrachtet.

1.2 Zielstellung

Ziel dieser Arbeit ist die Konzeption und Umsetzung eines technologie- und plattformneutralen Zustandsmodells. Des Weiteren wird ein Verfahren entwickelt, diese Zustandsinformationen zu extrahieren bzw. zu injizieren, um die Migration und Replikation von Anwendungskomponenten zur Laufzeit zu ermöglichen.
Um dieses Ziel zu erreichen, ist es zunächst wichtig ein Verständnis für Softwarekomponenten zu schaffen. Die Analyse von Herausforderungen hinsichtlich der Umsetzung eines Zustandsmodells für Softwarekomponenten, sowie der Erarbeitung eines geeigneten Extraktions- bzw. Injektionsalgorithmus, bilden den Ausgangspunkt für Betrachtungen zu bereits existierenden Ansätzen und Technologien in Forschung und Technik.
Die Ergebnisse dieser Analyse fließen dann in die Konzeption eines eigenen Zustandsmodells ein, welches die Extraktion und Injektion der serialisierten Zustandsdaten einer Komponente ermöglicht. Zur Verifizierung des Konzepts dient eine prototypische Implementierung innerhalb der CRUISe-Laufzeitumgebung.

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

Beschreibung näher betrachtet. Eine überblicksartige Beschreibung des Forschungsprojektes CRUISe bildet den Abschluss des 2. Kapitels.
In Kapitel 3 werden existierende Konzepte für Komponentenmodelle vorgestellt. Dabei wird der Fokus vor allem auf die jeweilige Zustandsmodellierung gelegt und die Technologien für die Extraktion bzw. Injektion dieser Informationen analysiert. In diesem Zusammenhang werden unter anderem die Forschungsprojekte *CoBRA*, *OPEN* und *COSMOD* näher betrachtet.
Zu Beginn des 4. Kapitels werden funktionale und nicht-funktionale Anforderungen an Komponenten und deren Laufzeitumgebung, unter der Berücksichtigung des Black-Box-Paradigmas, erhoben. Anschließend werden die untersuchten Ansätze aus dem vorangegangenen Kapitel mit Hilfe dieser Anforderungen evaluiert. Den Schwerpunkt des Kapitels bildet der Entwurf eines eigenen Ansatzes zur Modellierung des Komponentenzustandes, sowie die Entwicklung eines geeigneten Mechanismus zur Extraktion und Injektion von zustandsbehafteten Informationen.
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

In diesem Kapitel werden grundlegende Begriffe und Technologien definiert, auf denen die vorliegende Arbeit aufbaut. Zu Beginn werden Mashups als **komposite Webanwendungen** überblicksweise vorgestellt, da diese die Grundlage der vorliegenden Forschungsarbeit sind. Anschließend werden **Bestandteile und Konzepte der UI-Migration** erläutert und die Notwendigkeit sowie die Vorgehensweise einer komponentenbasierten Softwareentwicklung erklärt. Des Weiteren werden **Anforderungen an Softwarekomponenten** näher betrachtet, die für diese Arbeit relevant sind. Anschließend wird der Begriff des **Komponentenmodells** allgemein definiert und auf die Umsetzung dieses Konzeptes in dem Forschungsprojekt CRUISe eingegangen. Im letzten Abschnitt wird die **Architektur von CRUISe** überblicksweise vorgestellt und Defizite des Projekts, bezüglich der Migrationsfähigkeit, aufgezeigt.

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 [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.
Eine grundlegende Eigenschaft der Architektur von kompositen Webanwendungen ist die *Komponentenorientierung*. Dabei wird auf die Definition des Begriffs „Softwarekomponente“ von Szyperski [CD98] (siehe Abschnitt 2.3) Bezug genommen. Diese *Softwarekomponenten* verfügen über klar definierte Schnittstellen und können mit anderen Softwarekomponenten kombiniert werden. Des Weiteren besteht die Möglichkeit sie zu konfigurieren und auszutauschen. Eine Softwarekomponente unterliegt dabei einem bestimmten Komponentenmodell, welches u. a. den Inhalt, die Schnittstellen zu anderen Komponenten, die Kommunikationsart sowie die Schnittstelle zur Laufzeit definiert, und einem Kompositionsmodell, das Möglichkeiten zur Komposition von Komponenten spezifiziert.
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 [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 [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 **UI-Migration** zusammengefasst und im nachfolgenden Abschnitt näher untersucht.

2.2 UI-Migration

Nachdem der erste Abschnitt des Grundlagenkapitels den Begriff der *kompositen Webanwendung* näher erläutert hat, befasst sich dieser Abschnitt mit der *UI-Migration*. Nach einer grundlegenden Definition und Abgrenzung des Begriffs, werden verschiedene Migrationsarten näher betrachtet und wichtige Grundkonzepte des Migrationsprozesses erläutert. Abschließend wird eine beispielhafte Architektur in Zusammenhang mit einem möglichen Migrationsablauf vorgestellt, um ein grundlegendes Verständnis für die UI-Migration zu schaffen.

2.2.1 Begriffsdefinition und -abgrenzung

Unter dem Begriff **Migration** wird in der IT-Welt der Transfer von Datenbeständen auf eine andere Hard- oder Softwareplattform verstanden. Im Gegensatz zur Daten-Migration, bei der eine Plattform ersetzt wird, mit welcher Daten verwaltet und von einem Altsystem übernommen werden, steht bei der Software-Migration die Überführung eines Softwaresystems von einer Umgebung in eine andere Zielumgebung im Fokus [GW12]. In konventionellen Migrationsszenarien, wie beispielsweise die Anpassung plattformgebundener Software an ein neues Hardware-System, werden Backend-Systeme oder Datenbank-Server als solche Umgebungen verstanden. Im Kontext dieser Forschungsarbeit steht die UI-Migration im Fokus der Untersuchungen. Dabei wird die komplette Benutzerschnittstelle, oder Teile davon, von einem Ausgangsgerät auf ein Zielgerät übertragen. Nach Übertragung des UI, wird die Benutzerschnittstelle auf dem Ausgangsgerät komplett deaktiviert oder zwischen den beteiligten Geräten aufgeteilt. Wie in Abbildung 2.1 dargestellt, werden bei der UI-Migration verschiedene Geräte mit unterschiedlichen Spezifikationen unterstützt.

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

inklusive aller eingegebenen Daten und bisher ausgeführten Operationen von einem Ausgangsgerät auf ein Zielgerät zu übertragen. Um die Notwendigkeit der Zustandserhaltung bei der UI-Migration zu verdeutlichen, wird im Folgenden ein kurzes Anwendungsszenario beschrieben:
Christoph arbeitet zu Hause auf seinem Tablet mit einer kompositen Mashup-Anwendung, welche unter anderem aus einer Radio-Applikation, einem Chat-Client und einem Kartendienst inklusive Navigationsfunktion zusammengesetzt ist. Er verabredet sich mit einem Bekannten über die Chatfunktion, markiert den Treffpunkt in dem Kartendienst und ist aus Zeitgründen gezwungen sofort loszufahren. Auf der Autobahn bekommt er über das Radio seines Tablets mit, dass sich ein Stau auf seiner Strecke gebildet hat. Christoph entscheidet sich spontan die Abfahrt zu nehmen, obwohl er sich auf den Landstraßen der Umgebung nicht auskennt. Sofort migriert er den Kartendienst von seinem Tablet auf sein Smartphone über ein Sprachinterface. Aufgrund der Tatsache, dass der Zustand des Kartendienstes konsistent geblieben ist, kann er ohne weitere Eingaben des Zielortes seine Fahrt fortsetzen und nutzt sein Smartphone als Navigationsgerät.
Dieses Beispiel hat gezeigt, dass es möglich ist mit Hilfe der UI-Migration Benutzerschnittstellen von einem Ausgangsgerät auf ein Zielgerät zu übertragen. Findet die Migration, wie bei diesem Anwendungsszenario, dabei in einer intelligenten Umgebung mit heterogenen Geräten statt, ist in vielen Fällen eine Anpassung der UI an die jeweiligen Spezifikationen (z. B. Bildschirmgröße oder Prozessorleistung) des Zielgerätes notwendig. Diese Adaption erfolgt für den Endanwender transparent und ermöglicht es, die gegenwärtige Tätigkeit im Kontext des aktuellen Endgerätes zu unterbrechen und sie im Kontext eines anderen Endgerätes fortzusetzen. Somit kann der Nutzer seine Aufgaben ohne Unterbrechung auf den ihm zur Verfügung stehenden Geräten bearbeiten. Um dies zu ermöglichen sind die folgenden Punkte zu betrachten [SS06]:

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).

2. **Adaption.** In vielen Fällen ist es notwendig das migrierende User-Interface des Ausgangsgerätes an die Spezifikationen des Zielgerätes anzupassen, wenn Software beispielsweise von einem stationären PC auf ein ultra-mobiles Smartphone migriert wird. Dieser Adaptionsprozess kann unterschiedlich stark ausgeprägt sein und es wird dabei zwischen der Präsentations-, Navigations- und inhaltlichen Ebene der Benutzerschnittstelle unterschieden. Wenn eine Anwendung zwischen Geräten migriert wird, die unterschiedliche Bildschirmgrößen besitzen, ist es meist notwendig das Layout der Präsentation anzupassen (Skalierung). Die Navigationsstruktur zwischen den verschiedenen Präsentationen untereinander kann sich ebenfalls verändern, indem sich die Reihenfolge einzelner Interface-Objekte verändert oder wenn sich die Anzahl dieser Objekte reduziert bzw. zunimmt. Eine Reduktion der Interface-Objekte findet meis-

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 Netzwerkkonnektivität 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 Beschreibung 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 migriert. 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 Programmablauf 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 Ausgangsgerätes herausgelöst und auf das Zielgerät migriert. Der Rest bleibt dabei auf dem Ausgangsgerät erhalten. In [BP04] findet diese Art der Migration beispielsweise Anwendung. Dabei wird zwischen den Bereichen Benutzerinteraktion 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 ausschließlich zur Steuerung und der Smart-TV lediglich zur Darstellung genutzt werden.

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

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

Aufgrund der Tatsache, dass es wichtig ist einzelne Bestandteile der Benutzerschnittstelle aus einer Gesamtanwendung herauslösen zu können, wird im Kontext der Aufgabenstellung ausschließlich die *partielle Migration* im Fokus dieser Forschungsarbeit stehen. Neben diesen verschiedenen Migrationsarten, werden in [SS06] weitere Dimensionen 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.
Neben all diesen verschiedenen Eigenschaften des Migrationsvorgangs, ist vor allem die im vorangegangen Abschnitt erwähnte **Zustandserhaltung** ein sehr wichtiger Aspekt der UI-Migration. Hierbei ist es essentiell, dass am Anfang des Migrationsprozesses der aktuelle Zustand, der zu migrierenden Benutzerschnittstelle, erfasst wird. Dieser Zustand resultiert aus den vorangegangenen Benutzerinteraktionen mit der Anwendung, übertragenden Daten, bereits besuchten Seiten sowie Ergebnissen vorheriger Datenverarbeitungen [SS06]. Er kann auf 2 unterschiedliche Arten von dem Ausgangs- zum Zielgerät übertragen werden: Im Gegensatz zu einer Peer-to-Peer-Übertragung, bei der die Geräte direkt miteinander kommunizieren, behandelt bei einer Client/Server-Architektur eine dritte Komponente die Migrationsanfragen und sendet die Daten an das Zielgerät. Bei dem letztgenannten Ansatz, welcher in dieser Forschungsarbeit betrachtet wird, werden alle Zustandsdaten von der UI-Komponente selber gesammelt und an die serverseitige Migrationskomponente gesendet.
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,

dass der ursprüngliche Zustand der sendenden Komponente weitestgehend im Empfangskontext wiederhergestellt werden kann. Dieser Prozess der Zustandswiederherstellung im Zielkontext wird im weiteren Verlauf der Arbeit als *Zustandsinjektion* beschrieben und soll in Abschnitt 4.8 näher erörtert werden.
Nachdem in diesem Abschnitt wichtige Eigenschaften des Migrationsvorgangs beschrieben wurden, befasst sich der nächste Abschnitt mit einem Ansatz zur Durchführung der UI-Migration und der zugrunde liegenden Architektur im Überblick.

2.2.3 Durchführung der UI-Migration im Überblick

Nachdem der Begriff der UI-Migration definiert wurde und Teilkonzepte dieser Migrationsart vorgestellt wurden, soll in diesem Abschnitt die konkrete Durchführung einer UI-Migration anhand den Abbildungen 2.3 und 2.2 beschrieben werden. Beides wurde dabei auf Grundlage von Abbildungen aus [SS06] und [Nic+10] entwickelt.
Bild 2.2 zeigt eine beispielhafte Architektur, mit der eine UI-Migration ermöglicht werden kann. Die Nummerierung an den Pfeilen beziehen sich dabei auf die einzelnen Schritte des Ablaufdiagramms in Abbildung 2.3. Wie in der Architekturübersicht zu erkennen ist, sind diverse *Migration Clients* sowie ein *Migration Server* und ein weiterer Server, auf dem das *Component Repository* (Komponentenrepositorium, kurz: CoRe) liegt, an dem Migrationsprozess beteiligt. Die *Migration Clients* stellen die benutzten Geräte des Nutzers dar und enthalten somit die zu migrierenden UI-Komponenten. Auf dem Migrationsserver finden Berechnung statt, die für eine erfolgreiche UI-Migration notwendig sind. Eine Beschreibung dieser Berechnungen folgt im nächsten Abschnitt. Des Weiteren ist dieser Server, bei einer Migration zwischen zwei heterogenen Geräten, für die Auswahl kompatibler UI-Komponenten aus dem CoRe zuständig.

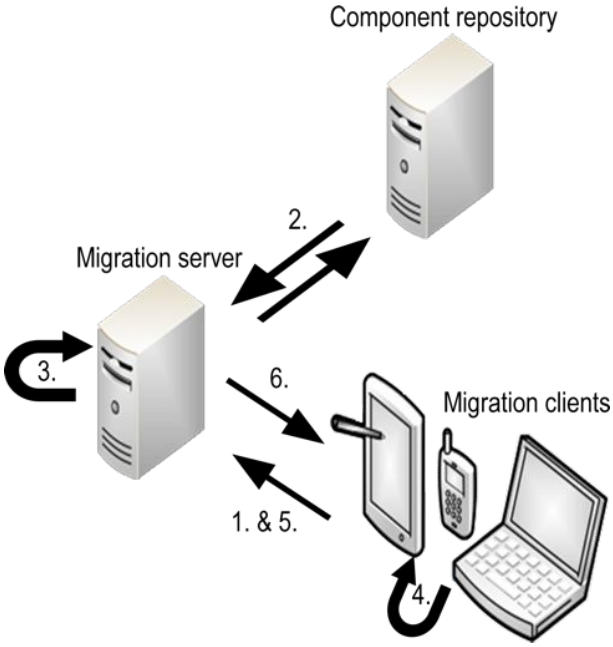

Abbildung 2.2: Mögliche Architektur für eine UI-Migration

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 initiierte (oder automatische) Anfrage eine UI-Komponente zwischen zwei Geräten zu migrieren.

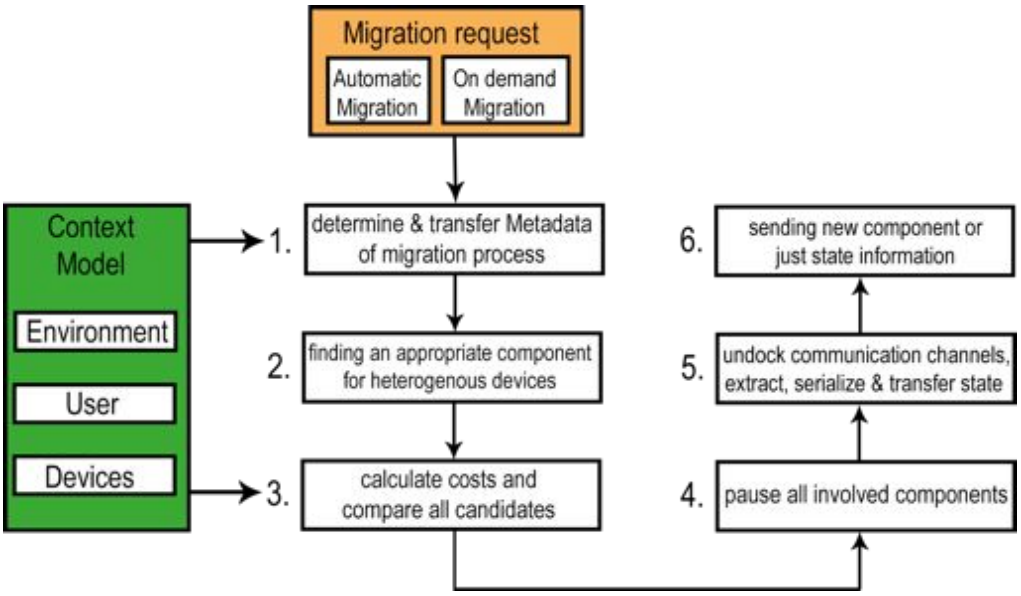

Abbildung 2.3: Möglicher Ablauf einer UI-Migration

1. Im ersten Schritt übermittelt das Ausgangsgerät (Teil der Migration Clients) erforderliche Metadaten des Migrationsprozesses an den Migration Server. Dazu zählen zum Beispiel die Kontextinformationen der Umgebung, des Nutzers und des Ausgangsgerätes (Context Model) [SS06]. Bestandteile der Umgebungsinformationen sind unter anderem die Umgebungsgeräusche. Befindet sich der Nutzer in einer lauten Umgebung, wäre es in diesem Fall vorteilhaft Audio-Komponenten durch passende Video-Komponenten wenn möglich zu ersetzen. Der Standort sowie mögliche Präferenzen zählen zu den Kontextinformationen des Nutzers. Technische Spezifikationen, wie beispielsweise Bildschirmgröße oder Prozessor, werden den Kontextinformationen der Geräte zugeordnet. Zusätzlich zu diesen Informationen gehört eine Referenz auf das Zielgerät sowie die ID der zu migrierenden UI-Komponente ebenfalls zu den Metadaten. Letzteres verwendet der Server, um aus dem zentralen CoRe die zur ID zugehörige Komponentenbeschreibung zu erhalten.

2. Nachdem die Infos übertragen wurden, ermittelt der Server mit Hilfe der Beschreibungen des Ausgangs- und Zielgerätes, ob die beteiligten Geräte homogen oder heterogen sind. Unterscheiden sich die beteiligten Geräte der UI-Migration nicht, so bleibt die UI-Komponente gleich. Bei heterogenen Geräten hingegen bestimmt der *Migration Server* mit Hilfe des CoRe passende UI-Komponenten. Diese müssen zum einen kompatibel zur ursprünglichen Komponente und zum anderen auf dem Zielgerät ausführbar sein.

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

Zielgerät erzeugen würde, um eine mögliche Überlast auf dem Zielgerät zu verhindern. Hierbei werden die zuvor übertragenen Informationen zum Kontext des Zielgerätes verwendet. Die Ergebnisse der Schätzfunktion aller passenden UI-Komponenten werden miteinander verglichen, um letztendlich die Komponente mit den geringsten Ausführungskosten zu wählen. Sollte keine passende UI-Komponente gefunden werden, wird der Migrationsprozess an dieser Stelle abgelehnt.

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.

5. Die Zustandsdaten der zu migrierenden UI-Komponente werden im nächsten Schritt extrahiert und anschließend serialisiert, um sie an den Migration Server übertragen.

6. Nach der erfolgreichen Übertragung werden alle Informationen auf dem Server zwischengespeichert, um eine Wiederholung oder Fortsetzung der UI-Migration, bei einem Verbindungsabbruch, gewährleisten zu können. Danach ermittelt der Server, ob eine alternative UI-Komponente an das Zielgerät gesendet werden muss oder ob die Migrationsdaten ohne Veränderungen an das Zielgerät weitergeleitet werden können. Anschließend wird die UI-Komponente auf dem Zielgerät integriert und mit den Zustandsdaten der Ursprungskomponente bestückt. Nach dem erfolgreichen Migrationsvorgang, wird im letzten Schritt die pausierte UI-Komponente aus der Gesamtkomposition des Ausgangsgerätes herausgelöst, wobei bestehende Kommunikationskanäle mit anderen Komponenten abgekoppelt werden.

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

Der Abschnitt 2.2 definierte zuerst grundlegend den Begriff der UI-Migration und grenzte ihn von anderen Migrationsarten ab. Dabei haben sich essentielle Bestandteile einer UI-Migration, wie der Aspekt der Zustandserhaltung oder die Adaption, herauskristallisiert. Anschließend wurden Teilkonzepte der UI-Migration erläutert. Nach einer kurzen Vorstellung verschiedener Migrationsarten, wurde aufgrund der zugrunde liegenden Anforderungen der Aufgabenstellung festgelegt, dass im Rahmen dieser Forschungsarbeit die partielle Migration im Fokus stehen muss. Des Weiteren wurden Aktivierungstypen und Migrationsmodalitäten überblicksweise beschrieben. Hierbei wurde festgestellt, dass diese Teilkonzepte für die Lösung der Aufgabenstellung nicht relevant sind. Im letzten Teil wurde ein beispielhafter Migrationsprozess im Zusammenhang mit einer möglichen Architektur beschrieben. Dabei wurde festgelegt, in welchen Prozessschritten sich diese Arbeit ansiedelt. Betrachtungen von CRUISe in den Abschnitten 2.3.3 und 2.4 werden in diesem Zusammenhang zeigen,

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

Dieser Abschnitt des Grundlagenkapitels wird den Begriff „Softwarekomponente“ definieren und Anforderungen an diese spezifizieren. Anschließend wird der Begriff „Komponentenmodell“ geklärt und die Umsetzung dieses Konzeptes in CRUISe beschrieben.
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.

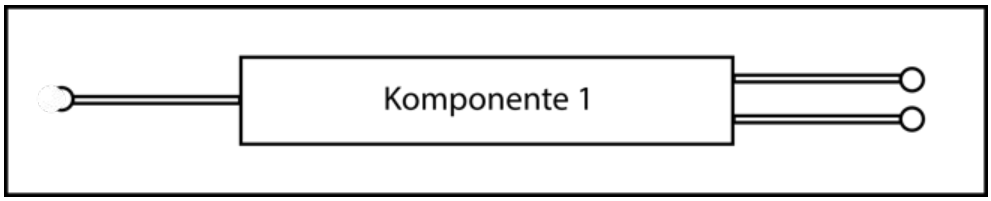

Abbildung 2.4: Mögliche schematische Darstellung einer Softwarekomponente

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).

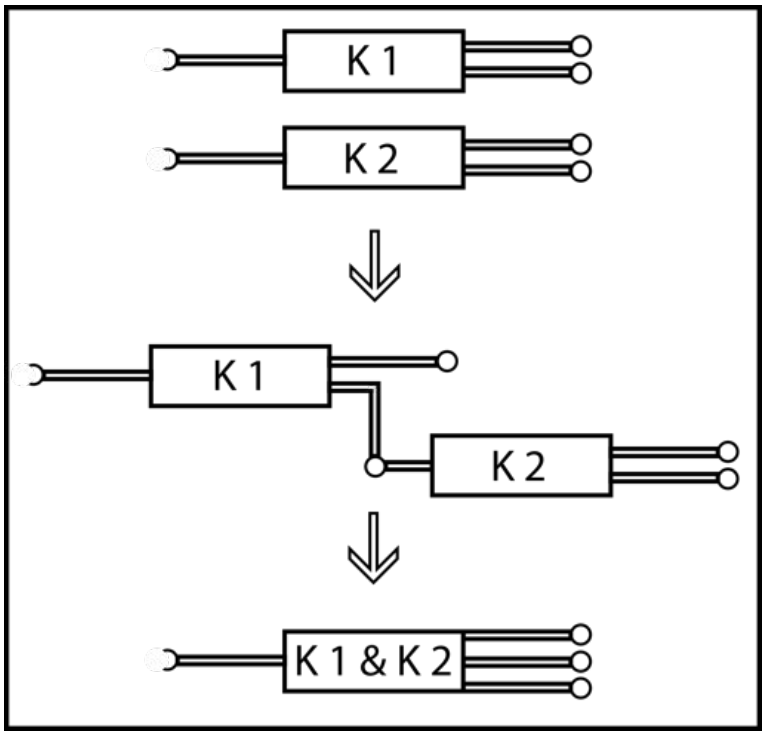

Abbildung 2.5: Komposition - direkte Kopplung

Hierbei werden die angebotenen Schnittstellen der einen Komponente mit den benötigten Schnittstellen der anderen Komponente direkt verknüpft. Dies setzt voraus, dass die verschiedenen Komponenten ein einheitliches Komponentenmodell besitzen. Auch hier gilt, dass von Außen nicht ersichtlich ist, dass es sich um eine Komposition handelt. Eine weitere Form der Komposition ist die lose Kopplung (siehe Abbildung 2.6). Der Grundgedanke besteht darin, den Grad der Abhängigkeit zwischen den Komponenten auf ein Minimum zu reduzieren. CRUISe verwendet dabei das *Publish-Subscribe-Paradigma*, um die lose Kopplung zu realisieren. Hierbei registriert sich zunächst jede Komponente auf ein bestimmtes Event bei einem Server

(*subscribe*). Wenn jetzt eine Nachricht veröffentlicht wird (*publish*), wird diese an den Server geschickt und von da aus an jede Komponente weitergeleitet, die sich auf dieses Event beim Server registriert hat.
Eine weitere Eigenschaft, die eine Softwarekomponente im Rahmen dieser Forschungsarbeit aufweisen soll, ist die Fähigkeit der **Migration**. Wie in Kapitel 2.2.1 und 2.2.2 beschrieben, spielt dabei der Komponentenzustand eine zentrale Rolle. Kurz zusammengefasst kann man diesen Zustand als das Resultat bereits besuchter Seiten, übertragener und verarbeiteter Daten sowie vorangegangener Benutzerinteraktionen angesehen werden. Mit Hilfe von Schlüssel-Wert-Paar-Variablen, abgespeichert innerhalb einer Komponente, wäre es somit theoretisch möglich den Zustand zu serialisieren. Dabei stellt sich aber vor allem die Frage, wie feingranular der Komponentenzustand beschrieben werden kann und ob diese Methode mögliche Implementierungsveränderungen an einer Komponente beeinflusst bzw. sich automatisch anpasst. Um die Antwort auf diese Frage zu erhalten und weitere Möglichkeiten der Zustandsserialisierung sowie -extraktion und -injektion zu entdecken, werden im kommenden Kapitel die „Komponentenmodelle“ näher betrachtet.

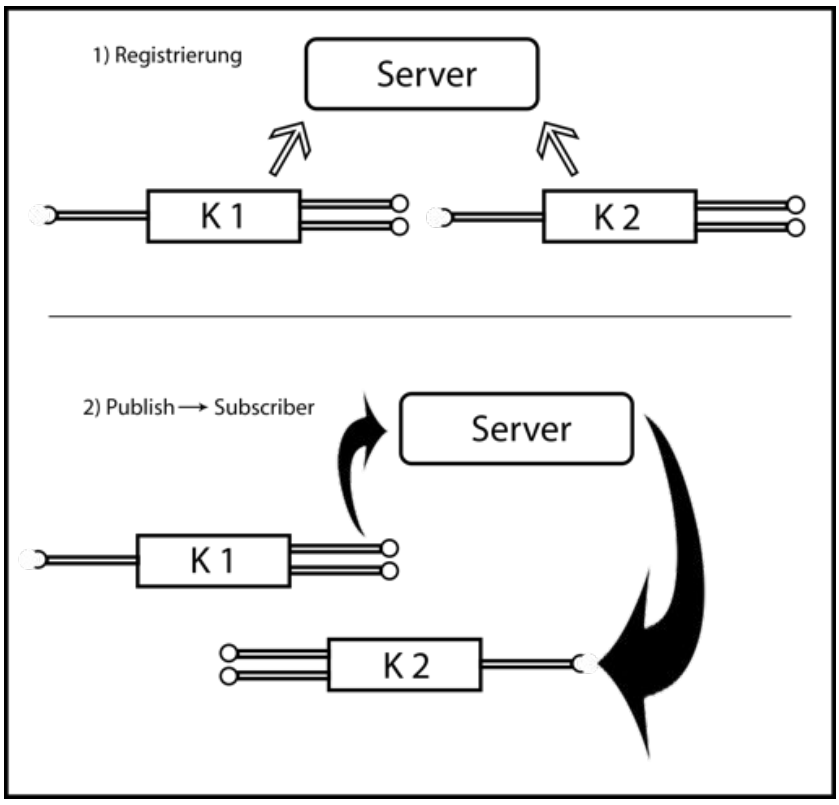

Abbildung 2.6: Komposition - lose Kopplung

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]

Nach Anne Thomas definiert das Komponentenmodell die grundlegende Struktur der Komponentenschnittstelle sowie die Mechanismen der Interaktion zwischen der Komponente selbst und deren Container oder anderen Komponenten. Des Weiteren bieten diese Modelle einen Rahmen für die Entwicklung und Ausführung von Komponenten und stellen strukturelle Anforderungen hinsichtlich der Kompositionsmöglichkeiten. Demnach können Komponentenmodelle als „Bauplan“ angesehen werden, welcher dem Entwickler Standards und Vorgehensweisen zur Softwareentwicklung bereitstellt. Im nächsten Abschnitt wird das Komponentenmodell des CRUISe-Projektes überblicksweise vorgestellt.

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.
- **Kontext-Komponenten** generieren und verarbeiten Kontextinformationen wie beispielsweise die aktuelle geographische Position des Nutzers. Diese Komponenten können auch mit anderen Kontextdiensten zusammenarbeiten.
- **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.

Das Komponentenmodell stellt keine Vorgaben über den internen Aufbau der Softwarekomponenten auf, spezifiziert aber grundlegende Charakteristiken der externen Schnittstellen. Abbildung 2.7 zeigt den grundlegenden Aufbau einer Softwarekomponente in CRUISe. Wie in der Abbildung zu sehen ist, besteht jede Komponente in CRUISe grundsätzlich aus 3 Teilen, welche im Folgenden näher erläutert werden: In den **Properties** werden Schlüssel-Wert-Paare abgespeichert, die es ermöglichen verschiedene Eigenschaften einer Komponente zur Entwicklungs- bzw. Laufzeit anzupassen. Welche Parameter genau in den Properties enthalten sind und wie feingranular diese sind liegt im Ermessen des Komponentenentwicklers. Zustandsänderungen

werden mit Hilfe von ausgelösten **Events** publiziert, welche durch einen Namen und einer Menge von Parametern gekennzeichnet werden. **Operations** sind parametrisierte Methoden und werden zur Verfügung gestellt, um anderen Komponenten den Zugriff auf eigene Funktionalitäten zu erlauben und beispielsweise auf publizierte Events zu reagieren.

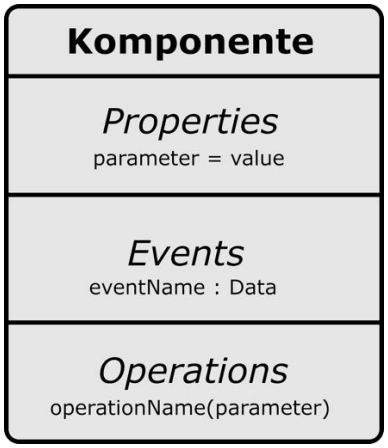

Abbildung 2.7: CRUISe - Komponentenmodell

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 SOA[1]-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 funktionalen Bausteine einer Anwendung).

- Das **Communication Model** spezifiziert den Daten- und Kontrollfluss. Dabei wird ein datenflussorientiertes Paradigma genutzt, welches sich an den Möglichkeiten des Komponentenmodells ausrichtet. Demnach stehen Properties, Events und Operations zur Verfügung, die eine ereignisgesteuerte Kommunikation implizieren.

- 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.

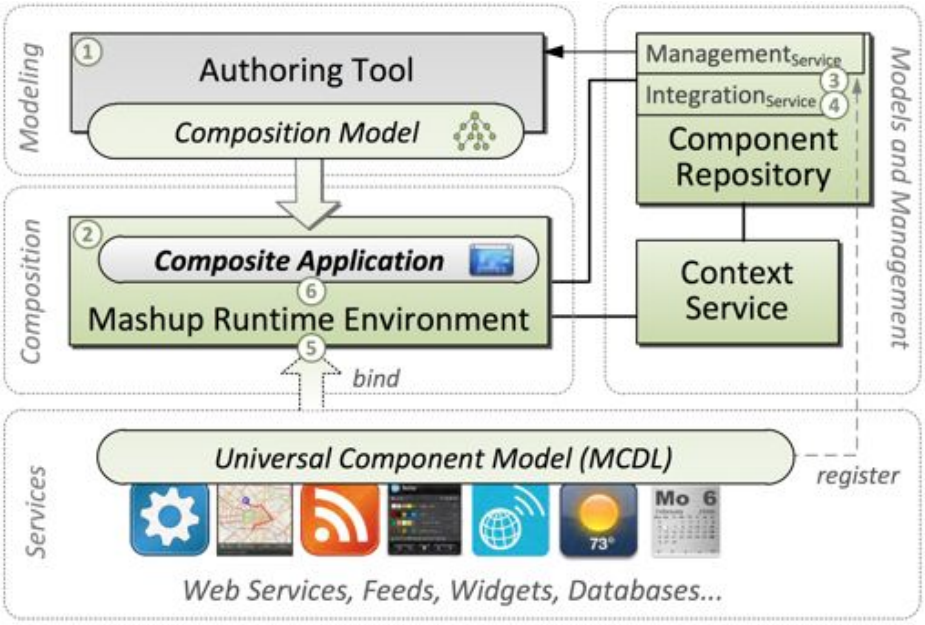

Abbildung 2.8: CRUISe - Architekturübersicht [Pie09]

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

In CRUISe verwaltet das *Component Repository* (CoRe) die Komponentenbeschreibungen und den jeweils zugehörigen Quellcode der Komponenten. Dabei bietet es dem Komponentenentwickler im Wesentlichen die folgenden Verwaltungsfunktionalitäten an: Hinzufügen, Entfernen und Aktualisieren der Komponentenbeschreibung und des ausführbaren Codes. Der Dienstnutzer kann mit Hilfe des CoRe nach Instanzdokumenten von Komponenten (Bindings) suchen - z. B. anhand von Schlüsselworten oder IDs. Demzufolge kommuniziert das Component Repository zur Laufzeit mit dem Integration Manager und zur Entwicklungszeit mit dem Autorenwerkzeug. Die Mashup Component Description Ontology (MCDO) wird intern vom CoRe für die Beschreibung von Komponenten genutzt. Somit wird eine transparente Transformation von MCDL hin zu MCDO notwendig. Um dies zu ermöglichen existieren definierte Basisontologien, welche für Meta-Elemente der MCDL passende Konzepte spezifiziert, und Domänenontologien, welche bspw. Komponentenklassen repräsentieren [Rad11]. Sowohl die MCDL, als auch die MCDO-Beschreibung werden in einer Datenbank persistent gespeichert.
Bevor Komponenten integriert werden, durchlaufen sie einen kontextunabhängigen *Matching*- sowie einen kontextabhängigen *Ranking*-Prozess [Mei11]. Beim *Matching* wird die Zielbeschreibung der gesuchten Komponente mit den vorhanden Komponenten im CoRe abgeglichen. Das Ergebnis dieses Prozesses ist eine gewichtete Liste von möglichen Kandidaten. Diese werden beim *Ranking* anhand nicht-funktionaler Eigenschaften / (Kontext-) Parametern sortiert. Dies geschieht auf Basis von Regeln, die auf alle in Frage kommenden Komponenten angewendet werden. Ein Beispiel einer Regel ist die *OccuranceRole*, welche eine binäre Entscheidung (bspw. die Unterstützung einer bestimmten Sprache) über die Erfüllung einer Bedingung ausdrückt. Abschließend wird die passendste Komponente, i. d. R. die Erstplatzierte, ausgewählt und integriert.
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 Instanziierung, 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.

- Der **Event Broker** ermöglicht die ereignisbasierte Kommunikation der lose gekoppelten Komponenten. Dabei kommunizieren Ereignisse mit Operationen über typisierte Kanäle nach dem Publish-Subscribe-Paradigma. Zu den drei verschiedenen Kanalarten zählen der *PropertyLink*, verantwortlich für die Synchronisation zwischen Eigenschaften, der *BackLink*, verantwortlich für die Datenanfrage und -lieferung zwischen einem Ereignis und einer Operation sowie der einfache *Link*, welcher für die pushbasierte Publikation zwischen Ereignissen und Operationen oder Eigenschaften verantwortlich ist.

- Für die Initialisierung der eben definierten Managerbestandteile ist der **Application Manager** verantwortlich, welcher den Rahmen der CRUISe Client Runtime bildet und eine Sicherung des Anwendungszustandes 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 notwendig, um die Fortführung der migrierten Anwendungskomponente im Kontext des Zielgerätes 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

sichtbaren, u. U. sehr grobgranular strukturierten, Zustand der Komponente ermöglichen. Ein möglichst geringer Entwicklungsaufwand für den Komponentenentwickler setzt voraus, dass der Zustand einer Komponente automatisch und so feingranular wie möglich erfasst wird - entweder von der Runtime Environment oder der Komponente selbst.

- 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.

Im Fokus dieser Arbeit steht die Extraktion des Zustandes einer UI-Komponente eines Ausgangsgerätes, die Übertragung dieser Information an den Server sowie die anschließende Injektion in eine andere, kompatible UI-Komponente eines Zielgerätes. Des Weiteren wird in diesem Zusammenhang ein geeignetes Zustandsmodell zur Serialisierung der Zustandsdaten entwickelt.

2.5 Zusammenfassung

Zu Beginn dieses Grundlagenabschnittes wurde der Begriff der Softwarekomponente geklärt sowie Anforderungen an diese näher spezifiziert. Danach wurde der Begriff des Komponentenmodells definiert und die Umsetzung dieses Konzeptes in CRUISe erläutert. Anschließend wurde die Architektur von CRUISe überblicksweise beschrieben. In diesem Zusammenhang wurde auf das Kompositionsmodell, die Laufzeitumgebung, das Komponentenrepositorium sowie auf derzeitige Defizite dieses Forschungsprojekt in Bezug auf einen UI-Migrationsprozesses eingegangen. Dabei stellte sich heraus, dass das Komponentenmodell von CRUISe dem Dienstanbieter einen großen Freiraum bei der Entwicklung seiner Komponenten lässt, aber für eine genau Feststellung des aktuellen Zustandes einer UI-Komponente demnach nicht immer ausreicht. Des Weiteren wurde festgestellt, dass keine Methoden für eine Zustandsextraktion, -serialisierung oder -injektion existieren und ein plattform- und technologieneutrales Zustandsmodell fehlt.
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 CRUISe 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

Nachdem im vorherigen Kapitel grundlegende Begriffe und Technologien definiert und erläutert wurden, werden in diesem Kapitel Vorgehensweisen für die Zustandserfassung, -extraktion, -serialisierung und -injektion näher betrachtet. Wie in Abschnitt 2.2.1 angeführt, ist die Zustandsübertragung im Kontext einer UI-Migration ein wichtiger Bestandteil, denn er ermöglicht es dem Nutzer unterbrechungsfrei auf dem Zielgerät genau an der Stelle weiterzuarbeiten, an der er auf dem Ausgangsgerät aufgehört hat. Aufgrund der Tatsache, dass die Menge aller Komponentenzustände anwendungsspezifisch strukturiert und repräsentiert ist, existiert derzeitig keine generische Lösung für die Erfassung des Komponentenzustandes. Des Weiteren erschwert der Austausch einer UI-Komponente während des Migrationsprozesses zusätzlich den Vorgang, da eine Desktop-Komponente andere Properties beinhalten könnte, als eine Smartphone-Komponente. Dieser Fall tritt zum Beispiel dann ein, wenn die Smartphone-Komponente zusätzliche Properties für die Erfassung des Kontextes besitzt.
Dieser Abschnitt soll den aktuellen Stand der Forschung und Technik, im Zusammenhang mit den soeben beschriebenen Herausforderungen der UI-Migration, darstellen. Dazu wird zunächst das Forschungsprojekt **CRUISe**, in Hinblick auf den Austauschvorgang von Komponenten, näher untersucht. In diesem Zusammenhang werden Isolationsmechanismen vorgestellt, welche beispielsweise verhindern, dass UI-Komponenten während des Migrationsprozesses einen inkonsistenten Zustand erreichen. Des Weiteren wird der mit dem Austauschvorgang verbundene Zustandstransfer beschrieben. Im letzten Teil des CRUISe-Abschnitts wird abschließend auf das Mapping von Zustandsvariablen eingegangen, welches für die korrekte Wertzuweisung zweier semantisch gleicher aber syntaktisch unterschiedlicher Komponenten verantwortlich ist.
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 Zustandserfassung durch Properties und illustriert die Verwendung des Entwurfsmusters *Memento*.
Des Weiteren sollen außerdem der Aufbau sowie die Arbeitsabläufe und Funktionen der Middleware **OPEN** untersucht werden. Hierbei wird der Fokus ebenfalls auf die Extraktion und Serialisierung des Komponentenzustandes gelegt. Zunächst wird in diesem Zusammenhang die Architektur der OPEN-Plattform im Überblick beschrieben. Anschließend wird auf die Migration von Komponenten in OPEN eingegangen und zum Abschluss dieses Abschnittes werden Herausforderungen beschrieben, welche bei der Serialisierung von JavaScript-Variablen auftreten, sowie passende Lösungsansätze, wie sie in OPEN implementiert sind, betrachtet.
Abschnitt 3.4 befasst sich mit dem Forschungsprojekt **COSMOD** und beschreibt in diesem Kontext eine Möglichkeit zur Generierung von Zustandsgraphen.

Der vorletzte Abschnitt des 3. Kapitels befasst sich mit **DOM-Mutation-Events**, welche Veränderungen innerhalb eines DOM-Baumes signalisieren. In diesem Zusammenhang soll untersucht werden, ob diese Events hilfreich bei der Wiederherstellung des Komponentenzustandes sind.
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

In Abschnitt 2.4 wurde die Architektur der CRUISe Thin-Server-Runtime vorgestellt. Eine ihrer Funktionen ist der Komponentenaustausch, welche im Rahmen dieses Abschnittes näher untersucht werden soll. Dieser dynamische Austauschprozess wurde gewählt, weil er elementare Methoden zur Verfügung stellt, die im Anbetracht der Untersuchung der UI-Migration von Interesse sind. Dazu zählen zum Beispiel das Pausieren von aktiven UI-Komponenten, welche ausgetauscht werden sollen, und die Erfassung und Extraktion des Komponentenzustandes der Ausgangskomponente sowie die Injektion dieses Zustandes in die Zielkomponente. Dies findet im Kontext eines Gerätes statt und beschränkt sich beim Austausch auf Komponenten, welche der selben Klasse der zugrundeliegenden Klassifikation angehören und somit bereits bekannte Schnittstellen und Zustandsräume verwenden [Rad10].
Die folgenden Abschnitte stellen wichtige Konzepte vor, die einen reibungslosen Komponentenaustausch ermöglichen. Dabei wird zuerst auf die Sicherstellung der Isolation eingegangen, der Ablauf des Komponentenaustausches beschrieben und abschließend wird der Zustandstransfer sowie das Mapping von Zustandsvariablen näher betrachtet.

3.1.1 Sicherstellung der Isolation

Ein wichtiger Aspekt, welcher sowohl für den Komponentenaustausch als auch für die UI-Migration essentiell ist, ist die Isolation. Damit ist gemeint, dass die auszutauschende Komponente während des Prozesses keinen inkonsistenten Zustand erreicht. Um dies gewährleisten zu können, kommt das Adapter-Konzept [Gam+95] zum Einsatz. Bei diesem Entwurfsmuster werden die Komponenteninstanzen umschlossen und ihre Schnittstellen auf die Spezifikationen gemäß dem Kompositionsmodell abgebildet. Dabei sichern *Wrapper* auf syntaktischer Ebene die Schnittstellenkompatibilität von Komponenten und repräsentieren diese. Dieses Vorgehen ermöglicht es, die auszutauschende Instanz problemlos von der Kommunikation der Anwendung zu entkoppeln und ist in Anlehnung an das Entwurfsmuster *Proxy* [Gam+95] implementiert [Rad11]. Demnach sind die *Wrapper / Proxies* das Bindeglied zwischen Komponenten und dem *Event Broker*. Aus Sicht einer Komponente nehmen sie die Rolle des Event Brokers ein und repräsentieren für diesen wiederum je eine Komponente. Dies ermöglicht es beispielsweise ausgehende Nachrichten um die Komponenten-ID zu erweitern. Des Weiteren kann der Proxy im Fall des Komponentenaustausches blockiert werden, sodass eingehende und ausgehende Nachrichten von ihm in eine Warteschlange eingereiht werden. Demzufolge gehen keine Nachrichten verloren, da

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

Unter Voraussetzung, dass die komposite Webanwendung bereits vollständig instantiiert ist, beschreibt dieser Abschnitt den Austausch einer Komponente, welche aufgrund eines Programmierfehlers abgestürzt ist. Zu Beginn dieses Austauschprozesses steht der in Abschnitt 2.4.1 beschriebene Discovery-Prozess von Komponenten, der für die Suche nach Alternativen die zu diesem Zeitpunkt integrierte Komponente als Ausgangspunkt ansieht. Demnach wird sie in eine Blacklist hinzugefügt. So wird sie von der Suche ausgeschlossen und kann nicht versehentlich ein weiteres Mal integriert werden. Um die Atomarität und Isolation der Aktion sicherzustellen sowie den Zustandstransfer zwischen alter und neuer Komponente zu ermöglichen, wurde ein Verfahren entwickelt, welches sich an den Ansätzen von [IFM08] und [GM05] orientiert. Dabei wird die Nutzung eines Stellvertreters, der für die Sicherung des Komponentenzustandes zu Beginn und der Isolation während des Austauschprozesses zuständig ist, vorgesehen. Für den Austausch kommt ein erweitertes Phasen-Commit-Protokoll zum Einsatz, deren einzelne Stufen im Folgenden näher erläutert werden [Rad11]:

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.
   2. Anschließend wird die Methode *prepare()* von Komponente A aufgerufen und signalisiert dieser den Austausch, um ihr die Möglichkeit zu geben ausstehende innere Berechnungen oder Transaktionen abzuschließen. Dies gewährleistet das Erreichen eines sicheren Zustandes.
   3. Komponente A signalisiert durch ein entsprechendes Event den Abschluss der Aktionen. Es kann ebenfalls zur Anzeige von Fehlern oder zu einem Timeout kommen.

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.
   2. Anschließend wird die Komponente durch den Aufruf von *disable()* deaktiviert.
   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.
   6. Im nächsten Schritt werden alle Verweise des Wrappers von Komponente A auf den Wrapper der Komponente B gesetzt. Des Weiteren wird die Liste der vorgehaltenen Nachrichten von dem Wrapper der Komponente A auf den anderen Wrapper übertragen.
   7. Komponente B wird initialisiert.
   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.
   10. Nachdem die Methode *dispose()*, welche alle komponenteninternen Ressourcen freigibt, von Komponente A ausgeführt wurde und der Destruktor von Komponente A aufgerufen wurde, wird der Wrapper der Komponente ebenfalls entfernt.

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.

   Instanziierungsfehler 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 Adaptions- und Kommunikationssperren gelöst.

1. Die Adaptionssperre des Adaption Managers wird gelöst und die initiale Blockierung des Wrappers von Komponente B wird aufgehoben. Somit können nun alle alle zwischengespeicherten Nachrichten an die Komponente weitergeleitet werden.

Werden diese Schritte erfolgreich abgeschlossen ist der Komponentenaustausch vollzogen und Komponente B besitzt den gleichen öffentlichen Zustand wie die Ausgangskomponente A. Es ist zu erkennen, dass dieser Algorithmus prinzipiell eine gute Grundlage für den UI-Migrationsprozess bereitstellt. Der folgende Abschnitt wird den Zustandstransfer einer Komponente auf eine andere Komponente genauer betrachten.

3.1.3 Transfer des Kompontenzustandes

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:

1. Zunächst werden alle nicht-transienten Eigenschaften von Komponente A ermittelt. Dies geschieht durch die Analyse der entsprechenden MCDL, welche entweder lokal gehalten oder vom CoRe abgerufen werden kann.

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

3. Schließlich wird der Zustand von Komponente B durch den Aufruf der Methode *setProperty(key)* für alle gesicherten Eigenschaften gesetzt. Properties, welche als *transient* gekennzeichnet wurden, bekommen die Initialbelegung aus dem Kompositionsmodell zugewiesen. Sollten diese nicht definiert sein, werden die Standardwerte aus der MCDL verwendet.

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 Inkonsistenzen 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 Variablenbezeichnungen besitzen. Das daraus resultierende Problem ist die Frage, mit welchen Mitteln festgestellt werden kann, welcher Eigenschaftswert der Ursprungskomponente 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 Matching**s 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:

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

Listing 3.1: Semantische Typisierung

Wie in diesem Codeausschnitt dargestellt, wird die Eigenschaft *„Standort“* mit dem Ontologiekonzept *„Location“* typisiert. Demzufolge verweist der Typ vvo:Location der Property Standort auf das Konzept Location einer externen Ontologie, welche über den Namespace vvo definiert ist. Somit ermöglicht diese Technologie dem *Mediator*, über mehrstufige Transformationen der Daten, eine korrekte Wertzuweisung zwischen zwei syntaktisch unterschiedlichen, aber semantisch gleichen Properties zweier UI-Komponenten. Grundlage hierfür sind die bereits erwähnten *gemeinsamen Domänenmodelle* beider Komponenten sowie die Nutzung einheitlicher, vordefinierter Ontologien. Aufgrund der Tatsache, dass eine detaillierte Beschreibung der Funktionsweise dieser Technologie nicht Teil der Aufgabenstellung ist, wird an dieser Stelle für weiterführende Informationen auf [Rad11] verwiesen.

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-

stellenkompatibilität der beteiligten Komponenten. Weiterhin sichert das Phasen-Commit-Protokoll die Atomarität sowie die Konsistenz der Aktion und erlaubt im Fehlerfall einen Rollback. Ein weiterer positiver Aspekt ist das verwendete *Semantic Matching*, welches das Mapping unterschiedlich benannter Zustandsvariablen zweier Komponenten übernimmt.
Im Gegensatz zu diesen positiven Aspekten weist der Zustandstransfer Defizite auf, die im Laufe dieser Arbeit verbessert werden sollen. Beispielsweise werden Variablen, welche nicht als Teil der Schnittstelle in der MCDL definiert wurden, oder einfach beim Komponentenentwickler in Vergessenheit geraten sind, bei dem Austauschprozess nicht erfasst. Des Weiteren entsteht durch die Notwendigkeit einer manuellen Implementierung von Methoden, welche den Wert jeder einzelnen Zustandsvariable auslesen, ein erhöhter Entwicklungsaufwand für den Komponentenentwickler. An dieser Stelle soll die vorliegende Forschungsarbeit ansetzen und Verbesserungsvorschläge sowie mögliche Algorithmen für eine automatisierte Zustandserfassung vorstellen.
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 Atomariät 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 Zustand-

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

Um einen unterbrechungsfreien Service-Austauschprozess in CoBRA zu gewährleisten, wurde das *protection proxy pattern* [Gam+95] implementiert. Der Proxy ist dafür verantwortlich während der Laufzeit jeden Service eines Dienstanbieters in der CoBRA-Umgebung zu replizieren und alle Anfragen entgegenzunehmen. Der folgende Abschnitt soll dazu dienen ein Verständnis für die Notwendigkeit und die Arbeitsweise dieses Entwurfsmusters zu bekommen. Des Weiteren sollen die Unterschiede zu den Wrappern aus CRUISe verdeutlicht werden. Die nachfolgende Abbildung zeigt die Umsetzung des Entwurfsmusters in dem Forschungsprojekt CoBRA.

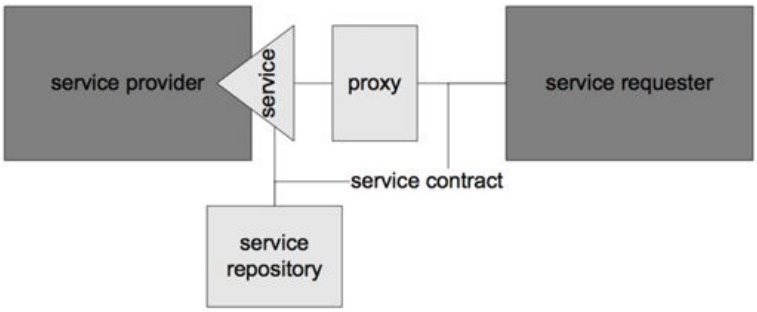

Abbildung 3.1: Protection proxy pattern in CoBRA [IFM08]

Wie in Abbildung 3.1 gezeigt, orientiert sich die Implementierung des Proxy-Entwurfmusters in CoBRA stark an dem Grundgerüst einer SOA-Architektur. Dabei hat ein Dienstnutzer (*service requester*) allerdings kein direkten Zugriff auf den *Service* eines Dienstanbieters (*service provider*) (welcher von einem Repositorium (*service repository*) bereitgestellt wird), sondern kommuniziert ausschließlich mit dem *Proxy*-Objekt. Der Grund für die Wahl dieser indirekten aber transparenten Kommunikation (*service contract*) ist die Notwendigkeit einer Schnittstelle, welche einen atomaren Austauschprozess ermöglichen kann. Sollte während des Austauschprozesses eine Anfrage an den auszutauschenden Service gestellt werden, wird diese von dem Proxy-Objekt geblockt, um zu verhindern, dass der Service einen inkonsistenten Zustand erreicht.
Abbildung 3.2 zeigt in Form eines Zeitdiagramms den gesamten Adaptionsvorgang, welcher in 3 Phasen aufgeteilt ist (*store phase*, *switch phase* sowie *restore phase*) und als nächstes näher betrachtet wird:
Wie zuvor beschrieben, werden während des gesamten Adaptionsprozesses (*Adaption process*) Anfragen durch den Service-Proxy geblockt (*blocked invokation*). Anders als bei CRUISe werden die Anfragen nicht gespeichert, um sie zu einem späteren Zeitpunkt erneut auszuführen. Dies würde bei einer Benutzereingabe zu einem Informationsverlust führen, da der Nutzer kein Feedback bekommt ob seine Eingabe angekommen ist oder nicht. Dies stellt somit ein großes Defizit dar.
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.
Die *switch phase* repräsentiert den Zeitintervall in dem der Service ausgetauscht wird. Sollte der Austausch unterbrochen werden oder erfolglos sein, stellt eine Rollback-Funktion den ursprünglichen Service wieder her - inklusive letztem Zustand. Nach einer erfolgreichen Integration des gewünschten Zielservices (B1a) wird das State-Container-Objekt, welches den Komponentenzustand des Ausgangsservices beinhaltet, in den Service injiziert (*restore*).
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 Adaptionsprozess abgeschlossen und konnte sowohl atomar, als auch transparent durchgeführt werden.
Im folgenden Abschnitt wird das verwendete *Memento*-Entwurfsmuster [Gam+95] näher beschrieben.

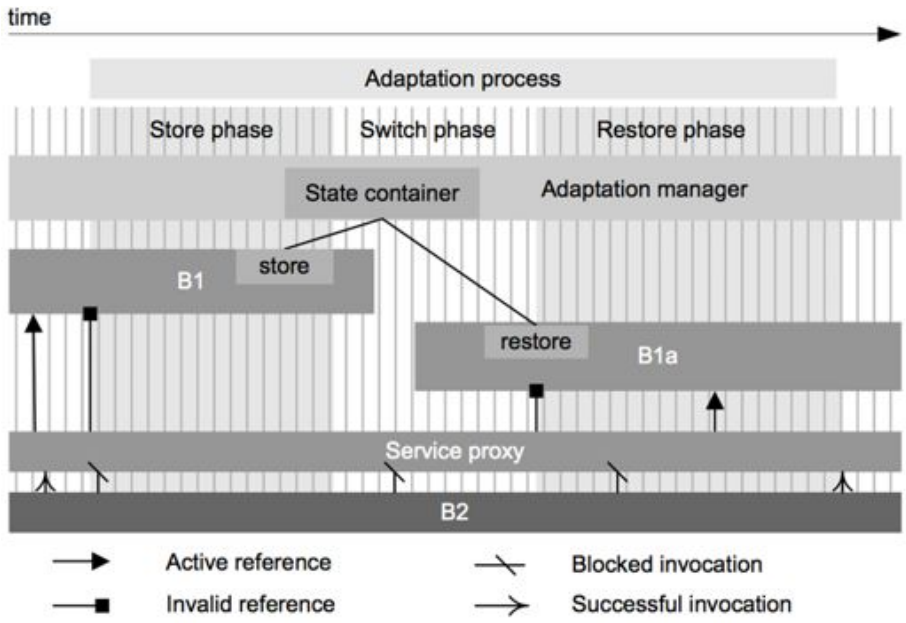

Abbildung 3.2: Adaptionsprozess in CoBRA [IFM08]

3.2.2 Zustandserhaltung mit Hilfe des Memento-Patterns

Aufgrund der Tatsache, dass der Anwender möglichst unterbrechungsfrei und ohne Wiederholung vorangegangener Arbeitsschritte, nach dem Austausch bzw. der Migration einer Komponente, weiterarbeiten will, ist es notwendig, dass eine ausgetauschte / migrierte Komponente den selben Zustand besitzt wie die Ausgangskomponente. Dazu wird mit Hilfe des *Memento*-Entwurfsmuster [Gam+95] der interne Komponentenzustand zunächst in ein *Memento-Objekt* gesichert und nach der Übertragung auf die Zielkomponente wiederhergestellt. Dieses Vorgehen eignet sich gut für Black-Box-Komponenten, da es für andere Komponenten nicht sichtbar ist und somit keine Implementationsdetails veröffentlicht.

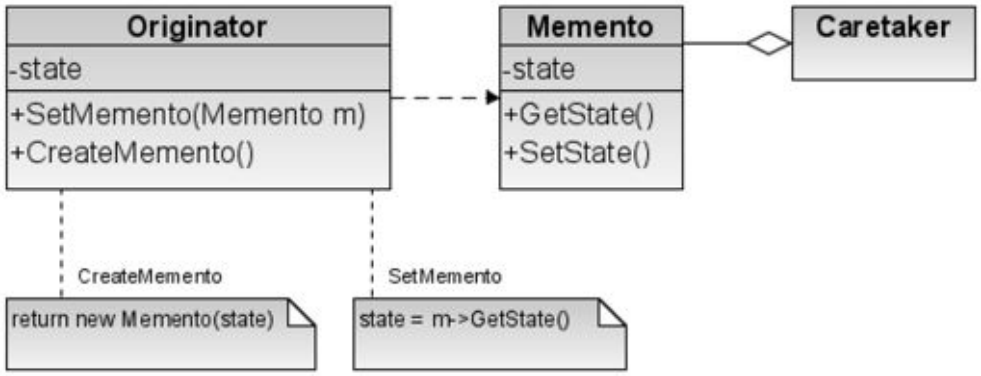

Abbildung 3.3: Struktur des Memento-Patterns [Gam+95]

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.

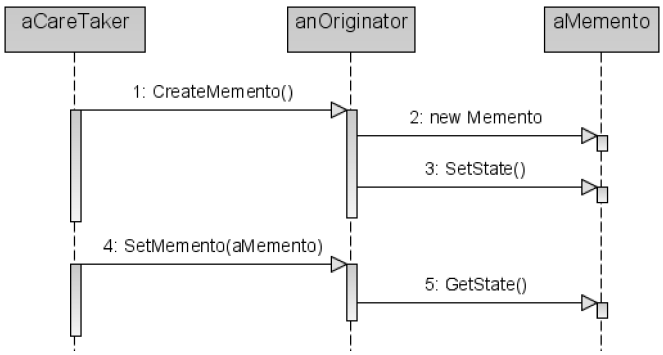

Abbildung 3.4: Sequenzdiagramm des Memento-Patterns [Gam+95]

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

Zusammenfassend kann festgestellt werden, dass das *Protection Proxy Pattern* und das *Memento Pattern* für einen UI-Migrationsprozess hilfreich sein könnten, weil sie bei der Einhaltung verschiedener Eigenschaften, wie bspw. Atomarität und Zustandserhaltung, angewandt werden könnten.
Dieser Abschnitt hat sich mit dem Serviceaustausch in CoBRA befasst und ist dabei vor allem auf die Verwendung der beiden Entwurfsmuster *Proxy* und *Memento* eingegangen. Der folgende Abschnitt 3.3 wird das Forschungsprojekt OPEN näher betrachten, wobei zunächst ein Überblick über die Gesamtarchitektur und das Konzept gegeben wird, um anschließend den Fokus wieder auf die Zustandserfassung sowie -extraktion und -injektion zu legen. Des Weiteren werden in diesem Abschnitt Herausforderungen beschrieben, welche bei der Serialisierung von JavaScript-Variablen auftreten und passende Lösungsansätze, wie sie in OPEN implementiert wurden, präsentiert.

3.3 Die OPEN Migration Service Platform

Die *OPEN Migration Service Platform (MSP)* ist eine Middleware, welche entwickelt wurde, um die Migration von Web-Anwendungen zwischen verschiedenen Endgeräten in einem Netzwerk (Heimnetzwerk, Firmennetzwerk oder Internet) zu regeln [Nic+10]. Dabei wird der Fokus auf HTML/JavaScript-basierte *Rich Internet Anwendungen* gelegt, welche zwischen heterogenen Plattformen migriert werden können [Pat11]. Des Weiteren ist es möglich diese RIAs auf mehreren Geräten zu verteilen. Beispielsweise kann im Kontext eines Rennspiels ein Smartphone als Eingabegerät genutzt werden und der TV als Ausgabemonitor der Web-Anwendung. Demzufolge wird der Fokus bei der OPEN-Plattform auf **verteilte HTML/JavaScript-Web-Anwendungen** gelegt und nicht auf komposite Web-Anwendungen, wie beispielsweise in CRUISe.
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-Plattform stellt ein Framework bereit, welches sowohl eine totale Migration, als auch eine partielle Migration ermöglicht [Pat11]. Des Weiteren stellt sie der Web-Anwendung die notwendigen Mechanismen für die Adaption an das Zielgerät zur Verfügung. Dazu zählen Anpassungen der Netzwerkverbindungen und der Benutzerschnittstellen sowie die Wiederherstellung des vorherigen Anwendungszustandes.
Wie in Abbildung 3.5 gezeigt, basiert OPEN auf einer *Client-Server-Architektur*, bei der ein OPEN-Server ausschließlich einen oder mehrere OPEN-Clients sieht, ohne dabei Implementierungsdetails der dahinterliegenden Web-Anwendung zu kennen. Somit sieht der OPEN-Server zum Beispiel nicht, dass der OPEN-Client aus einem *Terminal* und einem *Application Server* besteht, wie in Abbildung 3.5 dargestellt. Dabei ist es möglich, dass ein OPEN-Server auf jedem beliebigen Gerät ausgeführt werden kann, welches die notwendigen Hardwarespezifikationen erfüllt und von jedem anderen OPEN-Client erreichbar ist.
Der Client beinhaltet die Benutzerschnittstelle sowie die Anwendungslogik der Web-Anwendung. Des Weiteren besitzen die Clients *OPEN-Adaptoren*, welche im nächsten Abschnitt näher beschrieben werden. Diese Adaptoren stellen verschiedene Migrationsfunktionalitäten bereit wie beispielsweise das Pausieren der zu migrierenden Anwendungen oder das Auslösen der UI-Migration. Sobald der Migrationsprozess ausgelöst wurde, ist es Aufgabe des OPEN-Servers die Migration zu koordinieren und durchzuführen. Sämtliche Teilprozesse, welche in Abschnitt 3.3.2 näher beschrieben werden, werden von dem Server abgearbeitet. Des Weiteren kann der Server den Clients anweisen bestimmte Methoden auszuführen um beispielsweise den aktuellen Anwendungszustand bereitzustellen.

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-

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.

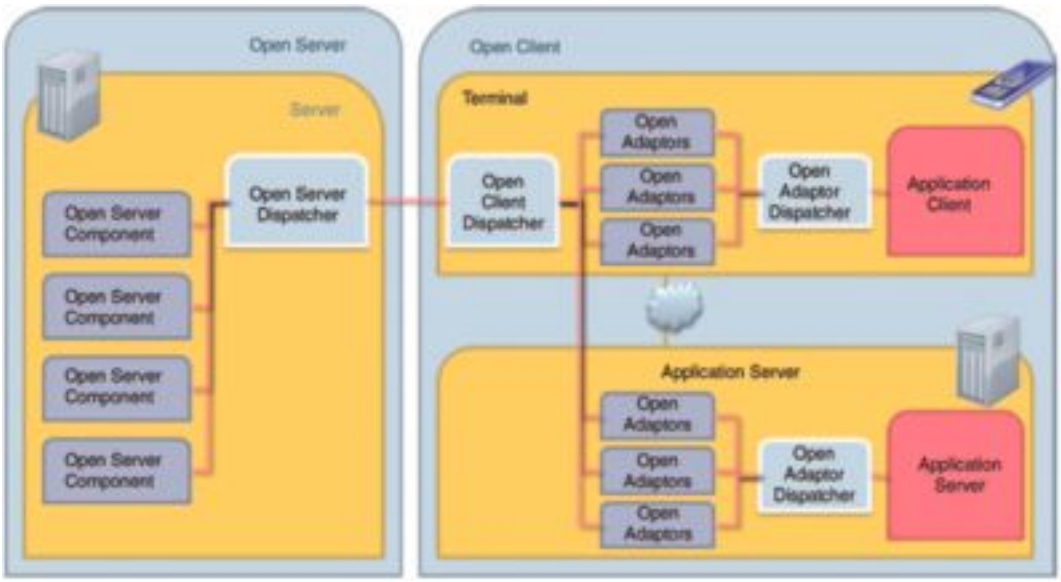

Abbildung 3.5: OPEN - Architektur [Pat11]

Neben diesen zwei Adaptoren 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.
Eine Anwendung ist migrationsfähig, sobald sie die OPEN-Adaptoren implementiert. Zusätzlich zu den Migrationsfunktionalitäten stellen die OPEN-Adaptoren den OPEN-Clients die Schnittstelle zum OPEN-Server bereit. Solange diese Schnittstelle zwischen dem Server und den Clients vollständig berücksichtigt und richtig implementiert wurde, kann der Anwendungsentwickler die OPEN-Adaptoren überschreiben und somit zusätzliche, interne Funktionalitäten implementieren.

Die OPEN-Dispatcher

Die modulare Strukturierung der OPEN-Plattform ermöglicht zum einen die Verteilung der OPEN-Server-Funktionalitäten über mehrere Serverkomponenten (*Open Server Component*) und zum anderen die Verteilung der OPEN-Clients auf mehrere Endgeräte. Dabei stellt sich die Frage, woher ein Client die Information bekommt welche Komponente des OPEN-Servers adressiert werden muss, sobald ein bestimmter Teil der Schnittstelle aufgerufen wurde.
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

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

- Der **OPEN-Server Dispatcher** existiert für jede OPEN-Anwendung nur ein einziges mal und repräsentiert den Endpunkt des gesamten OPEN-Servers. Demnach wird jede eingehende Anfrage eines OPEN-Clients durch diesen Dispatcher server-intern an die passende Serverkomponente weitergeleitet. Analog dazu wird das Versenden jeder Nachricht eines OPEN-Servers durch diesen Dispatcher koordiniert.

- Jeder OPEN-Client besitzt einen **OPEN-Client Dispatcher**. Dieser ist der Kontaktpunkt für eingehende Nachrichten eines OPEN-Servers, welche an die Anwendung bzw. die Adaptoren des OPEN-Clients gesendet wurden. Ähnlich wie bei dem OPEN-Server Dispatcher werden ausgehende Nachrichten eines OPEN-Clients von seinem Dispatcher versendet. Im Falle einer Aufteilung des OPEN-Clients (vgl. Abbildung 3.5) ist der OPEN-Client Dispatcher für die Weiterleitung der Nachricht an den adressierten Endpunkt zuständig.

- 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

Der Ansatz der UI-Migration, welcher im Forschungsprojekt OPEN verfolgt wird, zielt darauf ab eine möglichst allgemeine Lösung für Web-Anwendungen, welche in (X)HTML, CSS und JavaScript programmiert wurden, bereitzustellen. Des Weiteren werden Anwendungen unterstützt, die auf Programmiersprachen wie JSP, PHP und ASP aufbauen. Dieser Ansatz ist dafür ausgelegt den Zustand der aktuellen Web-Seite, auf welche der Anwender zum Zeitpunkt der Migration zugreift, zu erfassen und an das Zielgerät anzupassen. Dabei ist es vollkommen egal, mit welcher Entwicklungsumgebung die Web-Anwendung programmiert wurde. Demzufolge ermöglicht es OPEN Anwendungen zu migrieren, auch wenn diese keine speziellen Migrations-Methoden besitzen. Der folgende Abschnitt beschreibt den Ablauf einer UI-Migration in OPEN und geht dabei speziell auf die Funktionsweise der beteiligten Komponenten ein.

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 unangetastet 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 ist.
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 ausließt (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

Wie bereits in den vorangegangenen Abschnitten erwähnt, ist es im Rahmen der OPEN-Plattform möglich Web-Seiten (bzw. Teile von diesen) zu migrieren. Um den Migrations-Prozess leicht verständlich darzustellen, wurde in [Pat11] ein bekanntes Computerspiel als Browser-Game, auf Basis von (X)HTML und JavaScript, umgesetzt.

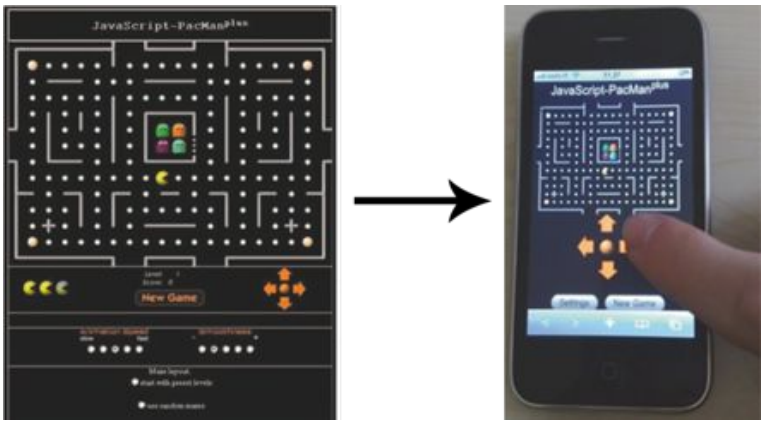

Abbildung 3.6: OPEN - Migration eines PC-Spiels [Pat11]

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

die Smartphone-Variante. Dies ist auf die geringere Bildschirmgröße und den somit beschränkten Ein- und Ausgabemöglichkeiten des mobilen Gerätes zurückzuführen. Beispielsweise werden die Grafik-Optionen („Animation Speed“ und „Smoothness“) sowie die Anzeige der verbleibenden Leben in der mobilen Version (rechte Bildhälfte) unter dem Punkt „Settings“ zusammengefasst. Demzufolge muss eine Anpassung der Benutzerschnittstelle an das jeweilige Gerät erfolgen. Aufgrund der Tatsache, dass der Adaptionsvorgang für diese Forschungsarbeit nicht relevant ist, wird darauf nicht näher eingegangen. Des Weiteren ist die Zustandserhaltung während der UI-Migration eine weitere wichtige Voraussetzung, damit der Anwender in Bezug auf dieses Beispiel an jener Stelle weiterspielen kann, an der auf dem Ausgangsgerät aufgehört hatte. Konkret ist es demnach notwendig, den Standort des PacMan und der Geister, die bereits eingesammelten Punkte, das derzeitige Level sowie die Anzahl der verbleibenden Leben zu speichern. Dies wird mit Hilfe von **globalen JavaScript-Variablen** sowie **Objekten**, welche von Funktionen manipuliert werden können, realisiert. Somit ist es möglich, dass jedes Objekt und jede Variable des JavaScript-Codes, als eine Eigenschaft des globalen Web-Seiten-Fensters angesehen werden kann. Für die eigentliche Erfassung des Zustandes, ist in OPEN die Methode *saveState()* zuständig, welche über jede Eigenschaft des globalen Browser-Fensters iteriert und jedes Element in ein jSON[2]-Objekt (siehe Abschnitt 3.3.3) speichert. Alle Objekte zusammen bilden den Zustand der Komponente und werden zusammengefasst in einem Array gespeichert. Dieses Array wird an den Migration Server geschickt, welcher zu gegebenem Zeitpunkt, mit Hilfe der Methode *restoreState()*, die Injektion der Zustandsdaten in die Zielkomponente einleitet. Dabei wird von jedem jSON-Objekt des Arrays der Name sowie dessen zugehöriger Wert ausgelesen. Mit diesen Daten werden die Eigenschaften der neuen Web-Seite, falls vorhanden, auf den gewünschten Zustand aktualisiert. Falls eine Eigenschaft nicht zugewiesen werden kann, wird sie verworfen und geht somit für zukünftige Migrationen verloren.

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.
Während der Entwicklung des Algorithmus, welcher den aktuellen Zustand der Web-Anwendung extrahieren sollte, traten Probleme auf, die sich mit einem normalen jSON-Serialisierer nicht lösen ließen. Dazu gehörten unter anderem zyklische Referenzen, nicht-nummerische Eigenschaften von Arrays und Timer [Pat11]. Um diese Herausforderungen zu lösen, wurde die JavaScript Library *dojoX*[3] genutzt und angepasst, um die eben beschriebenen Objekte serialisieren zu können. Die folgenden Abschnitte geben einen Überblick über die Lösungsansätze für jedes einzelne Teilproblem [Pat11].

[2] JavaScript Object Notation
[3] https://dojotoolkit.org

Globale und BOM-Variablen

Jedes Objekt bzw. jede Variable, die im JavaScript-Code definiert wurde, kann als eine Eigenschaft des globalen Fenster-Objektes (Browser-Fenster) angesehen werden. Um alle diese Werte erfassen zu können, wird in OPEN eine *for...in*-Schleife genutzt, welche die Eigenschaften jedes Objektes ausließt und abspeichert. Ein Teil dieser Informationen, sollte allerdings nicht in die finalen Zustandsdaten miteinbezogen werden - die Eigenschaften des *Browser Object Model (BOM)*. Diese Eigenschaften beinhalten Informationen wie beispielsweise die Adresse der aktuell geladenen Seite oder die Browser-Historie. Aufgrund der Tatsache, dass diese Informationen browserspezifisch sind, müssen sie aus den finalen Zustandsinformationen entfernt werden. Dies wird mit Hilfe von *Filtern* realisiert, welche die BOM-Variablen jedes Browsers aus den finalen Zustandsinformationen herauslöschen.

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

der eigentliche Wert. Demzufolge wird ist der Wert der Eigenschaft „sohn“ eine Referenz. Dabei wird ein Pfad-basierter Referenzierungsalgorithmus angewandt, welcher den Wert eines Objektes anhand seiner Position innerhalb der Objektstruktur ermittelt. Auf diese Weise wird für den Wert der Eigenschaft „sohn“ ein *Pfad*, ausgehend vom Wurzelelement der Variable bis hin zum gewünschten Knoten, abgespeichert.

Timer

Timer werden von Entwicklern genutzt, um zum Beispiel einen Teil des Codes erst nach Ablauf einer gewissen Zeitspanne auszuführen. Dies wird bspw. mit den Methoden *setTimeout* (startet einen Countdown, welcher nach Ablauf der Zeit eine Aktion durchführt) oder *setInterval* (für wiederholtes Ausführen eines Codesegmentes nach Ablauf einer Zeitspanne) realisiert. Analog dazu werden *clearTimeout* und *clearInterval* für den Abbruch des laufenden Timers bzw. das Löschen einer zuvor festgelegten Zeitspanne verwendet. Im folgenden Beispiel stellt „function“ den auszuführenden Codeausschnitt dar, welcher nach Ablauf einer festgelegten Zeit in „ms“ ausgeführt werden soll.

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

Listing 3.4: Timer

Solche Timer können den Zustand einer Komponente auf zwei Arten beeinflussen. Zum einen ist es möglich, dass ein Timer zum Zeitpunkt der Migration gerade aktiv ist und zum anderen können Variablen eine Referenz auf diesen Timer besitzen. Aufgrund der Tatsache, dass es weder möglich ist den aktuellen Zustand eines Timers auszulesen noch eine Liste mit gerade aktiven Timern auszugeben, werden in OPEN die Methoden *setTimeout* und *setInterval* erweitert. Dabei wird eine globale Liste von Timern erstellt, welcher ein *Timer-Objekt* hinzugefügt wird, sobald eine der beiden Methoden aufgerufen wurde. Jedes dieser Timer-Objekte wird von einem „zentralen Timer“ aktualisiert. Dies wird mit Hilfe von regelmäßigen Aufrufen der Updatefunktion jedes aktiven Timers realisiert. Um eine korrekte Wiederherstellung der Timer nach Abschluss der UI-Migration zu gewährleisten, wird auf dem Zielgerät zunächst eine Liste der gespeicherten Timer erstellt. Anschließend wird der zentrale Timer sowie auch die mit ihm verbundenen abgespeicherten Timer neu gestartet.

Objekte, die ein Datum enthalten

Aufgrund der Tatsache, dass jSON ein Objekt, welches ein Datum beinhaltet, als ein void-Objekt serialisiert und die benutzte Library *dojoX* nicht in der Lage ist, das zuvor in ein ISO-UTC-formatierten String serialisierte Datums-Objekt, wieder zu deserialisieren, wird in OPEN das ISO-formatierte Datum in einem Objekt gekapselt und mit einem Wert versehen, welcher dem Deserialisierer signalisiert, dass es sich bei dem vorliegenden Objekt um ein Datums-Objekt handelt. Somit kann eine korrekte Wiederherstellung des Objektes gewährleistet werden.

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 berücksichtigt und erfasst, wenn das zu serialisierende Objekt ein assoziatives Array ist. Das folgende Codebeispiel soll diese Problematik verdeutlichen:

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

Listing 3.5: Dynamische Wertzuweisung

Für die Lösung dieses Problems wurde der Serialisierer der Library dojoX dahingehend erweitert, dass jedes Array in einem Objekt gekapselt wird, welches zusätzlich eine Liste mit allen dynamisch hinzugefügten Eigenschaften enthält. Diese Vorgehensweise 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 Erhaltung des Zustandes ist es wichtig, dass auch nach der Migration der Zugriff auf den DOM-Baum möglich ist. Um auf die einzelnen Elemente des DOM-Baums zuzugreifen 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:

```
1 var element = getElementById("myHtmlElement");
2 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 Zustandserhaltung abgespeichert. Andernfalls wird eine ID generiert und dem Objekt zugewiesen. Sollte das Objekt nicht Teil des DOM-Baums sein, wie beispielsweise die Variable „image“ des oberen Beispiels, so werden alle zur Verfügung stehenden Eigenschaften dieses Objektes mit Hilfe der JavaScript Library JsonML[4] ausgelesen und abgespeichert.

3.3.4 Zusammenfassung

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

[4] http://jsonml.org/

einer *Reverse Engineering*-Technik ermöglicht. Dabei werden während der Laufzeit *logische Beschreibungen* der zu migrierenden Web-Seite erzeugt, welche anschließend an die Zielplattform angepasst werden. Im darauffolgenden Schritt wird der aktuelle Komponentenzustand mit Hilfe von JavaScript-Methoden aus dem DOM-Baum der Ausgangs-Web-Seite extrahiert und in die, bereits angepassten, logischen Beschreibungen integriert. Schließlich wird aus diesen Beschreibungen die neue Web-Seite generiert und an das Zielgerät übertragen.
Aus diesem Forschungsprojekt konnten interessante Ansätze für die Zustandserfassung sowie dessen Extraktion und Injektion gewonnen werden. Dazu zählt z. B. die Idee, aus dem DOM-Baum der Ausgangs-Web-Seite notwendige Zustandsdaten zu extrahieren. Inwiefern dieses Vorgehen im Forschungsprojekt CRUISe umgesetzt werden kann, wird im folgenden Kapitel 4 diskutiert. Zunächst allerdings wird im nächsten Abschnitt das Projekt *COSMOD* vorgestellt, in dessen Fokus die Generierung eines Zustandsgraphen steht.

3.4 COSMOD: Generierung von Zustandsgraphen

In [SM09] beschreiben Rajiv Ranjan Suman und Rajib Mall die Arbeitsweise ihrer Methodik, welche es ermöglicht das Zustandsmodell aus einer Black-Box-Komponente zu extrahieren. Dies gelingt ihnen, indem sie die Änderungen einer Komponente aufzeichnen, welche beispielsweise dann auftreten, wenn andere Software-Komponenten Methoden bzw. Operationen der beobachteten Komponente aufrufen. Dabei wird vorausgesetzt, dass alle Methoden einer Komponente in einer Schnittstellenbeschreibung definiert sind. Die Entwickler haben ihre Methodik COSMOD[5] genannt und nutzen endliche Automaten, bestehend aus Zuständen, Transitionen / Zustandsübergängen und Aktionen (Berechnungen innerhalb eines Zustandes), für die Modellierung des Komponentenzustandes. Der folgende Abschnitt beschreibt überblicksweise den Vorgang, wie das Zustandsmodell aus einer Komponente extrahiert werden kann.

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

- Anschließend werden all diese Methoden m_{0u}, aus der soeben generierten Liste, zuerst im Anfangszustand Z_0 und anschließend in allen ermittelten Folgezuständen ausgeführt, wobei mögliche Eingabeparameter p_{0u} per Zufall generiert werden. Ein Zustandswechsel wird dabei mit Hilfe der zur Verfügung stehenden Methoden bestimmt. So wird jederzeit überprüft, welche Teilmenge aller Methoden m_{0u} 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_{0u} mit den Werten p_{0u} wird die Sequenz $m_{0u}(p_{0u})$ abgespeichert.

[5] „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.

Wie bereits beschrieben, werden ausgehend von dem Startzustand alle zur Verfügung stehenden Methoden mehrmals mit unterschiedlichen, zufallsgenerierten Eingabeparametern ausgeführt, bis ein vordefinierter Endzustand erreicht wird. Dabei ist es möglich, dass ein bereits entdeckter (**a**lter) Zustand (Z_a) ein weiteres Mal erreicht wird. In diesem Fall werden die verfügbaren Methoden von Z_a mit den Methoden von dem **n**euen Zustand Z_n verglichen. Unterscheiden sich die Methoden, so wird Z_n als ein neuer Zustand abgespeichert. Bei einer Übereinstimmung werden alle Methoden der beiden vermeintlich gleichen Zustände ausgeführt und beobachtet, ob unterschiedliche Folgezustände erreicht werden. Auf diesem Weg kann festgestellt werden, ob es sich tatsächlich um den gleichen Zustand. In diesem Fall würde der Zustand Z_n verworfen werden.
Für den beschriebenen Algorithmus wird in [SM09] eine polynomiale Zeitkomplexität von $T = O[k^2 * (alleZustaende^2 + alleZustaende * \sum_{i=1}^{k} T(m_i))]$ 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

Nachdem die vorangegangenen Abschnitte verschiedene Forschungsprojekte, wie *CoBRA* oder *OPEN*, in Bezug auf die Zustandserhaltung von UI-Komponenten untersucht haben, beschäftigt sich dieser Abschnitt mit einer Technologie, welche es ermöglicht sämtliche *Strukturveränderungen* innerhalb eines DOM-Baumes festzustellen. Diese **Mutation-Events** können neben strukturellen Veränderungen auch Benutzereingaben oder Netzwerkaktivitäten signalisieren [Ann12]. So ist es bspw. möglich jeden Mausklick auf ein Button oder jede Tastatureingabe in einem Textfeld zu registrieren. Aufgrund der Tatsache, dass nach dem Hinzufügen von *DOM-Mutation-Listenern* zu einem HTML-Dokument spürbare Performance-Einbrüche zu verzeichnen sind, wurden die ursprünglichen *Mutation-Events* durch **DOM-Mutation-Observer** ersetzt [Nic12]. Der wesentliche Unterschied zwischen beiden Technologien besteht darin, dass die *Mutation-Events* jede Änderung sofort signalisieren, wohingegen die *Mutation-Observer* Callback-Funktionen nutzen, welche es ermöglichen mehrere Veränderungen gebündelt zu erfassen.
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-

sache, dass Flash-Komponenten keine vergleichbare Schnittstellenspezifikation für den Zugriff auf deren Elemente bieten wie das **D**ocument **O**ject **M**odel, lässt sich ausschließlich der Kerngedanke dieses Ansatzes auf andere Programmiersprachen anwenden. **Die Erfassung *zustandsrelevanter* UI-Events zur Rekonstruierung des Komponentenzustandes** stellt eine Technik dar, welche in nahezu jeder Programmiersprache eingesetzt werden kann. Diese Behauptung resultiert aus der Tatsache, dass in allen gängigen Programmiersprachen *„Listener“* bzw. *„Observer“* zur Überwachung von UI-Elementen eingesetzt werden können.
Dieser Abschnitt hat eine Technologie vorgestellt, mit der es ermöglicht wird auftretende Strukturveränderungen innerhalb einer JavaScript-basierten Komponente zu registrieren. Dabei stellte sich heraus, dass der zugrundeliegende Kerngedanke dieses Ansatzes auf weitere Programmiersprachen übertragen werden könnte. Inwiefern sich diese Technik letztendlich für den Einsatz im Forschungsprojekt CRUISe eignet, wird in Abschnitt 4.2 des 4. Kapitels genauer untersucht. Im letzten Abschnitt dieses dritten Kapitels werden abschließend die gewonnen Erkenntnisse kompakt zusammengefasst und alle Besonderheiten der einzelnen Forschungsprojekte werden in diesem Zusammenhang noch einmal kurz hervorgehoben.

3.6 Zusammenfassung

In diesem Kapitel wurden ausgewählte Lösungsansätze und Konzepte untersucht, welche für den Prozess der UI-Migration nützlich sind. Dazu wurden zunächst Methoden für die Erfassung, Extraktion und Injektion des Komponentenzustandes untersucht und im Anschluss daran wurde überblicksweise beschrieben, welche Formate für die Serialisierung dieser Informationen genutzt werden können und wie ein Zustandsmodell durch Beobachtung der Methodenaufrufe generiert werden kann.
Zu Beginn dieses Kapitels wurde zunächst das Forschungsprojekt **CRUISe** auf eventuell vorhandene Methoden bzw. Werkzeuge untersucht, welche für die UI-Migration nützlich sind. Dazu wurde der Austausch von Komponenten näher betrachtet. In diesem Zusammenhang stellte sich heraus, dass sowohl die Sicherstellung der Isolation mit Hilfe von Wrappern / Proxies, als auch das verwendete Phasen-Commit-Protokoll eine gute Grundlage für die UI-Migration bieten. Der Zustandstransfer in CRUISe weist allerdings Defizite auf, welche im Laufe dieser Arbeit behoben werden sollen. Dazu könnte beispielsweise eine Erweiterung der bestehenden, automatisierten Zustandserfassung zählen, die dem Komponentenentwickler Entwicklungsaufwand erspart, indem sie den Zustand der zu migrierenden UI-Komponente so feingranular wie möglich erfasst und somit den Anforderungen der UI-Migration entspricht. Dies impliziert, dass es nicht ausreicht die vom Komponentenentwickler definierten Properties auszulesen.
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.
Im dritten Abschnitt dieses Kapitels wurde die Migrationsplattform **OPEN** im Überblick vorgestellt. Zu Beginn wurden die wesentlichen Bestandteile der Architektur beschrieben und im Anschluss wurde das Konzept vorgestellt, welches in OPEN verwendet wird, um UI-Komponenten zu migrieren. Dabei stellte sich heraus, dass in OPEN mit Hilfe des DOM-Baums einer (X)HTML-Seite sowie mit speziellen JavaScript-Methoden der Zustand der zu migrierenden UI-Komponente erfasst wird. In diesem Zusammenhang wurde ein Beispielszenario beschrieben, bei dem eine Web-Anwendung von einem Desktop-PC auf ein Smartphone migriert wurde. Dieser Ansatz ist, in Hinblick auf die Erfassung des Komponentenzustandes, im Vergleich zu den Vorgehensweisen in CRUISe und CoBRA feingranularer. Allerdings beschränken sich die genutzten Methoden und Werkzeuge ausschließlich auf Web-Anwendungen, welche in (X)HTML und JavaScript implementiert wurden.
Nach der Vorstellung dieser drei Forschungsprojekte wurde im vorletzten Abschnitt der Algorithmus **COSMOD** vorgestellt, welcher es ermöglicht ein graphenbasiertes Zustandsmodell einer Komponente zu generieren. Dabei werden ausgehend vom Anfangszustand alle Methoden jedes Folgezustandes mit zufallsgenerierten Parametern ausgeführt bis ein vordefinierter Endzustand erreicht wird.
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

Im vorangegangenen Kapitel wurden mögliche Lösungsansätze und Konzepte verschiedener Forschungsprojekte untersucht, welche für den Prozess der UI-Migration genutzt werden können. Basierend auf den dort gewonnen Erkenntnissen wird nun in diesem Kapitel die Konzeption des Zustandstransfers der UI-Migration erarbeitet. Dafür sollen zunächst die funktionalen und nicht-funktionalen Anforderungen an eine Migrations-Umgebung aufgestellt werden. Darauf aufbauend werden die in Kapitel 3 untersuchten Ansätze mit Hilfe eines Kriterienkataloges, welcher aus den Anforderungen heraus entsteht, evaluiert. Im Anschluss an die Analyse wird das entwickelte Gesamtkonzept vorgestellt. Im nächsten Abschnitt werden dann die Teilkonzepte des UI-Migrationsprozesses erläutert und auf deren Funktionsweise näher eingegangen. Eine Diskussion ausgewählter Teilaspekte sowie eine Bewertung der Konzeption hinsichtlich der definierten Anforderungen schließen das Kapitel ab.

4.1 Anforderungsanalyse

Aus dem Beispielszenario in Abschnitt 1.1 des Einleitungskapitels sowie aus den Erkenntnissen des 3. Kapitels lassen sich diverse Anforderungen ableiten, welche für eine erfolgreiche UI-Migration notwendig sind. Dabei soll zwischen funktionalen und nicht-funktionalen Anforderungen unterschieden werden.

4.1.1 Funktionale 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.

2 Serialisierung des extrahierten Zustandes: UI-Komponenten müssen bei der Bereitstellung ihrer Zustandsdaten das technologie- und plattformneutrale Format des Zustandsmodells berücksichtigen und einhalten.

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 CRUISe 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 Zustandmodells.

5 Kategorisierung der Zustandsinformationen: Mit Hilfe des Zustandsmodelles soll es ermöglicht werden, jede zustandsbehaftete Eigenschaft einer UI-Komponente genau einer vordefinierten Kategorie (UI-Zustand, Verhaltenszustand, Style, Layout – vgl. Abschnitt 4.4.1) zuzuweisen. Mit Hilfe dieser klaren Trennung der erfassten Zustandsdaten, kann beispielsweise eine Filterung erfolgen, um für die UI-Migration nicht benötigte Zustandsinformationen bereits vor dem Migrationsprozess auszusortieren.

4.1.2 Nicht-funktionale Anforderungen

Nachdem der letzte Abschnitt die funktionalen Anforderungen definiert hat, folgen nun die Anforderungen nicht-funktionaler Natur. Diese berücksichtigen unter anderem Schwerpunkte in den Bereichen Plattformunabhängigkeit, Stabilität, Effizienz und Benutzbarkeit.

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 kompositen Anwendung gefährden.

9 Effizienz: Der Migrations-Prozess soll möglichst geringen Einfluss auf das Laufzeitverhalten der gesamten Web-Anwendung ausüben. Das Konzept soll eine zeitlich 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.

1. **Nutzbarkeit des erfassbaren Zustandes:** Das erste und wichtigste Vergleichskriterium untersucht die Effektivität des vorliegenden Ansatzes. Im Kontext dieser Forschungsarbeit wird in diesem Zusammenhang untersucht, wie *feingranular* der Zustand einer Komponente mit den zur Verfügung stehenden Methoden und Werkzeugen erfasst werden kann und ob eine *Kategorisierung* der Zustandsinformationen stattfindet.

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.

3. **Effizienz:** Unter dem Aspekt der mobilen Endgeräte bewertet dieses Vergleichskriterium die Leistung sowie das Zeitverhalten der notwendigen Prozesse. Des Weiteren wird in diesem Zusammenhang der damit verbundene (geschätzte) Ressourcenverbrauch untersucht.

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

Dieser Abschnitt dient der Bewertung aller vorgestellten Konzepte in Bezug auf die UI-Migration mit Hilfe der zuvor definierten Kriterien. Dabei werden spezifische Aspekte von CRUISe mit ähnlichen Aspekten der anderen Ansätze verglichen. In diesem Zusammenhang werden in jedem Teilabschnitt zunächst die Vor- und Nachteile der verschiedenen Vorgehensweisen jedes einzelnen Forschungsprojektes beschreiben und im Anschluss daran folgt die Evaluation auf Grundlage des Kriterienkataloges. Die sich im Anhang befindende Tabelle A.1 fasst das Ergebnis der Evaluation zusammen und wird nachfolgend detailliert erläutert.

CRUISe

Wie im Kapitel des Standes der Forschung und Technik beschrieben, stellt die derzeitige Architektur von CRUISe bereits Methoden bereit, welche für den UI-Migrationsprozess genutzt werden können. Abschnitt 3.1 hat den Austauschvorgang von Komponenten beschrieben und ist dabei unter anderem auf den Aspekt der *Zustandserhaltung* näher eingegangen. Der nachfolgende Abschnitt bewertet in diesem Zusammenhang die dazu verwendeten Technologien anhand des Kriterienkataloges.

(-) 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

dem Austauschprozess nicht erfasst und gehen somit verloren. Des Weiteren kann keine eindeutige Kategorisierung der erfassten Zustandsdaten erfolgen. Alle für den Komponentenaustausch notwendigen Zustandsinformationen werden mit Hilfe der Methode *getProperty(key)* ausgelesen und in serialisierter Form (**Schlüssel-Wert-Paare**) gespeichert. Anschließend wird der Zustand der Zielkomponente durch den Aufruf der Methode *setProperty(key)* für alle gesicherten Eigenschaften wiederhergestellt.
Diese Vorgehensweise ist für die UI-Migration unzureichend, aufgrund der Tatsache, dass ausschließlich Variablen, welche als Teil der Schnittstelle vom Komponentenentwickler definiert worden sind, berücksichtigt werden.

(-) Entwicklungsaufwand für den Komponentenentwickler: Der derzeitige Ansatz zur Erfassung der Zustandsinformationen von UI-Komponenten setzt voraus, dass der Komponentenentwickler zur Implementierungszeit entscheiden muss, welche Variablen für eine mögliche UI-Migration von Bedeutung sind und welche nicht. Des Weiteren müssen Methoden für jede einzelne Zustandsvariable implementiert werden, welche den Wert situationsbedingt extrahieren bzw. injizieren. Demzufolge ist der Entwicklungsaufwand für den Komponentenentwickler als hoch einzustufen.

Aufgrund der Tatsache, dass CRUISe zur zeit keine Möglichkeit bietet UI-Komponenten zu migrieren, lassen sich die Kriterien **Effizienz** und der **Entwicklungsaufwand zur Erweiterung der Thin-Server-Runtime** an dieser Stelle nicht evaluieren.
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.

(+) *Isolation*: Um Komponenten davor zu bewahren einen inkonsistenten Zustand zu erreichen, werden *Wrapper / Proxies* eingesetzt, welche die Komponenteninstanzen umschließen und deren Schnittstellen auf die Spezifikationen gemäß dem Kompositionsmodell abbilden und repräsentieren. Dadurch ist es möglich, die auszutauschende Instanz problemlos von der Kommunikation der Anwendung zu entkoppeln und während des Austauschvorganges ein- und ausgehende Nachrichten zu blockieren und in eine Warteschlange einzureihen. Somit gehen keine Nachrichten während eines Austauschprozesses verloren. Dieses Verfahren stellt die Konsistenz sowie die Integrität der Gesamtanwendung sicher.

(+) *Ablauf des Komponentenaustausches*: Für den Austauschvorgang kommt ein erweitertes Phasen-Commit-Protokoll zum Einsatz, welches ausführlich in Abschnitt 3.1.2 beschrieben wurde. Zusammen mit dem Isolationsmechanismus bildet dieses Phasen-Commit-Protokoll eine gute Grundlage und kann für die UI-Migration verwendet 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 Ursprungskomponente 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.
Die Untersuchungen von CRUISe haben gezeigt, dass die Möglichkeit Komponenten während der Laufzeit auszutauschen ein solides Grundgerüst bietet, auf dem die UI-Migration aufbauen kann. Dabei ist vor allem die Kapselung von Komponenten mit Hilfe von Proxies bzw. Wrappern hervorzuheben, das erweiterte Phasen-Commit-Protokoll sowie der Semantic Matching Prozess. Die derzeitigen Methoden zur Erfassung, Extraktion sowie Injektion des Komponentenzustandes weisen im Gegensatz dazu Defizite auf. Beispielsweise hängt die Granularität des erfassten Komponentenzustandes komplett von dem Komponentenentwickler ab und verschafft diesem somit ein erhöhten Entwicklungsaufwand. Die vorliegende Forschungsarbeit wird an dieser Stelle ansetzen und ein Verfahren entwickeln, mit dem eine feingranulare Zustandserfassung möglich ist, wobei sich der Komponentenentwickler möglichst wenig Gedanken über die Implementierung machen muss.
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.
Ähnlich wie in CRUISe, wird auch in CoBRA das **Entwurfsmuster *Proxy*** eingesetzt, um die notwendige Atomarität bei dem Austausch von Service-Implementierungen zu gewährleisten. Es existiert allerdings ein wesentlicher Unterschied in der Implementierungsebene: Ankommende Nachrichten werden während des Austauschvorganges ausschließlich blockiert und im Gegensatz zu CRUISe nicht gepuffert. Dies führt bei einer Benutzereingabe beispielsweise zu einem Informationsverlust.
Für die Zustandserhaltung wird in CoBRA ein weiteres Entwurfsmuster verwendet, welches in Abschnitt 3.2.2 bereits ausführlich beschrieben wurde: das ***Memento*-Design-Pattern**. Dieses Entwurfsmuster ermöglicht es Komponenten zu jedem Zeitpunkt ihren aktuellen Zustand in ein *Memento-Objekt* abzuspeichern.
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-

sem Zusammenhang müssen alle zustandsrelevanten Attribute bzw. Klassen einer Komponente die Serializable Schnittstelle der *Java Serialization API*[1] implementieren, um in serialisierter Form in dem *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 Thread mittels der Java Serialization API nicht serialisiert werden können.

(-) 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.

(-) Effizienz: Die in diesem Beispiel verwendete Standardserialisierung von Java serialisiert alles, was von dem Basisknoten aus erreichbar ist. Demzufolge besteht die Möglichkeit, dass der Objektgraph sehr groß werden kann. Somit steigt auch der Zeitbedarf sowie das benötigte Datenvolumen. Des Weiteren ist es nicht möglich ausschließlich die Änderungen zu serialisieren. Somit muss immer der komplette Objektgraph neu generiert werden.

(+) 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.

Der klassische Weg von einem Objekt zu einer persistenten Speicherung führte in diesem Beispiel über den Serialisierungsmechanismus von Java. Diese Serialisierung in Binärdaten ist dabei nicht ohne Nachteile. Neben den weiter oben beschriebenen Performanceeinbußen, ist beispielsweise die Weiterverarbeitung von Nicht-Java-Programmen oder eine nachträgliche Änderung der Objektverbunde schwierig.
Zusammenfassend kann festgestellt werden, dass dieser Ansatz für die UI-Migration nicht ausreichend ist. Die Entwurfsmuster *Memento* und *Proxy* bilden zwar eine gute Grundlage, aber die Standard-Serialisierung von Java weist erhebliche Defizite auf. Beispielsweise sind keine Teil-Serialisierungen möglich. Demzufolge müssen immer alle Informationen, ausgehend von dem Basisknoten, serialisiert werden. Dies impliziert einen erhöhten Zeit- und Ressourcenbedarf. Des Weiteren wird bei diesem Ansatz der Komponentenentwickler nicht entlastet, da ihm beispielsweise die Entscheidung, welche Variable zustandsrelevant ist, nicht abgenommen wird. Diese Anforderung ist allerdings essentiell für die vorliegende Forschungsarbeit.
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.

[1] http://java.sun.com/developer/technicalArticles/Programming/serialization/

OPEN

In den Abschnitten 3.3 ff. wurde die OPEN-Plattform überblicksweise beschrieben. In diesem Zusammenhang wurden neben dem allgemein Aufbau der Architektur auch Besonderheiten wie die *OPEN-Adaptoren* oder die *OPEN-Dispatcher* erläutert. Im Anschluss daran wurde der Aspekt der Migration von Komponenten in OPEN detailliert untersucht. Dabei wurde zunächst der generelle Ablauf der UI-Migration beschrieben und anschließend wurde auf die Erfassung des Komponentenzustandes sowie dessen Extraktion von / Injektion in Komponenten eingegangen.
Bei den Untersuchungen wurde festgestellt, dass in OPEN eine *Reverse Engineering-Technik* zum Einsatz kommt, bei der *abstrakte UI-Beschreibungen* aus den (X)HTML-Quelldokumenten generiert werden. Diese werden dann an die Zielplattform angepasst und mit den Zustandsinformationen, welche aus DOM-Baum sowie globalen JavaScript-Variablen bestehen, angereichert. Auftretende Probleme während der Zustandserfassung, wie beispielsweise zyklische Referenzen oder Timer, wurden von den Entwicklern berücksichtigt. In diesem Zusammenhang wurden entsprechende Lösungsansätze erläutert. Im letzten Schritt der UI-Migration wird aus der angepassten, zustandsbehafteten Beschreibung eine Web-Seite generiert und auf das Zielgerät übertragen.
Nach dieser kurzen Zusammenfassung wird OPEN anhand des Kriterienkataloges im nächsten Abschnitt bewertet.

(+) Nutzbarkeit des erfassbaren Zustandes: Mit Hilfe der zuvor beschriebenen *Reverse Engineering-Technik* gelingt es der OPEN-Plattform einen feingranularen Zustand von Web-Anwendungen zu erfassen, welche mit (X)HTML, CSS und JavaScript, oder sogar mit Programmiersprachen wie JSP, PHP oder ASP erstellt wurden. Dabei werden sowohl JavaScript- und BOM-Variablen als auch alle Eigenschaften des DOM-Baums in Form von **Schlüssel-Wert-Paaren** serialisiert. Aufgrund der Tatsache, dass für jeden dieser Teilzustände, welche zum einen plattformspezifisch (z. B. die BOM-Variablen) und zum anderen allgemeingültiger Natur sind (JavaScript-Variablen), unterschiedliche Methoden zur Extraktion der Informationen implementiert werden müssen, *könnten* diese Zustandsdaten bereits während des Extraktionsprozesses annotiert werden, um sie später beispielsweise zu kategorisieren. In diesem Zusammenhang könnte zwischen dem eigentlichen Wertezustand (bspw. der Inhalt eines Textfeldes) und dem Styling- sowie Layout-Zustand, welcher den Aufbau der zu migrierenden Web-Seite beschreibt, unterschieden werden. Somit besteht die Möglichkeit ausschließlich den Wertezustand auf das Zielgerät zu übertragen, um die Effizienz des Migrationsprozesses zu steigern. Diese Feststellung resultiert aus der einfachen Annahme, dass eine geringe Zustandsdatenmenge effizienter verarbeitet werden kann, als eine große Zustandsdatenmenge.
Das Defizit der OPEN Migration Service Platform ist die fehlende Unterstützung von Flash-basierten UI-Komponenten und Java Applets. Diese werden *nicht analysiert* und im Zielkontext entweder ausgetauscht oder einfach weggelassen. Diese Entscheidung wurde aufgrund der Tatsache getroffen, dass Flash-Komponenten keine vergleichbare Schnittstellenspezifikation für den Zugriff auf deren Elemente bieten wie das **D**ocument **O**bject **M**odel. Obwohl ei-

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.

(+) Effizienz: Die Performance der Serialisierung des DOM-Baums sowie des Auslesens der globalen JavaScript-Variablen ist abhängig von dem implementierten Algorithmus. Unter Berücksichtigung der Lösungsansätze für die auftretenden Schwierigkeiten während der Zustandserfassung, bietet die OPEN-Plattform in dieser Hinsicht eine gute Performance [Pat11], sodass auch mobile Endgeräte unterstützt werden können. Des Weiteren ist sowohl die Zeit- als auch die Platzkomplexität als gering einzustufen. Dazu trägt auch die Wahl des jSON-Formates, welches ein geringen Overhead aufweist, bei.

(-) Entwicklungsaufwand zur Erweiterung der Thin-Server-Runtime: Die Tatsache, dass die OPEN-Plattform als eine komplett eigenständige Middleware entwickelt wurde, resultiert in einem hohen Entwicklungsaufwand. Demzufolge könnten lediglich Teilkomponenten der Architektur entnommen und gegebenenfalls erweitert bzw. angepasst werden.

Die Evaluierung der OPEN-Architektur hat gezeigt, dass einige Teilkonzepte von OPEN für die UI-Migration in CRUISe in Frage kommen könnten. Dazu zählt zum Beispiel die Zustandserfassung mit Hilfe des serialisierten DOM-Baumes, inklusive der Lösungsansätze für die auftretenden Schwierigkeiten bei der Erfassung der JavaScript-Variablen. Selbstverständlich müssten diese Technologien im Laufe der Konzeption noch angepasst sowie eventuell erweitert werden, da sich die Architekturen von CRUISe und OPEN grundlegend voneinander unterscheiden. Das große Defizit dieses Ansatzes ist die fehlende Unterstützung von Flash-basierten UI-Komponenten sowie Java Applets.
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.

(-) Nutzbarkeit des erfassbaren Zustandes: Aufgrund der Tatsache, dass die Methoden einer Komponente mit *zufallsgenerierten* Eingabeparametern ausgeführt werden, kann die Granularität des erfassbaren Zustandes unter Umständen sehr grob sein. Dies hängt von der Anzahl der Testdurchläufe sowie den generierten Parametern ab. Demzufolge kann nicht gewährleistet werden, dass alle Zustände einer Komponente durch den Algorithmus entdeckt werden können. Der in COSMOD verwendete Algorithmus berücksichtigt zudem ausschließlich selbst-aufgerufene Properties einer Komponente. Dies impliziert, dass nicht-sichtbare Eigenschaften, welche sich während eines Zustandsüberganges verändern, von dem Algorithmus nicht berücksichtigt werden und somit möglicherweise verloren gehen können.
Bei diesem Vorgehen werden demzufolge keine Zustandsinformationen einer Komponente abgespeichert. Der Nachteil dieses Prinzips besteht darin, dass bei der Wiederherstellung des Zustandes, in der Endphase der UI-Migration, die beiden involvierten Komponenten eine identische Schnittstellenbeschreibung aufweisen müssen. Sollten beispielsweise Methoden in der Zielkomponente existieren, welche zwar die gleiche Aufgabe erledigen, aber zusätzliche Parameter für die Ausführung benötigen oder eine andere Bezeichnung besitzen, müsste eine Transformation des Zustandsgraphen erfolgen, bevor die Sequenz der generierten Zustandsübergänge der Ausgangskomponente abgearbeitet werden kann. Würden Variablen oder Methoden, welche die Ausgangskomponente besitzt, in der Zielkomponente sogar komplett fehlen, so würde die Wiederherstellung des Zustandes in vielen Fällen ebenfalls scheitern, da der Algorithmus nicht terminiert, sobald eine erforderliche Methode in der Zielkomponente nicht aufgerufen werden kann. In diesem Zusammenhang müssten Mashup-Komponenten, im Kontext mobiler Endgeräte, den gleichen Funktionsumfang bieten können wie beispielsweise Desktop-Anwendungen. Da dieses Kriterium aufgrund eingeschränkter Rechenleistung sowie unterschiedlicher Ein- und Ausgabemodalitäten der mobilen Endgeräte allerdings nicht gewährleistet werden kann, müsste demzufolge ein Verfahren entwickelt werden, welches die generierte Sequenz von Zustandsübergängen – *während der Laufzeit* – in ein passendes Zustandsmodell transformiert.

(-) Entwicklungsaufwand für den Komponentenentwickler: Obwohl die eigentliche Generierung des Zustandsmodelles, unter der Voraussetzung einer vorhandenen Schnittstellenbeschreibung für die zu untersuchende Komponente, vollständig automatisiert abläuft, wäre der Entwicklungsaufwand für den Komponentenentwickler trotzdem als hoch einzustufen. Dies resultiert beispielsweise aus der Tatsache, dass für die Generierung des Zustandsgraphen ein von dem Komponentenentwickler aufgesetztes Testsystem notwendig ist.

(-) Effizienz: Wie in Abschnitt 3.4 beschrieben, wird für den Algorithmus in [SM09] eine polynomiale Zeitkomplexität angegeben. Demzufolge ist die Generierung sowie die Analyse des Zustandsmodelles während der Laufzeit ausgeschlossen, obwohl es theoretisch möglich wäre. Dies würde allerdings, abhängig von der Komplexität der zu migrierenden Komponente, zu unvorhersehbar langen Wartezeiten für den Anwender führen. In diesem Zusammenhang müsste demnach ein Komponentenentwickler, vor der Veröffentlichung, seine Kompo-

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

(-) Entwicklungsaufwand zur Erweiterung der Thin-Server-Runtime: Aufgrund der Tatsache, dass ein System von Grund auf neu entwickelt werden müsste, welches aus jeder Komponente dessen Zustandsmodell generiert, wird der Entwicklungsaufwand als hoch eingeschätzt. Des Weiteren weist der vorhandene Pseudo-Code des Algorithmus Defizite auf, welche zusätzlich behoben werden müssten. Dazu zählt zum Beispiel die Problematik, zu wissen wann der Endzustand einer Komponente erreicht ist.

Nach dieser Evaluierung kann geschlussfolgert werden, dass der untersuchte Ansatz zur Modellierung des Zustandsmodelles mit Hilfe von Methodenaufrufen 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 erfassbaren Zustandes: Aufgrund der Tatsache, dass alle UI-Elemente sowie deren aktuelle Zustandswerte, welche durch den Anwender modifiziert worden sind, erfasst werden, kann davon ausgegangen werden, dass alle zustandsrelevanten Informationen in dem resultierenden Zustandsmodell enthalten sind.
Bei dieser Herangehensweise können die Zustandsinformationen in verschiedenen 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 Zusammenhang 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.

(+) Effizienz: Bei der Verwendung von *Mutation Events* aus dem Jahr 2000 muss mit deutlich spürbaren Geschwindigkeitseinbrüchen bei der Ausführung von Web-Anwendungen gerechnet werden. Die aktuellen *Mutation Observer* hingegen, welche sowohl vom aktuellen Firefox, als auch von Chrome und Safari interpretiert werden können, beheben dieses Performance-Defizit, indem sie erst *nachdem* der Nutzer mit dem jeweiligen UI-Element interagiert hat, eine Liste mit allen Änderungen des DOM-Baum-Abschnittes bereitstellen.

(-) Entwicklungsaufwand zur Erweiterung der Thin-Server-Runtime: Wie bereits erwähnt, liegt die Herausforderung dieses Ansatzes in der Übertragung und Umsetzung der Kernidee auf verschiedene Programmiersprachen. Dabei müssen sowohl für die Erfassung, als auch für die Extraktion sowie Injektion der Zustandsinformationen *abstrakte Klassen* implementiert werden, welche diese Migrationsfunktionalitäten Komponenten- bzw. Programmiersprachenspezifisch bereitstellen.

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.

- **Erfassung von auftretenden DOM-Baum- bzw. UI-Events:** Die Grundidee dieses Ansatzes umfasst die *Überwachung und Registrierung aller ausgehenden Events* einer UI-Komponente. Bei JavaScript-basierten Komponenten bildet in diesem Zusammenhang der DOM-Baum, bzw. der Abschnitt des Baumes, in welchem die Komponente spezifiziert wird, die Basis dieser Vorgehensweise. Dabei können bspw. *Mutation Observer* verwendet werden, welche als *Listener* eines HTML-Dokumentes die Änderungen des DOM-Baumes überwachen.

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 resultierten Schlussfolgerungen, wird im nächsten Abschnitt das entwickelte Gesamtkonzept im Überblick beschrieben.

4.3 Überblick des Gesamtkonzeptes

Im bisherigen Verlauf der Arbeit wurden Grundlagen und existierende Lösungsansätze zu den Herausforderungen im Bereich der UI-Migration untersucht. Basierend auf den daraus gewonnenen Erkenntnissen und den aufgestellten Anforderungen, ist die Zielstellung dieses Abschnittes die Konzeption einer Infrastruktur für die Migration von UI-Komponenten im Kontext des Forschungsprojektes CRUISe. Dazu wird im ersten Schritt mit Hilfe eines Anwendungsszenarios die Zielstellung der UI-Migration ein weiteres Mal hervorgehoben. Danach wird ein grober architektonischer Überblick der Migrationsumgebung gegeben und im Anschluss daran werden die einzelnen Bestandteile dieser Migrationsinfrastruktur näher beschrieben. Nach diesen einleitenden Abschnitten folgt eine detaillierte Erläuterung der Funktionsweise des entwickelten Konzeptes Zustandsinformationen aus UI-Komponenten zu extrahieren bzw. zu injizieren. Des Weiteren wird in diesem Zusammenhang das verwendete Zustandsmodell näher betrachtet.

4.3.1 Anwendungsszenario

Als Grundlage eines Migrationsszenarios wird angenommen, dass ein Anwender mit Hilfe einer Webanwendung eine bevorstehende Reise zunächst an seinem stationären Desktop-PC zu Hause plant und die Anwendung während des Trips verwendet, um den Ablauf sowie etwaige Notizen einzelner Zwischenstationen abrufen zu können. Die in *Flash* programmierte Webanwendung ermöglicht es ihm an seinem PC interessante Orte auf einer Kartenkomponente zu markieren. Des Weiteren kann er jedem Ort geplante Tätigkeiten hinzufügen, indem er diese textuell beschreibt und sie als Notiz dem jeweiligen Ort zuweist. Zum Zeitpunkt der Reise möchte der Nutzer die Kartenkomponente der Webanwendung mit allen festgelegten Orten und geplanten Aktivitäten sowie dem aktuellen Interaktionszustand auf sein Smartphone migrieren, um die Reise antreten zu können. Aufgrund der Tatsache, dass sein Smartphone keine Flash-Unterstützung bietet, wird die UI-Komponente während des Migrationsprozesses vom durch eine funktional äquivalente *JavaScript-Komponente* ausgetauscht. In diesem Zusammenhang werden die Zustandsinformationen der Ausgangskomponente auf die Zielkomponente des Smartphones übertragen.
Nachdem dieser Abschnitt mit Hilfe eines Anwendungsszenarios die Notwendigkeit der UI-Migration erläutert hat, beschreibt der folgende Abschnitt das Grobkonzept zur Architektur der verteilten Laufzeitumgebung migrierender Komponenten sowie die darin enthaltenen Kernfunktionen und grenzt gleichzeitig die Verantwortlichkeiten der einzelnen Bestandteile klar voneinander ab.

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.

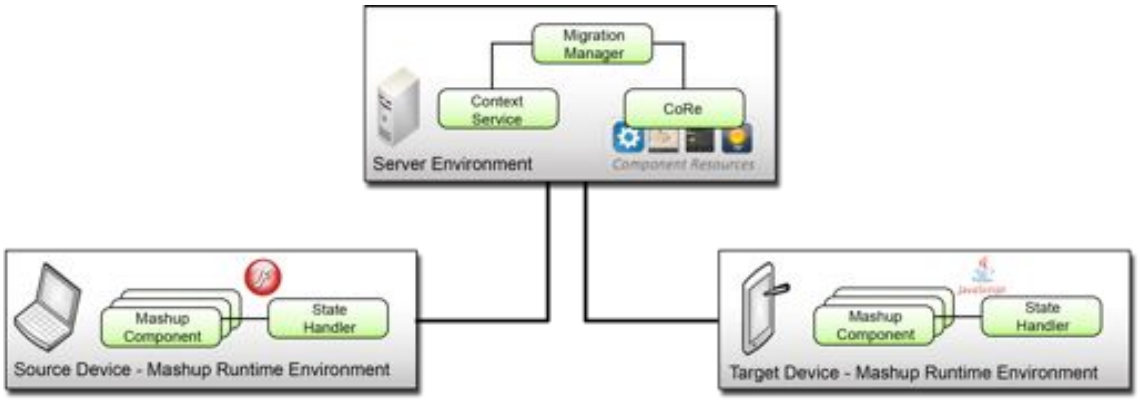

Abbildung 4.1: Grobkonzept der UI-Migration

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

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

Auf dem Ausgangsgerät (**Source Device**) befindet sich die in *Flash* programmierte Webanwendung (**Mashup Component**). Diese enthält, dem Anwendungsszenario entsprechend, eine Kartenkomponente sowie einen **State Handler**, welcher für die Extraktion der Zustandsinformationen aus der Ausgangskomponente verantwortlich ist. Des Weiteren ist er für die Transformation der extrahierten Zustandsdaten in ein plattform- und technologieneutrales Zustandsmodell verantwortlich. Die Notwendigkeit solch eines Zustandsmodelles wird bei der Betrachtung der beiden anderen Komponenten klarer.
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 ersetzten.
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.

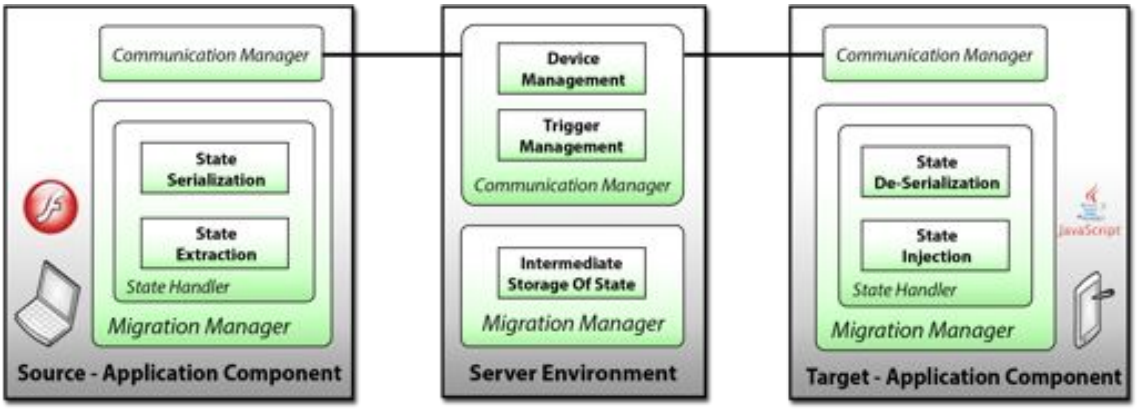

Abbildung 4.2: Detailansicht der beteiligten Komponenten

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 Migrations-Ziel(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.

Zusammenfassend kann festgestellt werden, dass der *Migration Manager* das Grundgerüst für die UI-Migration bildet und die wichtigsten Funktionen zur Durchführung der UI-Migration bereitstellt. Der MM ist sowohl Bestandteil der TSR des Ausgangs- sowie des Zielgerätes, als auch Bestandteil der serverseitigen Migrationsumgebung und erstreckt sich somit über die gesamte Migrationsinfrastruktur.
Bevor in den nächsten Abschnitten dieses Kapitels detailliert auf die Zustandserfassung und -transferierung eingegangen wird, diskutiert der nachfolgende Abschnitt zunächst grundlegende Vorbetrachtungen, wie beispielsweise die Erfassung, Verwaltung und Auswahl der beteiligten Geräte oder verschiedene Auslösungsmechanismen des UI-Migrationsprozesses. Diese sind zwar wichtig für die allgemeine Ausführung der UI-Migration, sind aber nicht Bestandteil der Aufgabenstellung und werden deshalb nur überblicksweise erörtert.

4.3.4 Vorbetrachtungen zur UI-Migration

Dieser Abschnitt dient dazu wichtige Voraussetzungen für den UI-Migrationsprozess zu klären sowie den Fokus der vorliegenden Forschungsarbeit noch einmal hervorzuheben und von anderen Problemstellungen abzugrenzen. Dazu werden die einzelnen Prozessschritte, welche in Abbildung A.2 **grün** dargestellt sind, überblicksweise beschrieben und verschiedene Ansätze in diesem Zusammenhang diskutiert. Die **rot** dargestellten Teilprozesse werden in den Abschnitten 4.4 ff. detailliert erläutert. Eine vollständige Betrachtung *aller* Teilaspekte ist im Rahmen der gestellten Aufgabe, aufgrund der hohen Komplexität, an dieser Stelle nicht möglich.
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

Die *IP-Adresse* jedes Gerätes wird für die Übertragung der Zustandsinformationen (Ausgangsgerät → Zielgerät) benötigt und der *Gerätetyp* ist für das Auffinden einer passenden UI-Komponente (Punkt 4), welche im Kontext des Zielgerätes integriert werden soll, notwendig. Damit sich der Anwender keine IP-Adressen merken muss, wird jedem Gerät eine feste ID beziehungsweise Bezeichnung zugewiesen, welche dem *clientseitigen Migration Manager* weitergeleitet wird. Diese Aufgaben werden von dem *Communication Manager* erledigt, wobei grundsätzlich zwischen zwei Arten der Geräteregistrierung unterschieden werden kann. Diese werden nachfolgend näher erläutert:

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 TCP[2], UDP[3] oder ARP[4] derzeitig lediglich Informationen wie die MAC-Adresse, der Name des Endgerätes sowie der Name des Herstellers ausgelesen werden können, setzt dieses Vorgehen voraus,

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

dass sowohl serverseitig, als auch clientseitig Methoden zur automatischen Registrierung implementiert werden müssen, um weitere Informationen, wie zum Beispiel Gerätespezifikationen, eines Endgerätes erfassen zu können. In diesem Zusammenhang wäre die Verwendung etablierter Mechanismen zur automatischen Geräteerkennung wie UPnP[5] und Bonjour[6] denkbar, welche in Anbetracht der geforderten Plattformunabhängigkeit jedoch beide auf Client- und Serverseite implementiert werden müssten, da beispielsweise Geräte der Marke Apple ausschließlich *Bonjour* unterstützen. Auf eine detaillierte Beschreibung der Funktionsweisen wird an dieser Stelle verzichtet und für weiterführende Informationen zu diesem Thema auf [WP11] verwiesen.

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 Geräteregistrierung eine Benutzerinteraktion erforderlich, bei der zum Beispiel eine spezielle Web-Seite aufgerufen wird, welche die Metadaten des verwendeten Browsers ausließt, 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.
Nachdem alle in Frage kommenden Endgeräte registriert wurden, beginnt bereits der **Prozess der Zustandserfassung**. Detaillierte Informationen diesbezüglich werden an dieser Stelle allerdings ausgelassen, mit der Begründung, dass dieser Vorgang ausführlich in Abschnitt 4.6.2 diskutiert wird.

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 Ortswechsel 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*)

und eine automatisierte UI-Migration von dem Desktop-PC auf das Smartphone wird ausgelöst. Um dieses konkrete Beispiel jedoch zu ermöglichen, müssten die Beschleunigungs- und Ortungssensoren des Smartphones ständig überwacht und ausgewertet werden. Dies impliziert die Implementierung clientseitiger Kontextmonitore sowie einer serverseitigen Kontextverwaltung. In diesem Zusammenhang müsste das Regelwerk des Kontextservices *CroCo*, welcher bereits im Forschungsprojekt CRUISe verwendet wird, erweitert werden. Dabei müssten Kontextdaten wie bspw. Umgebungsinformationen, Hardwarespezifikationen oder sogar die Inbetriebnahme eines weiteren Endgerätes in Betracht gezogen werden. Aufgrund der Tatsache, dass diesbezüglich eine nicht überschaubare Menge von weiteren Regeln neu definiert und implementiert werden müsste, wird die automatische, kontextbasierte Auslösung der UI-Migration im weiteren Verlauf dieser Arbeit nicht näher betrachtet.

b) Die On-Demand-UI-Migration: Die zweite Möglichkeit den Migrationsprozess auszulösen ist die On-Demand-UI-Migration, welche durch eine Benutzerinteraktion ausgelöst wird. Somit bestimmt der Anwender den Zeitpunkt der UI-Migration vollständig selbst und macht die Einbindung externer Kontextdienste demzufolge überflüssig.
Voraussetzung für diese Auslösungsstrategie sind allerdings UI-Elemente, welche von dem *clientseitigen Migration Manager* bereitgestellt werden müssen. Mit Hilfe dieses UIs werden dem Anwender mehrere Auswahlmöglichkeiten bereitgestellt. Zunächst kann er die Komponente aus der Mashup-Anwendung auswählen, welche auf ein anderes Endgerät migriert werden soll. Anschließend präsentiert der clientseitige MM alle für die UI-Migration zur Verfügung stehenden Geräte. Diese Information bezieht er dabei von dem *serverseitigen CM* (vgl. Abschnitt 4.3.4). Sobald der Nutzer seine Auswahl getroffen hat, werden die Informationen an den *Migration Server* gesendet, welcher im nächsten Schritt eine passende UI-Komponente sucht.

Wie bereits angedeutet, wird im weiteren Verlauf der Arbeit ausschließlich der Ansatz der *On-Demand-UI-Migration* in Betracht gezogen.
Nachdem die UI-Migration durch den Anwender ausgelöst wurde, müssen im nächsten Prozessschritt alle UI-Komponenten auf dem Ausgangsgerät pausiert werden. Mit dieser Herausforderung befasst sich der nachfolgende Abschnitt und erläutert in diesem Zusammenhang die damit verbundenen Herausforderungen.

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

der Nutzer ununterbrochen weiter und verändert somit fortlaufend den Zustand der Kartenkomponente, sowie den Zustand der zweiten Komponente. Aufgrund der Tatsache, dass in diesem Beispiel nicht gewährleistet werden kann, dass der *aktuellste* Zustand *beider* UI-Komponenten auf das Zielgerät übertragen wurde, ist davon auszugehen, dass sich die migrierte Webanwendung in einem inkonsistenten Zustand befindet. Diese Annahme basiert auf der Tatsache, dass Prozesse eines Computersystems immer sequentiell und niemals parallel abgearbeitet werden.
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

Das nächste Teilkonzept, welches aufgrund der vorliegenden Aufgabenstellung lediglich überblicksweise in den Vorbetrachtungen beschrieben wird, ist die Zwischenspeicherung der extrahierten Zustandsdaten des Ausgangsgerätes. An dieser Stelle wird demzufolge angenommen, dass die Zustandsinformationen der zu migrierenden UI-Komponenten bereits extrahiert wurden (4.) und in serialisierter Form (5.) an den Server übergeben wurden (6.). Detaillierte Vorgehensweisen dazu werden in den Abschnitten 4.6.2 ff. beschrieben.
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-

nen des Zielgerätes, auf welchem die Zielkomponente ausgeführt werden soll. Diese Informationen erhält der *MM* von dem ebenfalls serverseitigen *Component Manager (CM)*. In diesem Zusammenhang wäre, ähnlich wie bei der kontextbasierten Auslösung der UI-Migration, die Einbindung und gegebenenfalls die Erweiterung des Kontextservices CroCo denkbar. Dabei könnte der Context Service für eine erweiterte Filterfunktion eingesetzt werden und alle in Frage kommenden UI-Komponente nochmals anhand der erfassten Kontextinformationen, wie beispielsweise der aktuelle Standort oder vorherrschende Umgebungsgeräusche, aussortieren. Analog zu den in Abschnitt 4.3.4 beschriebenen Herausforderungen und in Anlehnung an das eingangs beschriebene Anwendungsszenario einer Reiseplanung wird an dieser Stelle jedoch angenommen, dass lediglich eine mögliche UI-Komponente für die Integration auf dem Zielgerät integriert werden kann. Der *serverseitige MM* hält in diesem Zusammenhang den *Komponentendeskriptor* dieser Komponente für den weiteren Verlauf der UI-Migration bereit.
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.

2. Der Anwender hat durch seine Interaktion mit verschiedenen UI-Elementen, welche durch den *clientseitigen Migration Manager* bereitgestellt wurden (Abschnitt 4.3.4), eine beliebige UI-Komponente sowie ein migrationsfähiges Zielgerät in der Web-Anwendung auf dem Ausgangsgerät ausgewählt. Diese Informationen wurden nach der Auswahl des Benutzers an den *Migration Server* übermittelt.

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 Komponentendeskriptor) 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.

In den folgenden Abschnitten wird die Extraktion der Zustandsdaten auf dem Ausgangsgerät näher beleuchtet. Dieser Prozessschritt wird größtenteils von dem *State Handler* durchgeführt, dessen Aufgaben diesbezüglich detailliert in Abschnitt 4.6.2 betrachtet werden. Des Weiteren befasst sich der folgende Abschnitt zunächst mit dem Aufbau des entwickelten Zustandsmodelles und geht in diesem Zusammenhang auf die Kategorisierung von Zustandsdaten (Abschnitt 4.4.1) ein.

4.4 Aufbau des verwendeten Zustandsmodelles

Nachdem der vorherige Abschnitt wichtige Teilkonzepte der UI-Migration überblicksweise beschrieben hat, angefangen bei der Verwaltung der beteiligten Endgeräte bis hin zur initialen Integration der Zielkomponente, präsentiert dieser Abschnitt den Aufbau des entwickelten Zustandsmodelles und geht dabei auf dessen Besonderheiten ein. Dazu wird im folgenden Abschnitt erörtert, welche Arten von Zustandsinformationen existieren und ob es notwendig bzw. vorteilhaft ist diese voneinander zu unterscheiden.

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:

Zustand der Wertemenge Diese Art der Zustandsinformationen beschreibt den eigentlichen Inhalt einzelner Elemente einer UI-Komponente. Dazu zählen zum Beispiel der Inhalt von Textfeldern oder der Zustand eines Buttons (aktiviert oder deaktiviert).

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 selbst.

Aus diesen Definitionen der Zustandskategorien geht hervor, dass der *Zustand der Wertemenge* zusammen mit dem *Verhaltenszustand* ausschlaggebend für die Funktionalität einer UI-Komponente sind. Bei der UI-Migration ist demzufolge die Extraktion dieser beiden Zustände *obligatorisch*. Doch existieren Migrationsszenarien in denen es möglicherweise trotzdem sinnvoll wäre die Style- und Layoutinformationen zu migrieren? Tabelle 4.3 bietet in diesem Zusammenhang einen Überblick, welche Zustandsdaten zwischen welchen Endgerätetypen ausgetauscht werden sollten.

Ausgangsgerät	Desktop-PC	Desktop-PC	Desktop-PC	Tablet	Tablet	Smartphone
Zustand der Wertemenge	x	x	x	x	x	x
Verhaltenszustand	x	x	x	x	x	x
Style-Informationen	x			x		x
Layout-Informationen	x			x		x
Zielgerät	Desktop-PC	Smartphone	Tablet	Tablet	Smartphone	Smartphone

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.
Ein denkbares Szenario, bei welchem heterogene Geräte beteiligt sind und trotzdem alle Zustandsinformationen übertragen werden sollten, wäre die UI-Migration von einem 10"Netbook auf ein 10"Tablet. Hierbei wird das Argument der unterschiedlichen Bildschirmgröße außer Kraft gesetzt und auch die Leistung moderner Tablet-PCs kann in vielen Fällen mit der Leistung ultra-mobiler Laptops mithalten. Denkbar und notwendig wäre in diesem Zusammenhang ein System, welches die Style- und Layoutinformationen im Falle der Zustandssynchronisation heterogener Anwendungskomponenten transformiert. Dies soll im Rahmen der Arbeit allerdings nicht untersucht werden.
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 überblickswei-

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 beschreiben und im Anschluss daran folgt eine detaillierte Betrachtung jedes Teilaspektes.

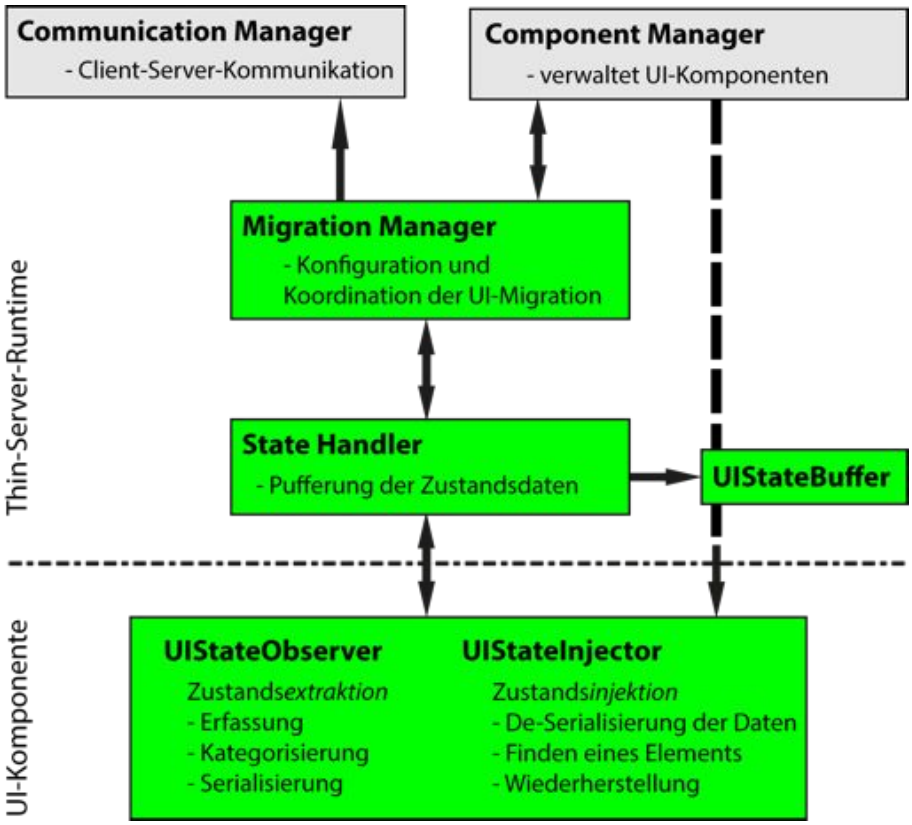

Abbildung 4.4: Zusammenhänge der beteiligten CRUISe-Komponenten

Ein wichtiger Grundgedanke des erarbeiteten Konzeptes ist die Tatsache, dass **jede UI-Komponente selbst für die *Erfassung und Injektion der Zustandsdaten* verantwortlich** ist. Demzufolge können sämtliche UI-Komponenten unterschiedlichster Programmiersprachen unterstützt werden. Die Voraussetzung dafür ist allerdings, dass für jede Plattform die entsprechenden Methoden des *UIStateObservers* sowie *UIStateInjectors* implementiert worden sind. Folglich stellen programmiersprachenspezifische Basisklassen, welche den UI-Komponenten die Migrationsfähigkeit verleihen, die Grundfunktionalitäten zur Verfügung und verringern damit den Implementierungsaufwand des Komponentenentwicklers. Demzufolge muss sowohl der *UIStateObserver* als auch der *UIStateInjector* für jede Programmiersprache

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.
Nachdem die zustandsrelevanten Daten extrahiert wurden, *puffert* der *State Handler* diese Informationen. Sobald der Anwender die UI-Migration ausgelöst hat, holt sich der *Migration Manager* die gepufferten Daten und übergibt sie dem *Communication Manager*, welcher die Zustandsinformationen zum Server bzw. zum Zielgerät transferiert. Des Weiteren stellt der *clientseitige MM* die Benutzerschnittstelle für die Konfiguration des Migrationsprozesses bereit. Somit wird es dem Anwender ermöglicht innerhalb einer kompositen Web-Anwendung *eine* UI-Komponente auszuwählen sowie das gewünschte Zielgerät, auf dem die Komponente migriert werden soll. Der *Component Manager* ist für den Lebenszyklus jeder UI-Komponente verantwortlich und instantiiert zusätzlich den *UI-State Observer* als auch den *UI-State Injector* in jeder Komponente.
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 Zustandsextraktionsmethode

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 (inklusive 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.

- Dieses Defizit besteht bei dem Ansatz der Event-Überwachung, bei welcher **alle auftretenden DOM-Baum-Struktur-Veränderungen erfasst** werden, nicht. Der Vorteil dieses Ansatzes besteht darin, dass sich der Grundgedanke - *die Erfassung und Verarbeitung auftretender Events* - auf andere Programmiersprachen, wie beispielsweise *Flash*, übertragen lässt. In diesem Zusammenhang muss allerdings ein nicht zu verachtender Implementierungsaufwand, bspw. von dem Entwickler der Laufzeitumgebung, geleistet werden, um für jede Programmiersprache die entsprechenden Extraktions- und Injektionsmethoden bereitzustellen.

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

Dieser Abschnitt beschreibt die prinzipielle Vorgehensweise der konzeptionierten Zustandsextraktion, welche im weiteren Verlauf der Arbeit als *Event-Detection* bezeichnet wird. Die Grundidee dieses Ansatzes umfasst die **Überwachung und Registrierung aller ausgehenden, zustandsbehafteten Events**. Dabei spielen insbesondere die Inhalte bzw. **Zustände aller UI-Elemente**, welche durch Benutzereingaben modifiziert worden sind, eine entscheidende Rolle. Des Weiteren stehen **Intervall-Timer**, welche nach Ablauf einer festgelegten Zeitspanne immer wieder eine bestimmte Funktion der UI-Komponente aufrufen (siehe Abschnitt 3.3.3), im Fokus dieses Ansatzes.
Ein *Event* wird in diesem Zusammenhang als eine Zustandsänderung der UI-Komponente verstanden (vgl. Abschnitt 2.3.3), welche beispielsweise dann auftritt, wenn der Nutzer eine Stecknadel auf der Kartenkomponente platziert, um einen bestimmten Ort zu markieren. Die Interaktion des Nutzers mit der UI-Komponente (Mausklick) könnte dabei zusätzlich als Benutzereingabe registriert werden. Da die Information „Mausklick auf Kartenkomponente“ für die Funktionalität der UI-Komponente keine entscheidende Rolle spielt und somit für die UI-Migration wertlos ist, kann sie allerdings verworfen werden. Wichtig hingegen ist die Information, dass eine Stecknadel auf der Karte gesetzt wurde und an welcher Stelle sie von dem Anwender platziert worden ist. Diese Erkenntnisse beruhen auf der Annahme, dass alle Inhalte bzw. Zustände der UI-Elemente, mit denen der Anwender interagiert hat, essentiell für den Erhalt des Komponentenzustandes sind. Im Gegensatz dazu,

werden Interaktionsobjekte, welche nicht durch den Nutzer verändert worden sind, bei der UI-Migration im Zielkontext mit den Ausgangswerten initialisiert. Eine Ausnahme bilden dabei UI-Elemente, welche in Abhängigkeit zu anderen Elementen der Komponente indirekt modifiziert wurden - beispielsweise durch Funktions- bzw. Methodenaufrufe. Dieser Zustand muss wiederum erfasst und in die Zielkomponente migriert werden. Eine detaillierte Aufschlüsselung, welche zustandsrelevanten Informationen erfasst werden, folgt in den kommenden Absätzen.
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 *UIStateObservers*** Bevor die erste Zustandsänderung erfasst werden kann, instantiiert der *Component Manager*, während der Initialisierung der Web-Anwendung (t_0 *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 instantiiert 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.

2. **Erfassung und Filterung von UI-Events** Die Hauptaufgabe des *UIStateObservers* ist die Erfassung zustandsrelevanter Informationen einer UI-Komponente während der Laufzeit. Wie bereits in Abschnitt 4.4.1 beschrieben, lässt sich die Menge aller Zustandsinformationen einer Komponente in vier Kategorien einteilen, wobei der Fokus dieser Arbeit auf den **Zustand der Wertemenge** sowie auf den **Verhaltenszustand** gelegt wird. Aufgrund der Tatsache, dass der *UIStateObserver* zunächst einmal alle auftretenden UI-Events erfasst, müssen diese im nächsten Schritt in ***zustandsirrelevante* UI-Events**, wie z. B. einfache Mausklicks, und ***zustandsrelevante* Mutation-Events**, wie bspw. der geänderte Inhalt einer Textbox, unterteilt werden. Des Weiteren werden zusätzlich **Intervall-Timer**, welche für *sich wiederholende Funktionsaufrufe* genutzt werden, registriert.
   Eine sofortige Weiterleitung erfasster UI-Events an den *State Handler* oder den *serverseitigen Migration Manager (MM)* wird an dieser Stelle bewusst nicht durchgeführt, weil dazu eine Transformation der Events notwendig ist, sobald die UI-Komponente nicht in JavaScript implementiert wurde. Des Weiteren muss für den Fall einer Weiterleitung zum *(serverseitigen) MM* eine ununterbrochene Verbindung für den ständigen Datenaustausch sichergestellt werden. Demzufolge werden alle Teilprozesse bis zum Zeitpunkt der Zustandsextraktion innerhalb der UI-Komponente durchgeführt.

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 ersten Fall werden der *Name des Elementes* sowie dessen *aktueller Wert* registriert, sobald eine Zustandsveränderung durch den *UIStateObserver* festgestellt wird. Demzufolge werden sowohl alle *direkt modifizierten*, als auch alle *indirekt veränderten* UI-Elemente, welche bspw. durch interne Prozesse der Komponente geändert wurden, erfasst. Im zweiten Fall wird angenommen, dass durch Betätigen eines Buttons z. B. ein *Intervall-Timer* aktiviert wird, welcher nach Ablauf einer bestimmten Zeitspanne immer wieder Codesegmente aufruft bis eine entsprechende *clearInterval*-Funktion aufgerufen wird. Diese speziellen *zyklischen Funktionen* werden mit in den *UIStateBuffer* aufgenommen. Konkret wird dabei der Funktionsaufruf und mögliche Parameter des Methodenaufrufes zwischengespeichert sowie die Länge der Zeitspanne des Intervall-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 Zielkontext eventuell implementierte „UNDO-Funktionalitäten“ direkt nach der UI-Migration *nicht* genutzt werden.

4. Blockierung der UI-Komponente Alle bis zu diesem Zeitpunkt beschriebenen Funktionen werden während der Laufzeit durchgeführt, wobei die Punkte 2 und 3 jeweils abwechselnd abgearbeitet werden. Bei dem Zeitpunkt t_3 wird die eigentliche UI-Migration durch den Anwender ausgelöst und wie in Abbildung A.3 zu erkennen, wird zum einen das *User Interface gesperrt* und zum anderen leitet der *clientseitige MM*, unter Zuhilfenahme des *Component Managers (CM)*, welcher für den Lebenszyklus jeder Komponente verantwortlich ist, die Blockierung der zu migrierenden UI-Komponente ein. Die Benutzungsoberfläche wird in diesem Zusammenhang mit einem *„Overlay“* deaktiviert, um keine weiteren Eingaben des Anwenders zuzulassen. Alle laufenden Aktivitäten bzw. aktiven Berechnungen werden von der UI-Komponente zu Ende geführt, bis ein konsistenter Zustand erreicht wird. Die ausgehenden Events in diesem Zeitraum werden analog zu Punkt 2 und 3 erfasst, kategorisiert, serialisiert und

gepuffert.
An dieser Stelle wird auf Abschnitt 3.1.2 verwiesen, in welchem das *2-Phasen-Commit-Protokoll* ausführlich beschrieben wurde [Rad11]. Des Weiteren wurde sowohl die Notwendigkeit als auch die Funktionsweise solch eines Blockierungsprozesses unter anderem in den Abschnitten 3.1.1 und 3.2.1 beschrieben. Demzufolge entfällt an dieser Stelle eine detaillierte Betrachtung.

5. **Extraktion der Zustandsdaten** Sobald die zu migrierende UI-Komponente alle laufenden Aktionen abgeschlossen hat, signalisiert sie dies durch ein entsprechendes *ready*-Event beim *CM* sowie beim *MM*, woraufhin die **Extraktion der Zustandsdaten** durchgeführt wird. Der *clientseitige Migration Manager* holt sich die Menge aller gepufferten Zustandsinformationen mit Hilfe des *State Handlers* aus dem *UIStateBuffer* und übergibt diese dem *clientseitigen Communication Manager*, welcher den Transport der erfassten Zustandsdaten zum Zielgerät vorbereitet. Eine detaillierte Beschreibung dieser **Einleitung der Transferphase** folgt in Abschnitt 4.7.

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

Im Anschluss an die Erfassung und Extraktion der Zustandsdaten einer UI-Komponente müssen diese Daten von dem *clientseitigen Migration Manager* des Ausgangsgerätes, über den *serverseitigen MM*, bis hin zum *Migration Manager* des Zielgerätes übertragen werden. Diese Aufgabe wird von dem **Communication Manager** (CM) übernommen, welcher sowohl clientseitig als auch auf Serverseite aktiv wird. In diesem Zusammenhang erläutert dieser Abschnitt auf welche Art und Weise die Kommunikation zwischen Client und Server realisiert wird, wobei zunächst die vorbereitenden Maßnahmen des *CM* in Bezug auf den bevorstehenden Datentransfer beschrieben werden. Im Anschluss daran wird in 4.7.2 zunächst auf die Nachrichtenübertragung von dem Ausgangsgerät zu dem Migration Server eingegangen und danach steht der weitere Datentransfer ausgehend vom Server hin zum Zielgerät im Fokus der Betrachtungen.

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

Der *Communication Manager* holt sich die *ID der UI-Komponente* sowie die *ID des Zielgerätes* von dem *clientseitigen MM*, welcher diese Informationen bereits zur Laufzeit der Web-Anwendung für den Nutzer bereitstellt, um die UI-Migration entsprechend zu konfigurieren (siehe Abschnitt 4.3.2). Die ID der Komponente wird in diesem Zusammenhang für das *Auffinden einer funktional äquivalenten UI-Komponente* benötigt und mit Hilfe der ID des Zielgerätes wird sichergestellt, dass die Zustandsdaten das gewünschte Endgerät erreichen. Nach diesen beiden Elementen werden am Ende des Datenstroms die extrahierten Zustandsinformationen der zu migrierenden UI-Komponente angehängt. An dieser Stelle wird auf Abbildung A.4 verwiesen, welche zusammenfassend den *Aufbau des entwickelten* ***Kommunikationsprotokolls*** in einer formalen Definition verdeutlicht.
Sobald das eben spezifizierte Datenpaket an den *serverseitigen Communication Manager*, welcher die Informationen an den *serverseitigen MM* weiterleitet, gesendet wurde, wird ein *Timer* gestartet. Somit kann in einem Fehlerfall eine *Timeout-Funktion* den Transfer wiederholen oder die UI-Migration abbrechen und die blockierte UI-Komponente mit Hilfe des *Component Managers* wieder aktivieren, sodass der Nutzer auf dem *Ausgangsgerät* weiterarbeiten kann. Bei einer erfolgreichen Datenübertragung wird die UI-Komponente des Ausgangsgerätes sowie dessen User-Interface wieder aktiviert. Demzufolge kann der Nutzer zeitnah nach Ausführung der UI-Migration mit der Web-Anwendung weiterarbeiten, wobei ab diesem Zeitpunkt alle auftretenden Zustandsänderungen nicht im Zielkontext wiederzufinden sein werden.

4.7.2 Ausgangsgerät → Migration Server → Zielgerät

Sobald alle notwendigen Informationen vom *clientseitigen Communication Manager* zusammengetragen wurden, transferiert dieser die Daten auf den *Migration Server*, auf welchem alle Daten sofort dem *Migration Manager* übergeben werden. Dieser sichert zunächst alle erhaltenen Daten, damit bei einem Verbindungsabbruch keine Zustandsinformationen verloren gehen (siehe Abschnitt 4.3.4, Punkt 7). Anschließend ermittelt der *MM* anhand der *Geräte-ID* den Typ des Zielgerätes und sucht im *CoRe*, unter Berücksichtigung des ermittelten Gerätetyps sowie der ID der Ausgangskomponente, nach einer funktional äquivalenten UI-Komponente. Die detaillierten Informationen, bezüglich des Gerätetyps z. B., enthält der *MM* von dem *Communication Manager*, welcher diese Daten bereits im ersten Schritt der UI-Migration (vgl. Vorbetrachtungen, Abschnitt 4.3.4) registriert hat.
An dieser Stelle wird noch einmal hervorgehoben, dass ausschließlich IDs einzelner

Endgeräte bzw. Komponenten zwischen den *UI-Migration-Managern* ausgetauscht werden und diese dann von dem jeweiligen „Verantwortlichen“ der Phase interpretiert werden, um Zugriff zu den eigentlichen Daten zu erhalten. Diese Designentscheidung wurde getroffen, da der Fokus der Arbeit auf der Extraktion sowie Injektion der Zustandsdaten auf Clientseite liegt. Auf eine detailliertere Betrachtung der serverseitigen Funktionslogik soll an dieser Stelle verzichtet werden.
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:

- **Komponentendeskriptor** 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 *Komponentendeskriptor*, 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.
Auf Clientseite wird der Komponentendeskriptor sowie das „Schlüsselwort“ dem *Component Manager* übergeben, welcher sofort die Integration der UI-Komponente im Kontext des Zielgerätes in die Wege leitet. Die ID und alle Zustandsinformationen werden zum *State Handler* des Zielgerätes weitergereicht, welcher für die *Wiederherstellung des Komponentenzustandes*, nach Abschluss der Integration, zuständig ist.
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
  - Methodenaufruf
  - 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:

1. **Blockierung der UI-Komponente** Unter der Annahme, dass die UI-Komponente durch den *Component Manager* erfolgreich im Kontext des Zielgerätes integriert wurde, beginnt der Prozess der Zustandswiederherstellung mit der Pausierung dieser Komponente. Der Grund für das Abwarten der abgeschlossenen Integration der UI-Komponente auf dem Zielgerät, resultiert aus der Annahme, dass spätestens zu diesem Zeitpunkt alle Elemente der Komponente vollständig geladen wurden. Der Vorgang ist dabei identisch mit dem Blockierungsprozess (4) der konzeptionierten Zustandserfassung. Aus diesem Grund entfällt an dieser Stelle eine detaillierte Beschreibung dieses ersten Prozessschrittes mit dem Verweis auf Abschnitt 4.6.2.

2. **Vorbereitung der Zustandsinjektion** Ein essentieller Bestandteil für eine erfolgreiche Zustandswiederherstellung ist der ***UIStateInjector***. Dieser wird, ähnlich wie der *UIStateObserver*, von dem *Component Manager* für jede UI-Komponente initialisiert und übernimmt alle kommenden Aufgaben bezüglich der Zustandsinjektion. Nachdem die *Benutzeroberfläche gesperrt* und die

Komponenten-Blockierung von dem *Component Manager* erfolgreich durchgeführt werden konnte (*„ready-for-injection“*-Event), beginnt der eigentliche Injektionsprozess unter Zuhilfenahme des *UIStateBuffers*.
Jeder einzelne Eintrag des Puffers wird von dem *State Handler* ausgelesen und an den *UIStateInjector*, welcher analog zum *UIStateObserver* innerhalb der UI-Komponente agiert, weitergegeben. Bevor der nächste Prozessschritt durchgeführt werden kann, muss der *UIStateInjector* anhand des Puffereintrages erkennen, ob Werte von UI-Elementen aktualisiert werden müssen (3a), oder der Zustand eines Intervall-Timers wiederhergestellt werden soll (3b). Diese Entscheidung wird anhand des Aufbaus eines Puffereintrages getroffen anhand zuvor definierter Schlüsselwörter.

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 übereinstimmender Bezeichnung gesucht und eine der folgenden Anweisungen ausgeführt:

a) Bei erfolgreicher Ermittlung eines Elementes ersetzte den aktuellen Wert des Objektes mit den in dem Puffer hinterlegten Daten. Nach Abschluss dieser Aktion, signalisiere dem *State Handler*, dass der Zustand des UI-Elementes geändert worden ist.

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* Für den Fall, dass die an den *UIStateInjector* übergebenen Zustandsdaten für die Reaktivierung eines Intervall-Timers vorgesehen sind, wird analog zum vorherigen Verfahren anhand der *Bezeichnung bzw. ID des Timers* ein übereinstimmendes Objekt innerhalb der UI-Komponente gesucht. Bei einem gefundenen Objekt wird im nächsten Schritt die Bezeichnung sowie die Eingabeparameter der vom Timer aufzurufenden Funktion verglichen. Des Weiteren wird zusätzlich die Länge des Zeitintervalls als Vergleichsmerkmal genutzt. Stellt der *UIStateInjector* fest, dass der ermittelte Intervall-Timer mit den im *UIStateBuffer* hinterlegten Informationen ausgeführt werden kann, wird im nächsten Schritt der Zustand des Timers der Ausgangskomponente wiederhergestellt. Sollte kein passendes Objekt innerhalb der Komponente ermittelt werden oder tritt ein Fehler bei der Überprüfung der Parameter auf, wird dies dem *State Handler* signalisiert, welcher den entsprechenden Eintrag im *UIStateBuffer*, analog zur Injektion von Zustandsdaten in UI-Elementen, markiert.

Eine detaillierte Beschreibung der genauen Umsetzung dieser Teilkonzepte folgt im folgenden Kapitel 5.

4. **Statusmeldung für den Benutzer** Nachdem alle Einträge des *UIStateBuffers* abgearbeitet worden sind, übergibt der *State Handler* dem *clientseitigen Migration Manager* die Bezeichnungen aller UI-Elemente bzw. Funktionen, bei denen die Zustandsinjektion *nicht* durchgeführt werden konnte. Aus diesen Daten generiert der *MM* eine Statusmeldung für den Anwender, sodass dieser über die fehlenden Zustandsdaten informiert werden kann. Die Notwendigkeit dieses Prozessschrittes besteht darin, das Ergebnis des Migrationsvorganges für den Anwender so transparent wie möglich zu gestalten. Für den Fall, dass die UI-Migration komplikationslos durchgeführt werden konnte, wird lediglich ein kleines Fenster eingeblendet, in welchem der Nutzer über die erfolgreiche Migration seiner Ausgangskomponente informiert wird.

5. **Reaktivierung der UI-Komponente** Der letzte Prozessschritt der Zustandsinjektion wird ausgeführt, sobald der Anwender die im vorherigen Prozessschritt beschriebene Statusmeldung auf dem Zielgerät bestätigt hat. In diesem Zusammenhang wird die migrierte UI-Komponente unter Zuhilfenahme des *Component Managers* wieder aktiviert.

Dieser Abschnitt hat den Prozess der Zustandsinjektion beschrieben und hat in diesem Zusammenhang alle einzelnen Teilschritte anhand der Abbildung A.5 detailliert erläutert. Zum Abschluss dieses Kapitels wird der nachfolgende Abschnitt noch einmal alle Kernaspekte zusammenfassen und somit einen umfassenden Überblick über die Konzeption der vorliegenden Arbeit geben.

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 grenzte 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 **Zustandsmodelles** beschrieben und dabei auf den Aspekt der ***Kategorisierung von Zustandsinformatio-***

nen eingegangen. Es wurde festgelegt, dass im Gegensatz zu den Style- und Layoutinformationen einer Komponente, ausschließlich der Zustand der Wertemenge sowie Zustände aktiver Intervall-Timer für das Konzept dieser Arbeit von Bedeutung sind.
Im Fokus des darauffolgende Abschnittes stand die **Extraktion der Zustandsinformationen**. In diesem Zusammenhang wurde das ***Mutation-Event-Detection***-Verfahren vorgestellt, welches das Ergebnis einer vorangegangenen Diskussion, bezüglich der Wahl einer geeignete Extraktionsmethode, darstellte.
Der nächste Abschnitte erläuterte den **Transfer der extrahierten Zustandsdaten**. 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 **Zustandsinjektion** - wobei ähnlich wie in dem Abschnitt der Zustandsextraktion, zunächst noch einmal die gegebenen Voraussetzungen beschrieben wurden, bevor unter Zuhilfenahme 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 Prototyp und somit das Konzept validiert.

5 Implementation

Nachdem im vorangegangen Kapitel 4 ein Konzept für die Migrationsinfrastruktur in CRUISe erarbeitet wurde, folgt in diesem Kapitel die Vorstellung der prototypischen Implementierung dieses Ansatzes. In diesem Zusammenhang werden grundlegende implementierungstechnische Entscheidungen dokumentiert und entsprechend begründet. Anhand einer Beispielanwendung soll der Prototyp validiert werden.

5.1 Das verwendete Implementierungsframework

Die *Thin Server Runtime* des CRUISe-Ansatzes setzt bereits mehrere Technologien ein, welche somit auch für die Implementierung der Migrationsinfrastruktur von Bedeutung sind. Als grundlegende Sprache wurde angesichts der Browserumgebung und dem Verbreitungsgrad *JavaScript* gewählt. Als Framework, welches die wichtigsten Funktionen zur clientseitigen Anwendungsentwicklung, wie z. B. Mittel zur objektorientierten Programmierung, ereignisbasierte Kommunikation oder die Manipulation des DOM-Baumes bereitstellt, wurde *ExtJS 3.4*[1] verwendet.

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-Managern, wobei aufgrund der Übersichtlichkeit nicht alle Verbindungen zwischen den einzelnen Teilkomponenten der TSR eingezeichnet wurden [Pie12]. Die **gelb** 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] http://www.sencha.com/products/extjs

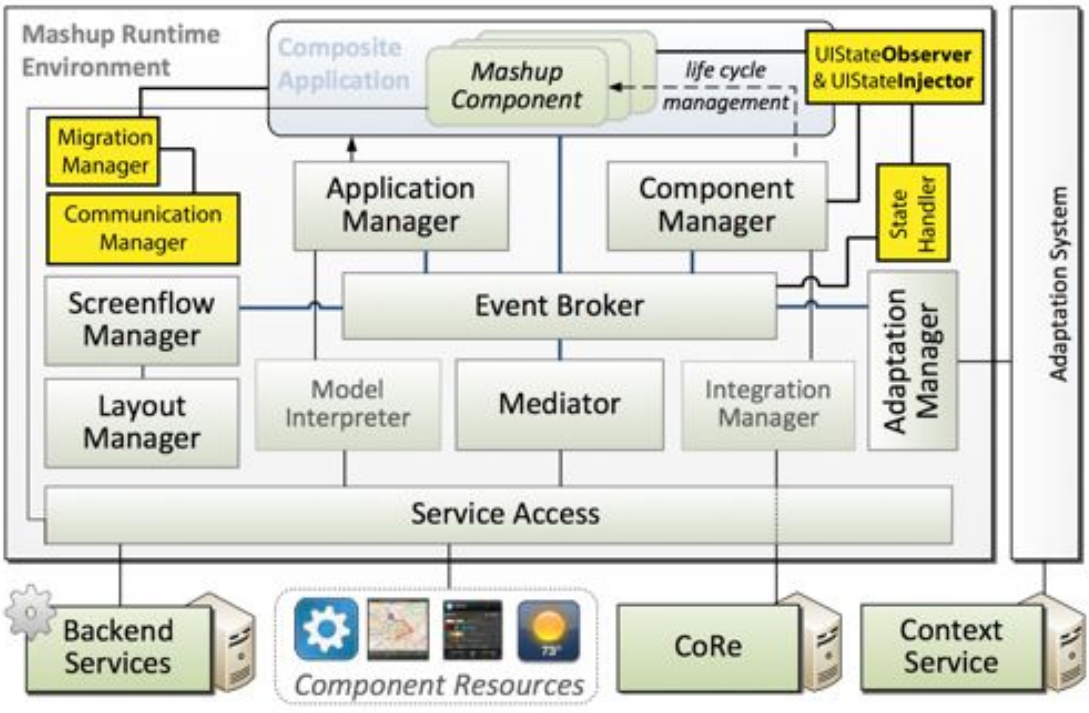

Abbildung 5.1: Erweiterung der CRUISe-Thin-Server-Runtime

zu den UI-Komponenten. Wie in Kapitel 4, Abschnitt 4.6.2 und 4.8 bereits erläutert wurde, agieren diese beiden Objekte *innerhalb* einer UI-Komponente und extrahieren bzw. injizieren zustandsrelevante Daten. Weitere Zusammenhänge werden im Laufe dieses Kapitels verdeutlicht und an dieser Stelle nicht detaillierter betrachtet. Der nächste Abschnitt befasst sich mit dem *Application Manager*, welcher an mehreren Stellen erweitert wurde, um die UI-Migration in CRUISe zu ermöglichen.

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:

```
1 ...
2 constructor: function(config){
3 Ext.apply(this, config, {
4   ...
5   useMigration: false
6 });
7 ...
```

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*:

```
...
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:

```
...
// 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 „stateEventInjSONformat“ dem zuvor definierten Kanal zuzuordnen:

```
if(this.useMigration) this.eventBroker.addEventToChannel('
    stateEventInjSONformat', '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:

```
1 if(this.useMigration) this.eventBroker.subscribe(undefined,
    undefined, 'object', 'stateChangeReceivedChannel', this.
    stateHandler.processStateChange, this.stateHandler);
```

Listing 5.5: Registrierung eines Event-Handlers

Abschließend wird der *Migration Manager* durch Aufruf von this.migrationManager.start() gestartet. Diese Funktion des *Migration Managers* erstellt das User-Interface mit dessen Hilfe der Anwender die UI-Migration konfigurieren kann. Dabei wird es dem Benutzer ermöglicht eine UI-Komponente der integrierten Web-Anwendung anzuwählen sowie ein Zielgerät, auf welchem die Komponente migriert werden soll.
Nachdem dieser Abschnitt einleitend die notwendigen Erweiterung der Thin-Server-Runtime vorgestellt hat, folgen im nächsten Abschnitt Implementierungsdetails bezüglich der clientseitigen Zustandserfassung.

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:

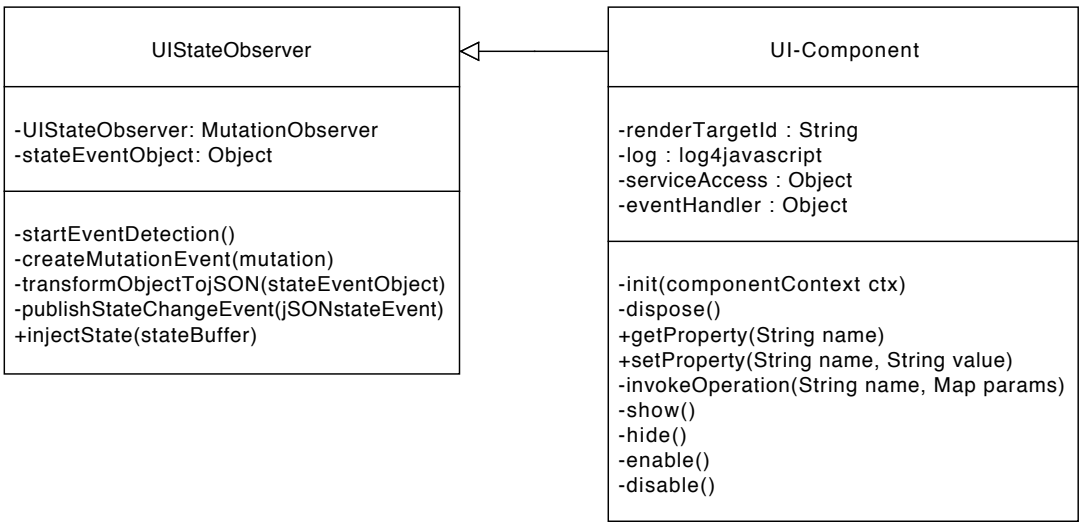

Abbildung 5.2: Klassendiagramm der JavaScript-basierten Basisklasse

Um jegliche Migrationsfunktionalitäten zu erhalten, müssen alle UI-Komponenten

von dieser Basisklasse erben. Sobald die Funktion startEventDetection aufgerufen wird, beginnt die Erfassung aller auftretenden Änderung der DOM-Baum-Struktur. Der Methodenaufruf erfolgt entweder am Ende der Initialisierung oder wird an den Anfang der obligatorischen show()-Methode, welche jede UI-Komponente in CRUISe besitzt, geschrieben. Demzufolge wird der eigentliche Lebenszyklus einer Komponente nicht verändert, da sich die Basisklasse nahtlos in die vorhandene Infrastruktur integriert. Eine detaillierte Betrachtung der einzelnen Methoden sowie die Funktion der Variablen folgt in den nächsten Abschnitten.

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:

```
1  // create an UIStateObserver instance
2  var UIStateObserver = new MutationObserver(function(mutations) {
3      mutations.forEach(function(mutationEvent) {
4          if(mutationEvent.target.type) {
5
6              // create MutationEventObject
7              var stateEventObject = createMutationEvent(mutationEvent);
8
9              // transform object to jSONString
10             var jSONstateEvent = JSON.stringify(stateEventObject);
11
12             // publish StateEvent to runtime
13             publishStateChangeEvent(jSONstateEvent);
14         }
15     });
16 });
17
18 // select the target node
19 var target = document;
20
21 // configuration of the observer:
22 var config = {
23     subtree: true,
24     attributes: true,
25     characterData: true,
26 };
27
28 // pass in the target node, as well as the observer options
29 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 Konfigurationsparameter 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 aussortiert werden. Dies wird durch Aufruf der Methode createMutationEvent(mutation) (7) realisiert:

```
1  function createMutationEvent(mutationEvent) {
2      // create an empty object
3      var stateObject = new Object();
4
5      // fill object
6      stateObject["type"] = mutationEvent.target.type;
7      stateObject["name"] = mutationEvent.target.name;
8      stateObject["id"] = mutationEvent.target.id;
9      stateObject["value"] = mutationEvent.target.value;
10
11     // return object
12     return stateObject;
13 }
```

Listing 5.7: Kategorisierung der signalisierten Zustandsveränderungen

Die Funktion bekommt als Eingabe ein *mutationEvent* und wandelt dieses zunächst in ein stateEvent**Object** 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:

```
1  function publishStateChangeEvent(stateEvent_jSON) {
2      var message = new Ext.cruise.client.Message();
3      message.setName("stateEventInjSONformat");
```

```
4     message.appendToBody("stateEvent", stateEvent_jSON);
5     eventHandler.publish(message);
6 }
```

Listing 5.8: Bekanntmachung von Zustandsveränderungen

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 programmiertechnisch 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

Nachdem die extrahierten Zustandsdaten vom Ausgangsgerät, über den Migration Server, bis hin zum Zielgerät übertragen und an den *State Handler* übergeben wurden, kann der Prozess der Zustandsinjektion gestartet werden. Dies setzt voraus, dass die UI-Komponente zuvor erfolgreich durch den *Component Manager* im Kontext des Zielgerätes integriert wurde und dass sowohl die Komponente als auch das User-Interface der Web-Anwendung für eingehende Nachrichten blockiert wurden. Sind diese Voraussetzungen erfüllt, so kann der Injektionsprozess gestartet werden. Die Umsetzung des Konzeptes der Zustandsinjektion wird anhand des Codebeispiels 5.9 erläutert. In diesem Zusammenhang werden die einzelnen Arbeitsschritte des zugrundeliegenden Algorithmus nach Abbildung des Quellcodes beschrieben.

```
1  injectStateToComponent: function(stateBuffer) {
2     for (var i=0; i<stateBuffer.length; i++) {
3        if(stateBuffer[i]["id"]) {
4           var id = stateBuffer[i]["id"];
5           var value = stateBuffer[i]["value"];
6           var selectedDOMelement = document.getElementById(id);
7           selectedDOMelement.value = value;
8        } else { alert("Unable to locate position in DOM –
             StateBufferEntry doesn't have an ID.") }
9     }
10 }
```

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.

Diese beschriebene Methode zur Injektion von Zustandsdaten setzt voraus, dass die „IDs“ der Elemente der Ausgangskomponente identisch mit denen der Zielkomponente sind. Für den Fall, dass kein Element mit der entsprechenden ID innerhalb des DOM-Baumes gefunden werden kann, wird die Zustandsinjektion abgebrochen. Dieser Abschnitt hat die Umsetzung des Prozesses der Zustandsinjektion bei JavaScript-basierten UI-Komponenten beschrieben. Bevor Abschnitt 5.6 dieses Kapitel abschließend zusammenfasst, wird im nächsten Abschnitt die entwickelte Beispielanwendung überblicksweise präsentiert.

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 welchem 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.

Sowohl der erste, als auch der letzte Button dienen lediglich zur Präsentation der *prototypischen* Implementation, da sie einen kleinen Einblick hinter die Kulissen der Komponente gewähren. Beide Buttons werden in der finalen Version der Migrationsinfrastruktur von CRUISe nicht integriert sein.

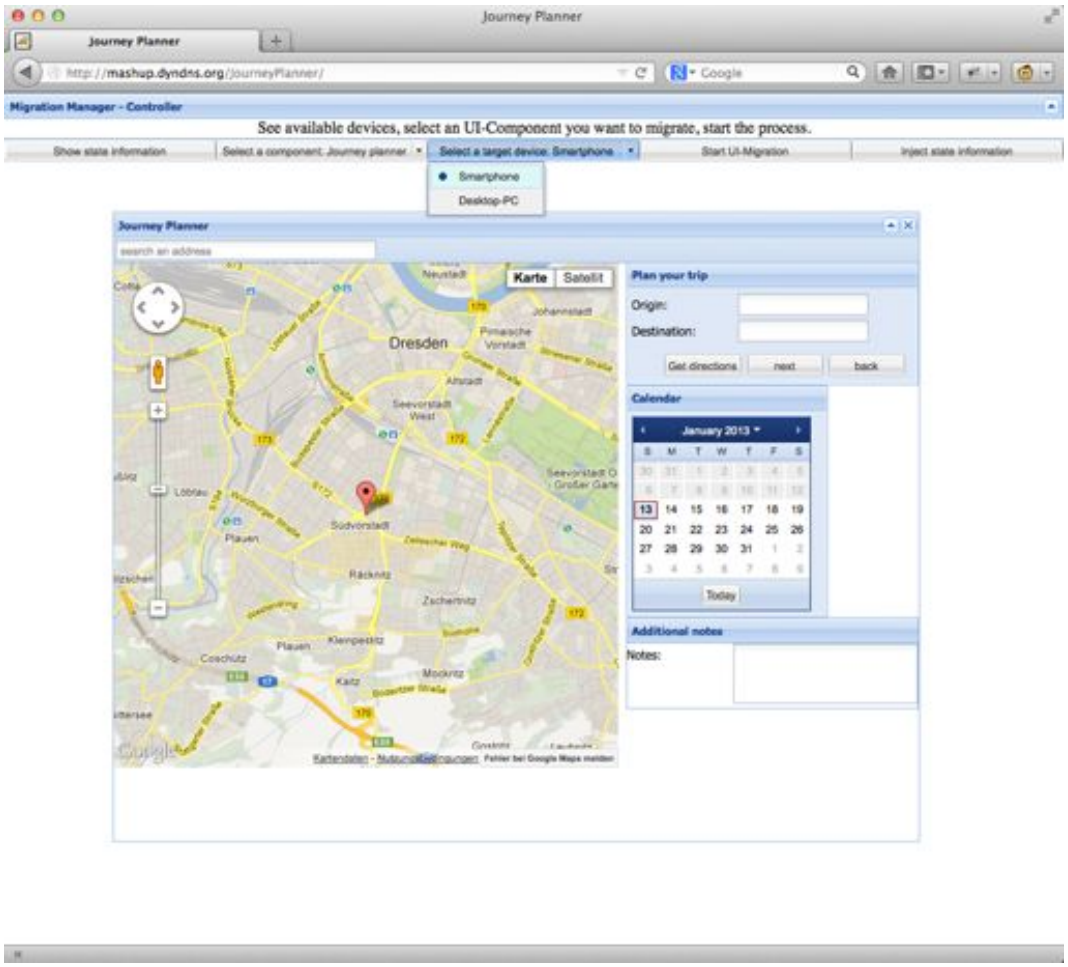

Abbildung 5.3: Screenshot der Beispielanwendung

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

Das letzte Kapitel der vorliegenden Arbeit soll die erreichten Ergebnisse präsentieren und abschließend bewerten. Des Weiteren werden offene Problemstellungen aufgezeigt und Anregungen gegeben, welche in zukünftigen Arbeiten im Forschungsprojekt CRUISe unter dem Aspekt der UI-Migration aufgegriffen werden können.

6.1 Ergebnisse

Das Ziel dieser Arbeit war die Konzeption eines Verfahrens zur **Zustandsextraktion bzw. -injektion**, um die UI-Migration im Forschungsprojekt CRUISe zu ermöglichen. Des Weiteren sollte in diesem Zusammenhang ein **technologie- und plattformneutrales Zustandsmodell** entworfen werden. Dazu wurden in Kapitel 2 zunächst grundlegende Begriffe wie *UI-Migration* und *Softwarekomponente* definiert und von anderen Technologien abgegrenzt. Dies diente der Identifikation zu lösender Aufgaben, zeigte bevorstehende Herausforderungen und fokussierte den Schwerpunkt der vorliegenden Arbeit. Nachfolgend wurde die CRUISe-Architektur überblicksweise vorgestellt, wobei das Komponentenrepositorium, die Laufzeitumgebung sowie die bestehenden Defizite von CRUISe in Bezug auf die UI-Migration beschrieben wurden.
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.
Auf Grundlage des vorherigen Kapitels stand in Kapitel 4 die **Konzeption** der Migrationsinfrastruktur im Vordergrund, wobei in diesem Zusammenhang zunächst eine *Anforderungsanalyse* aufgestellt wurde. Danach folgte eine **Evaluation** der untersuchten Ansätze auf Basis eines entworfenen *Kriterienkataloges*. Im Anschluss daran wurde das konzeptionierte Verfahren präsentiert. Nach der Beschreibung eines *Anwendungsszenarios* und der Vorstellung wichtiger Teilkonzepte der UI-Migration, folgten essentielle *Vorbetrachtungen*. Schließlich wurde der **Aufbau des entwickelten Zustandsmodelles**, die **Extraktion der Zustandsinformationen**, der **Transfer dieser Daten vom Ausgangs- zum Zielgerät** sowie die **Wiederherstellung des UI-Komponentenzustandes im Kontext des Zielgerätes** ausführlich diskutiert. In diesem Zusammenhang wurde ein Modell zur *Kategorisierung von Zustandsdaten* vorgestellt und dabei die Funktionalität jeder einzelnen Kategorie definiert. Im Anschluss daran wurde erörtert, warum es oft nicht notwendig ist, alle theoretisch erfassbaren Zustandsinformationen zu migrieren.
Die **prototypische Umsetzung** des Konzepts behandelte Kapitel 5. Mit der erfolgreichen Implementierung eines Großteils des Konzeptes und der darauf aufset-

zenden Beispielanwendung konnte das Konzept validiert werden. In Bezug auf die in Kapitel 4 aufgestellten Anforderungen, kann festgehalten werden, dass im Kontext der prototypischen Implementierung nicht alle Punkte vollständig umgesetzt werden konnten. Die *Übertragung der Informationen* wird im derzeitigen Prototyp beispielsweise nicht unterstützt. Eine explizite *Kategorisierung der Zustandsdaten* wird zum aktuellen Zeitpunkt nicht vorgenommen. In diesem Zusammenhang werden lediglich Änderungen des *UI-Zustandes* registriert und gepuffert, nicht aber Style- und Layoutinformationen. Die letzte Anforderung, welche bis zu diesem Zeitpunkt nicht umgesetzt wurde, ist die vollständige *Plattformunabhängigkeit*. Die Basis des Konzeptes lässt sich zwar auf alle gängigen Programmiersprachen übertragen, wurde aber ausschließlich in *JavaScript* umgesetzt.

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.

1. **Semantische Annotation der *Properties*** Der erste Lösungsvorschlag arbeitet mit der SMCDL-Beschreibung, welche für jede UI-Komponente existieren muss. Laut Definition enthält diese Datei alle Angaben bezüglich der *Properties*, *Events* und *Operationen* einer Komponente. Unter der Annahme, dass die UI-Komponente des Ausgangsgerätes bspw. nicht für andere Gerätetypen im CoRe zur Verfügung steht, muss eine funktional äquivalente Komponente migriert werden, welcher allerdings von einem anderen Entwickler implementiert wurde und somit völlig andere Variablenbezeichnungen besitzt. An dieser Stelle könnte eine Art *Semantic Matching* der Properties beider Komponenten durchgeführt werden, um festzustellen welcher Wert der Ausgangskomponente in welche Variable der Zielkomponente migriert werden muss. Voraussetzung für dieses Verfahren ist zum einen, dass der Komponentenentwickler alle zustandsrelevanten Properties in die SMCDL-Beschreibung eingepflegt hat. Des Weiteren müssen diese semantisch annotiert sein, damit das eben beschriebene „Mapping“ durchgeführt werden kann.

2. ***Transformation* der extrahierten *Zustandsinformationen*** Eine weitere Möglichkeit heterogene UI-Komponenten für die UI-Migration in CRUISe zu unterstützten, 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 Eine weitere Möglichkeit das vorgestellte Konzept zu erweitern, ist die Umsetzung des *UIStateObservers* in weiteren Programmiersprachen. In dieser Arbeit wurde die Aufgabe des UIStateObservers, welche darin besteht alle auftretenden Zustandsveränderungen zu registrieren und zu serialisieren, in *JavaScript* umgesetzt. Das Grundprinzip beruht dabei auf Techniken bzw. Entwurfsmuster wie „*Observer*“ oder „*Listener*“, welche in allen gängigen Programmiersprachen unterstützt werden.

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.

A Anhang

	Nutzbarkeit des erfassbaren Zustandes	Entwicklungsaufwand für den Komponentenentwickler	Effizienz	Entwicklungsaufwand zur Erweiterung der TSR	Besonderheiten
CRUISe	unzureichend → ausschließlich nicht-flüchtige Eigenschaften einer Komponente erfasst und keine Kategorisierung möglich	hoch → alle Methoden müssen vom Komponentenentwickler selbst implementiert werden	keine Einschätzung möglich	keine Einschätzung möglich	(+) Isolations-Mechanismus (+) 2-Phasen-Commit-Protokoll (+) Proxy-Entwurfsmuster (+) Semantic Matching
CoBRA	unzureichend → Verwendung von Java *Serializable*; Granularität und Kategorisierung abhängig vom Komponentenentw.	hoch → Komponentenentwickler muss Zustandsvariablen komplett eigenständig definieren	gering → Serialisierung muss immer komplett durchgeführt werden	gering → Erweiterungsaufwand wird auf Komponentenentwickler übertragen	(+) Memento-Entwurfsmuster
OPEN	gut → Serialisierung des *DOM-Baums* und Ansätze zur Erfassung von JS-Variablen sowie Möglichkeit der Kategorisierung	gering → OPEN-Adaptoren stellen Migrationsfunktionalitäten bereit	hoch → Serialisierung des DOM-Baums und Erfassung der JS-Variablen ist schnell	hoch → OPEN-Plattform ist für vollständige Integration in CRUISe zu komplex	(+) Generierung abstrakter UI-Beschreibungen mit Hilfe einer Reverse Engineering-Technik
COSMOD	unzureichend → zufällige Eingabeparameter führen Methoden einer Komponente aus; keine Kategorisierung möglich	hoch → Testsystem für Komponenten notwendig / muss vom Entwickler aufgesetzt werden	gering → polynomiale Zeitkomplexität (abhängig von der Mächtigkeit der Komponente)	hoch → Implementierung eines Testsystems ; unvollständiger Pseudo-Quellcode	
Event-Observer	gut → Registrierung aller modifizierten UI-Elemente sowie deren Werte; Möglichkeit der Kategorisierung	gering → unter Voraussetzung, class ist Programmiersprache abstrakte Klassen für die Event-Registrierung bereitgestellt werden	hoch → sowohl die in JS verwendeten *Mutation Observer*, als auch *Listener* anderer Programmiersprachen sind schnell	hoch → für jede Programmiersprache müssen einmalig abstrakte Klassen implementiert werden	(+) Übertragbar auf andere Programmiersprachen wie beispielsweise Flash

Abbildung A.1: Evaluation der untersuchten Forschungsprojekte

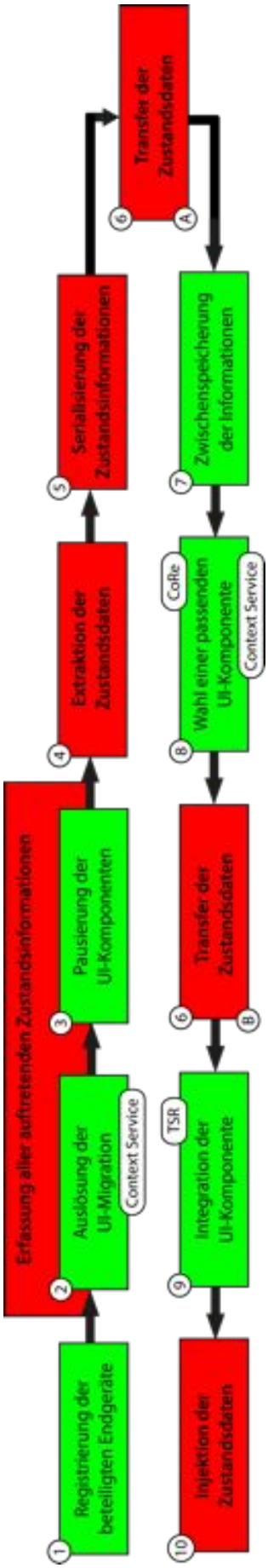

Abbildung A.2: Grundlegende Prozessschritte der UI-Migration

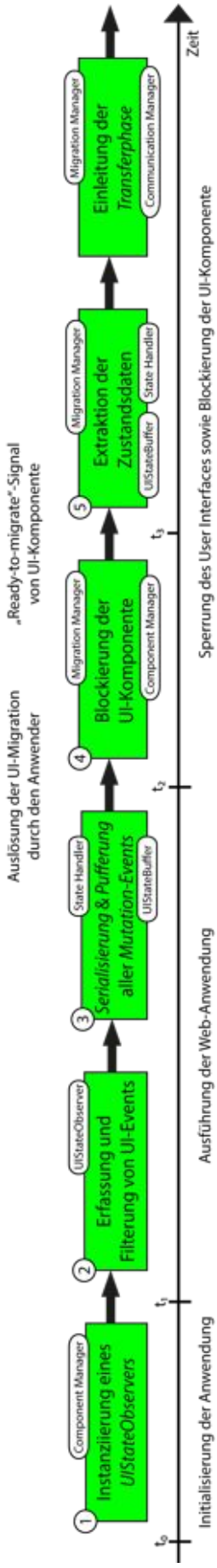

Abbildung A.3: Prozessschritte der konzeptionierten Zustandserfassung

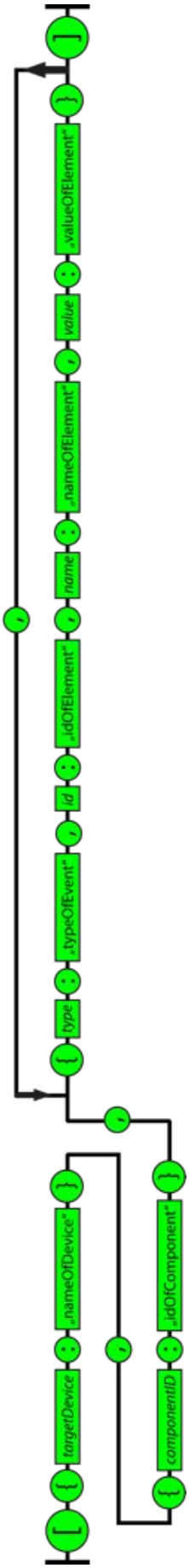

Abbildung A.4: Aufbau des Kommunikationsprotokolls

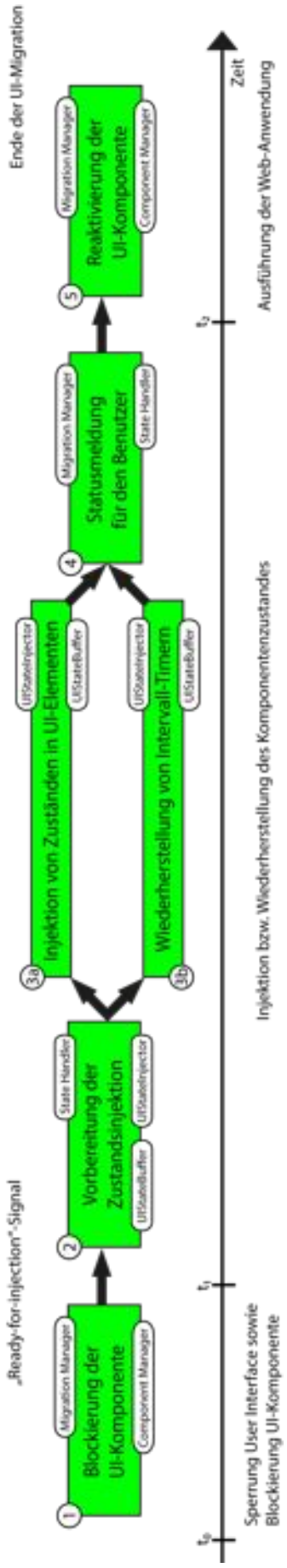

Abbildung A.5: Prozessschritte der konzeptionierten Zustandsinjektion

Literaturverzeichnis

[Ann12] Lachlan Hunt Anne van Kesteren Aryeh Gregor. *DOM4*. Dez. 2012. URL: http://www.w3.org/TR/dom/#introduction-to-dom-events (siehe Seite 41).

[Böh09] Thomas Böhm. »Grundlagen komponentenorientierter Softwareentwicklung«. In: (Mai 2009), Seiten 1–35 (siehe Seite 13).

[BP04] Renata Bandelloni und Fabio Paternò. »Flexible interface migration«. In: *Proceedings of the 9th international conference on Intelligent user interfaces*. IUI '04. Funchal, Madeira, Portugal: ACM, 2004, Seiten 148–155. ISBN: 1-58113-815-6. DOI: 10.1145/964442.964470. URL: http://doi.acm.org/10.1145/964442.964470 (siehe Seite 7).

[CD98] S Clemens und G Dominik. *Component software: beyond object-oriented programming*. 1998 (siehe Seiten 4, 12).

[Dan+07] Florian Daniel, Jin Yu, Boualem Benatallah u. a. »Understanding UI Integration: A Survey of Problems, Technologies, and Opportunities«. In: *IEEE Internet Computing* 11.3 (Mai 2007), Seiten 59–66. ISSN: 1089-7801. DOI: 10.1109/MIC.2007.74. URL: http://dx.doi.org/10.1109/MIC.2007.74 (siehe Seite 4).

[DSB99] Desmond D'Souza, Aamod Sane und Alan Birchenough. »First-class extensibility for UML packaging of profiles, stereotypes, patterns«. In: *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard* (1999) (siehe Seite 12).

[Gam+95] Erich Gamma, Richard Helm, Ralph E. Johnson u. a. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995. ISBN: 978-0-201-63361-0 (siehe Seiten 22, 25, 28–30, 33).

[GM05] Arun Mukhija Glinz und Martin. »Runtime Adaptation of Applications Through Dynamic Recomposition of Components«. In: (Feb. 2005), Seiten 1–15 (siehe Seite 23).

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

[IFM08] F. Irmert, T. Fischer und K. Meyer-Wegener. »Runtime adaptation in a service-oriented component model«. In: *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems* (2008), Seiten 97–104 (siehe Seiten 23, 27–29).

[Irm11] Florian Irmert. »Eine Infrastruktur für die Modularisierung und Laufzeitadaption bei Datenbanksystemen«. In: (Juni 2011), Seiten 1–254 (siehe Seite 49).

[Lev12] Nicole Leverich. *New research: Global surge in smartphone usage, UK sees biggest jump*. Jan. 2012. URL: http://googlemobileads.blogspot.de/2012/01/new-research-global-surge-in-smartphone.html (siehe Seite 1).

[Mei11] Klaus Meißner. »Vorlesung Web- and Multimedia-Engineering - 9. Mashup-Technologie«. 2011 (siehe Seite 18).

[Mer06] Duane Merrill. *Mashups: The new breed of Web app*. Aug. 2006. URL: http://www.ibm.com/developerworks/xml/library/x-mashups/index.html (siehe Seite 4).

[Mul12] Professur für Multimediatechnik - TU Dresden. *DoCUMA - Vision und Ziele*. Aug. 2012. URL: http://www.mmt.inf.tu-dresden.de/Forschung/Projekte/DoCUMA/visionandgoals.xhtml (siehe Seite 1).

[Nic+10] Anders Nickelsen, Fabio Paternó, Agnese Grasselli u. a. »OPEN: open pervasive environments for migratory interactive services«. In: *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*. iiWAS '10. Paris, France: ACM, 2010, Seiten 639–646. ISBN: 978-1-4503-0421-4. DOI: 10.1145/1967486.1967585. URL: http://doi.acm.org/10.1145/1967486.1967585 (siehe Seiten 9, 31).

[Nic12] Yoshino Nickolay. *MutationObserver*. Dez. 2012. URL: https://developer.mozilla.org/en-US/docs/DOM/MutationObserver?redirectlocale=en-US&redirectslug=DOM/DOM_Mutation_Observers (siehe Seite 41).

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

[Pat11] Fabio Patern. »Migratory Interactive Applications for Ubiquitous Environments, 1st edition«. In: *Migratory Interactive Applications for Ubiquitous Environments, 1st edition* (März 2011) (siehe Seiten 31–33, 35, 36, 52).

[Pie09] Stefan Pietschmann. »A Model-Driven Development Process and Runtime Platform for Adaptive Composite Web Applications«. In: (2009) (siehe Seiten 1, 17).

[Pie12] Stefan Pietschmann. »Modellgetriebene Entwicklung adaptiver, komponentenbasierter Mashup-Anwendungen«. In: (Apr. 2012), Seiten 1–276 (siehe Seite 81).

[PPM10] Stefan Pietschmann, Vincent Tietz Jan Reimann Christian Liebing Michèl Pohle und Klaus Meißner. »A Metamodel for Context-Aware Component-Based Mashup Applications«. In: (Sep. 2010), Seiten 1–8 (siehe Seiten 15, 16).

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

[Rad11] Carsten Radeck. »Unterstützung semantischer Komposition in Mashups«. In: (Feb. 2011), Seiten 1–132 (siehe Seiten 4, 18, 22, 23, 26, 74).

[Res12] Catalyst Resources. *Best Practices Guide for Cross-Platform/Cross-Device Development*. 2012. URL: http://www.catalystresources.com/cross-device-ui/ (besucht am 2012) (siehe Seite 5).

[Sch11] Steffen Schurig. »Entwicklung und Evaluation mobiler Anwendungsszenarien in CRUISe«. In: (Feb. 2011), Seiten 1–70 (siehe Seiten 16, 18).

[SM09] RR Suman und R Mall. »State Model Extraction of a Software Component by Observing its Behavior«. In: *ACM SIGSOFT Software Engineering Notes* (2009) (siehe Seiten 40, 41, 53).

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

[Tho98] Anne Thomas. »Enterprise JavaBeans«. In: (1998) (siehe Seiten 14, 15).

[WP11] Bastian Wollschläger und Christoph Pohl. »Vergleich von Universal Plug and Play und Bonjour im Kontext ubiquitärer Systeme«. In: (Juni 2011), Seiten 1–16 (siehe Seite 62).