Generierung von interaktiven OSGi-Komponenten für Android

Diplomarbeit

am Institut für Systemarchitektur

vorgelegt von:	Georg Berndt
Studiengang:	Medieninformatik
Hochschullehrer:	Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
Betreuer:	Dipl.-Inf. Marius Feldmann

Dresden, 2010

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Diplomarbeit mit dem Thema

Generierung von interaktiven OSGi-Komponenten für Android

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Dresden, den 30.08.2010

Georg Berndt

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Nachdem sich der Hype der letzten Jahre bezüglich Service-Oriented Architecture (SOA) gelegt hat, ist SOA mittlerweile zu einem fundamentalen Paradigma in der Softwareentwicklung für die Realisierung verteilter Geschäftsprozesse geworden und findet zunehmend mehr Akzeptanz. Die Ursache des Hypes lag unteranderem an einer Überschätzung der kurzfristigen Erfolge und Einsatzgebiete von SOA als neue Technologie und einer gleichzeitigen Unterschätzung der langfristigen Auswirkungen kleinerer Änderungen [Mas07]. Der Kerngedanke von SOA ist, unterschiedliche Systeme über Dienste lose miteinander zu koppeln und dadurch neue Anwendungen mittels Komposition zu entwickeln [ZF09]. Zum Zugriff auf die Funktionalitäten eines solchen Dienstes, besitzt dieser eine wohldefinierte veröffentlichte Schnittstelle. Nach [Lie07] ist ein Dienst eine sich selbst beschreibende, offene Komponente, die eine schnelle und kostengünstige Zusammenstellung von verteilten Applikationen ermöglicht. Da die Dienstbeschreibungen, für die Kommunikation und den Austausch von Nachrichten zwischen Softwareanwendungen entworfen wurden, werden SOAs mit Hilfe von Web-Services vor allem im Business-to-Business (B2B)-Bereich realisiert. Für Clientanwendungen im Business-to-Consumer (B2C)-Bereich, welche direkt mit dem Dienst interagieren, stellt hingegen das Fehlen eines User Interface (UI) ein fundamentales Problem dar. Das UI in Verbindung mit einer Kontrolllogik, müsste für jede Konfiguration einer solchen dienstbasierten Anwendung mit hohem Aufwand entwickelt werden, wobei bei Änderungen der Dienstbeschreibung, wiederum die dienstbasierte Anwendung angepasst werden müsste. Da sich aber für B2C-Lösungen für Web-Services ein potentieller Markt ergibt, wobei teilweise auf bereits bestehende Dienstinfrastrukturen zurückgegriffen werden kann, bzw. auch [Lie08] die Notwendigkeit für B2C-Lösungen für Web-Services zeigt, ist die Generierung von interaktiven dienstbasierten Anwendungen für Web-Services seit längerem Bestandteil der Forschung. Mit Hilfe von Inferenzmechanismen ist es möglich, ein UI auf alleiniger Basis der funktionalen Dienstbeschreibung abzuleiten. Im Allgemeinen fehlt es solch einem UI an Ausdrucksstärke, da UI-spezifische Eigenschaften, welche zur Erstellung eines aussagekräftigen UI benötigt werden, nicht beschrieben werden konnten. Weitere Ansätze zielten darauf ab, Web-Services um eben solche Beschreibungen in Form von Annotationen [Spi06] zu erweitern. Auf Basis einer annotierten funktionalen Dienstbeschreibung ist es dann möglich, eine servicebasierte Anwendung inklusive des dazugehörigen, reichhaltigen UIs automatisch zu generieren.

1.2. Einordnung in den Gesamtansatz

Im Feld der automatischen Generierung von UIs für Web-Services existieren bereits einige Ansätze. Um aber aussagekräftige UIs zu erhalten, reicht die reine Service-Beschreibung, wie bereit im vorherigen Absatz erklärt, z.B. auf Basis von Web Service Definition Language (WSDL)-Dokumenten (vgl. [CCMW01]), nicht aus. Die Services wurden daher um zusätzliche UI-Beschreibungen erweitert [Spi06]. Eine solche UI-Beschreibung ist im Rahmen des FP7-Projekt *ServFace*[1], welches das Ziel der Entwicklung einer modellgetriebenen integrierten Entwicklungsmethodik für die Erstellung dienstbasierter Anwendungen in Form von ServFace-Annotationen verfolgt, vgl. [Ser09], entstanden. Diese Annotationen ermöglichen es, UI-Fragmente wie z.B. Default-Werte, Autovervollständigung oder eine Eingaben-Validierung zu definieren. Neben den ServFace-Annotationen gibt es aber noch andere Beschreibungsformate, die einige Vor- aber auch wieder Nachteile mit sich bringen. Im Ansatz von [Fel11] wurde ein generisches Annotationsmodell eingeführt, für das mehrere Parser existieren, welche die verschiedenen Beschreibungsformate, unter anderem auch ServFace-Annotationen, in das generische Modell überführen. Ausgehend von einer Instanz eines solchen Modells und der Service-Beschreibung kann mit Hilfe von Modell-zu-Modell (M2M)-Transformationen, basierend auf einer komplexen Inferenz-Logik, eine Instanz eines Meta-Modells zur Beschreibung einer interaktiven dienstbasierten Anwendung erzeugt werden. Da sich diese Anwendungen aufgrund von enthaltenen Meta-Informationen zur Integration in einen gemeinsamen Anwendungskontext eignen, werden diese folgend als *interaktive Komponente* bezeichnet. Beschreibungen solcher interaktiver Komponenten werden im Rahmen dieser Arbeit nach Vorgabe der Themenstellung für die Generierung von OSGi-Komponenten verwendet, welche in eine Zielanwendung integriert werden. Die Zielanwendung soll nach Vorgabe der Themenstellung als *Android*-Anwendung unter Verwendung des *Apache Felix* OSGi-Containers realisiert werden. Der generische Ansatz von [Fel11] ermöglicht zusätzlich die Erweiterung einer bestehenden UI-Beschreibung für Web-Services, da nur der jeweilige Parser angepasst werden muss. Eine solche Erweiterung wird im Rahmen von [Mar10] diskutiert, welche die ServFace-Annotationen um Annotationen zur Beschreibung von Navigations- und Datenflüssen erweitert. Neben dieser Erweiterung ist ein Designtime-Werkzeug entstanden, mit dem funktionale Dienstbeschreibungen um eben diese Annotationen erweitert werden können und im Anschluss in einem Repository veröffentlicht werden. Daraufhin wird durch das Generierungsverfahren, welches Bestandteil von [Fel11] ist, eine Modell-Instanz erzeugt die wiederum die Grundlage für das Generierungsverfahren bildet, das im Rahmen dieser Arbeit entstanden ist.

Diese Arbeit behandelt somit das letzte Glied der Kette zur vollautomatischen Generierungen von interaktiven dienstbasierten Komponenten, deren Veröffentlichung und deren Integration. Bezogen auf den Gesamtansatz wird in dieser Arbeit ein konkreter *Platform Adapter* (vgl. [Fel11], Kapitel 5) für *Android* und die dazugehörige Zielanwendung behandelt. Abbildung 1.1 verdeutlicht den kompletten Ablauf ausgehend von den annotierten Dienstbeschreibungen, einer anschließenden Verfeinerung mit dem in [Mar10] beschrei-

[1] `http://servface.org/`

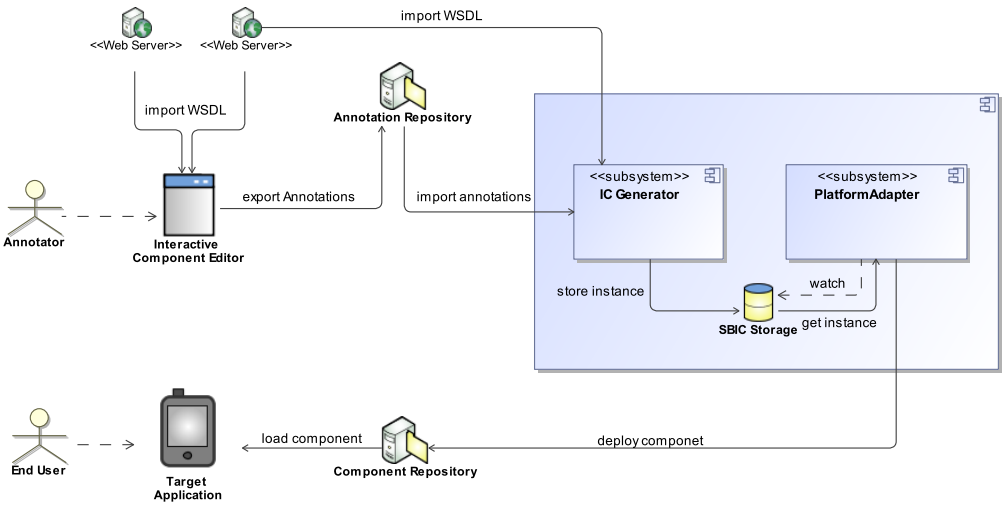

Abbildung 1.1.: Überblick über das Gesamtsystem

ben Werkzeug, der wiederum anschließenden Erzeugung einer Modell-Instanz durch das in [Fel11] beschriebenen Generierungsverfahren, welches die Grundlage zur Generierung interaktiver dienstbasierter Komponenten bildet. Die generierten Komponenten werden dann in einem zentralen Repository veröffentlicht, welches die transparente Integration dieser Komponenten, in eine Zielanwendung unterstützt.

Als Grundlage für die Generierung der Komponenten und zur Evaluation des Verfahrens wurden verschiedene Anwendungsfälle eines Heimautomatisierungsszenarios in einer Service-Infrastruktur implementiert und anschließend mit ServFace-Annotationen und Annotationen der Erweiterung von [Mar10] versehen.

1.3. Ziele der Arbeit

Folgende zentralen Fragen stellen sich im Rahmen dieser Arbeit:

/F1: Welche System-Architektur ermöglicht eine automatische Generierung von OSGi-Komponenten auf Basis von Instanzen eines Komponenten-Metamodells und einer anschließenden Veröffentlichung der generierten Komponenten.

/F2: Wie ist eine generische Zielanwendung zur transparenten Integration, auf technologischer Basis von Android unter Verwendung eines OSGi-Containers, zu strukturieren bzw. zu gestalten?

/F3: Wie sind die interaktiven Komponenten für die definierte technologische Basis, bestehend aus Android und OSGi, zu gestalten und zu strukturieren?

1.4. Überblick über die Arbeit

Diese Arbeit gliedert sich in fünf Kapitel bestehend aus *Analyse* (2), *Stand der Technik* (3), *Konzeption und Umsetzung* (4), *Evaluation* (5) und *Zusammenfassung und Ausblick* (6).

Kapitel 2 stellt die Grundlagen dieser Arbeit vor. Zunächst wird das Anwendungsszenario vorgestellt, im Anschluss werden, um die genannten zentralen Fragen beantworten zu können, die Kernanforderungen an das Gesamtsystem verdeutlicht. Die ermittelten Charakteristika werden in einer Stand-der-Technik-Analyse in Kapitel 3 mit bereits bestehenden Ansätzen verglichen und differenziert, nachdem ein kurzer Überblick über einige Grundlagen auf dem Gebiet der Generierung von OSGi-Komponenten für Android-Anwendungen gegeben wurde. Nach diesen vorbereitenden Schritten wird in Kapitel 4 die Konzeption des Gesamtsystems veranschaulicht und die Umsetzung ausgewählter Details präsentiert. Das vorletzte Kapitel (5) validiert die Funktionalität des entwickelten Prototypen, der generierten Komponenten und des Gesamtsystems anhand der aufgestellten Anforderungen und des eingeführten Szenarios. Abschließend fasst Kapitel 6 die grundlegenden Inhalte und Ergebnisse der gesamten Arbeit zusammen und gibt einen Ausblick auf mögliche Erweiterungen und Verbesserungen der Lösung.

2. Analyse

In diesem Kapitel wird zunächst ein Anwendungsszenario eingeführt und eingesetzte Dienste vorgestellt. Weitergehend beschäftigt sich dieses Kapitel mit der Analyse der Anforderungen an das System zur Generierung interaktiver Komponenten sowie den Anforderungen an die Zielanwendung.

2.1. Szenariobeschreibung

Als Szenario zur Evaluierung der konzipierten Modell-zu-Code (M2C)-Transformation und der dazugehörigen Zielanwendung wurde ein Heimautomatisierungsszenario, welches aus heterogenen verteilten Geräten in einem Heimnetzwerk besteht, vorgegeben. Die einzelnen Geräte bieten hierbei ihre Funktionalitäten als Web-Services an, wobei SOAP als Kommunikationsprotokoll zur Übertragung der Daten und zum Ausführen der *Remote Procedure Calls* bzw. dem Nachrichtenaustausch dient.

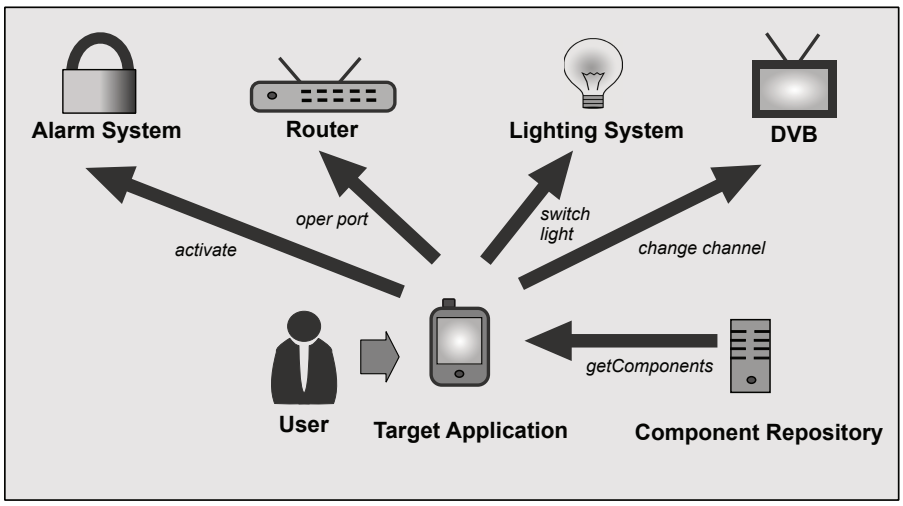

Abbildung 2.1.: Heimautomatisierungsszenario - Überblick

Die Benutzerschnittstellen der einzelnen Geräte werden in Form von interaktiven Komponenten in einem zentralen Verzeichnis veröffentlicht, welches Bestandteil des Heimnetzwerks ist. Aus diesem Verzeichnis, ist es der Zielanwendung möglich, die veröffentlichten interaktiven Komponenten zu laden und zu integrieren. Die Zielanwendung setzt sich somit aus einer Menge von interaktiven Komponenten zur Steuerung der einzelnen Geräte zu einer komplexen Anwendung zusammen, wobei sich die integrierten Komponenten nahtlos in das UI der Zielanwendung einfügen. Abbildung 2.1 bietet einen Überblick über

die einzelnen Bestandteile des Heimautomatisierungsszenarios. In Abbildung 2.2 wird die nahtlose Integration interaktiver Komponenten in die Zielanwendung, im Hinblick auf eine einheitliche Visualisierung der integrierten Komponenten, veranschaulicht.

Abbildung 2.2.: Zielanwendung - Integration interaktiver Komponenten

DVB-Information-Service

Der *DVB-Information-Service* stellt Funktionalitäten im Kontext eines DVB-Receivers bereit. Die Funktionen erlauben eine entfernte Steuerung der Basisfunktionen eines solchen Gerätes. Abbildung 2.3 veranschaulicht die Anwendungsfälle und die Zustände aus denen diese aufgerufen werden können.

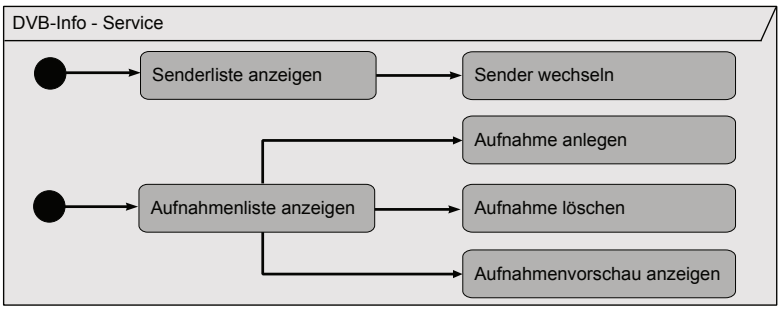

Abbildung 2.3.: DVB-Information-Service - Zustandsdiagramm

Der *DVB-Information-Service* ermöglicht im Detail eine Auflistung der verfügbaren Sender, das Wechseln des aktuellen Senders, als auch das Anlegen und Löschen von Aufnahmen. Zusätzlich kann eine Vorschau einer bereits angelegten Aufnahme angezeigt werden.

Alarm-System-Service

Der *Alarm-System-Service* ermöglicht die Steuerung eines Sicherheitssystems wie z.B. einer Alarmanlage. Nach einer Authentifizierung ist es möglich, das System zu aktivieren bzw. zu deaktivieren oder ein Bild einer Überwachungskamera anzeigen zu lassen. Abbildung 2.4 zeigt die einzelnen Zustände und die damit verbundenen Anwendungsfälle des Services.

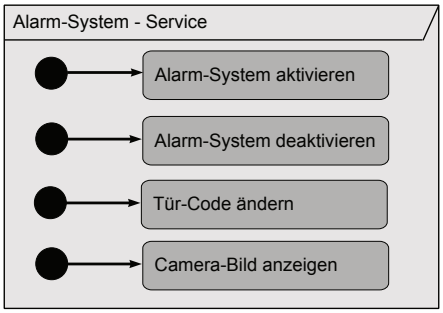

Abbildung 2.4.: Alarm-System-Service - Zustandsdiagramm

Neben dem Aktivieren bzw. dem Deaktivieren der Alarmanlage, kann der Türcode des Alarmsystems geändert werden. Hierfür muss die Aktualisierung mit dem zurzeit eingestellten Türcode bestätigt werden.

Router-Configuration-Service

Der Router-Configuration-Service ermöglicht den Zugriff auf die Basisfunktionalitäten eines DSL-Routers in einem Heimnetzwerk. Nach einer Authentifizierung ist es möglich Portfreigaben anzulegen, zu löschen oder zu ändern.

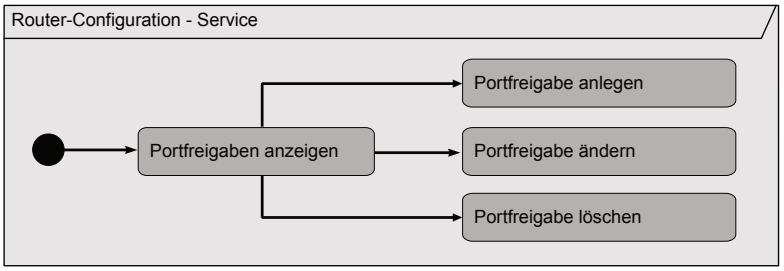

Abbildung 2.5.: Router-Configuration-Service - Zustandsdiagramm

Lighting-Service

Der Lighting-Service ermöglicht die Steuerung von voneinander unabhängigen Lichtquellen, die über mehrere Räume verteilt sind. Jede Lichtquelle bietet ihre Funktionalitäten, welche aus einer Operation zum Ein- bzw. Ausschalten und aus einer Operation zur Statusabfrage bestehen, als eigenständigen Dienst an.

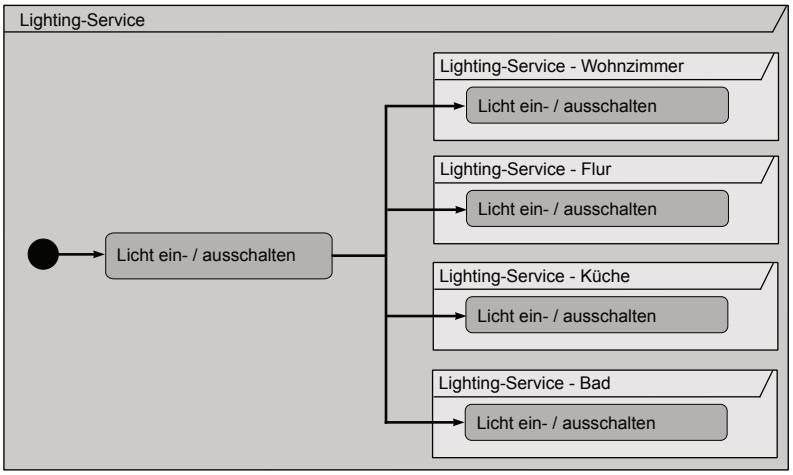

Abbildung 2.6.: Lighting-Service - Zustandsdiagramm

Die Dienste der einzelnen Lichtquellen werden gebündelt und in einen einheitlicher UI präsentiert.

Authentication-Service

Zur Authentifizierung am *Alarm-System-Service*, am *Router-Configuration-Service*, als auch am *DVB-Information-Service* wird ein zentraler Authentifizierungsdienst verwendet. Die Authentifizierung wird hierbei zentral von der Zielanwendung übernommen. Bei einer erfolgreichen Authentifizierung wird der erhaltene Session-Key als Parameter den sicherheitskritischen Operationen des Szenarios übergeben.

2.2. Anforderungen

Die Anforderungen an das komplette System können wie folgt gegliedert werden. Zum ersten in die Basisanforderungen, welche direkt aus der Themenstellung dieser Arbeit hervorgehen, zum zweiten in die Anforderungen, die sich an das allgemeine Generierungsverfahren richten, zum dritten in die Anforderungen bezüglich der Distribution generierter interaktiver Komponenten, zum vierten in die Anforderungen, welche die Zielanwendung betreffen und zum fünften in übergeordneten Anforderungen, welche aus der Einordnung dieser Arbeit in den Gesamtansatz resultieren.

I. Basisanforderungen

Die Basisanforderungen, welche direkt aus der Themenstellung dieser Arbeit resultieren, sind zum einem die Verwendung von *OSGi* zur Entwicklung und Integration interaktiver Komponenten und zum anderen die Verwendung von *Android* als Zielplattform für die Zielanwendung. Teile der Implementierung, bestehend aus einer Android-Anwendung zur Integration von OSGi-Bundles auf Basis von *Apache Felix* und eine Anwendung zum automatischen Starten eines Generierungsprozesses, waren zu Beginn der Arbeit prototypisch vorhanden und bildeten den Ausgangspunkt zur Weiterentwicklung.

II. Anforderungen an das Generierungsverfahren

/AG01/ M2C-Transformation zur Generierung von OSGi-Komponenten

Die zentrale Anforderung an das System, ist die Konzeption und prototypische Umsetzung einer M2C-Transformation zur Generierung von OSGi-Komponenten aus Instanzen eines Meta-Modells interaktiver dienstbasierter Anwendungen.

/AG02/ Erweiterbarkeit der M2C-Transformation

Es soll ein Mechanismus entwickelt werden, der es ermöglicht, ausgehend von der zugrundeliegenden Modell-Instanz, Teile der M2C-Transformation dynamisch zu integrieren, welche zur Generierung spezieller UI-Komponenten benötigt werden. Nach einer Analyse der Modell-Instanz, sollen die benötigten Bestandteile aus einem Repository geladen werden und in die M2C-Transformation integriert werden.

/AG03/ Automatisierung der Generierung

Die Generierung der Komponenten soll automatisch erfolgen, sobald eine neue Modell-Instanz im System vorliegt.

/AG04/ Generalität

Die Generierung der Komponenten soll für beliebige Modell-Instanzen möglich sein, welche aus M2C-Transformation resultieren, die im Rahmen der Dissertation [Fel11] entstanden sind.

/AG05/ Benutzbarkeit der generierten Komponenten

Auf Basis der Informationen in den zur Generierung verwendeten Modellinstanzen, sollen reichhaltige Benutzerschnittstellen generiert werden. Das bedeutet, dass sich die generierten Benutzerschnittstellen nur unwesentlich von Benutzerschnittstellen konventionell erstellter Anwendungen für Android unterscheiden sollen und die gewohnten androidtypischen Verhaltensweisen aufweisen.

III. Anforderungen an die Distribution interaktiver Komponenten

/AD01/ Mechanismus zur Veröffentlichung und Verbreitung generierter Komponenten

Die Komponenten sollen automatisch nach der Generierung in einem Repository veröffentlicht werden. Der Zielanwendung soll es dann möglich sein, die veröffentlichten Komponenten zu laden. Bereits bestehende Komponenten sollen durch die aktuellere Version ersetzt werden.

/AD02/ Versionierung veröffentlichter Komponenten

Beim Laden der veröffentlichten Komponenten sollen bestehende Komponenten durch die aktuellere Version ersetzt werden, falls diese verfügbar ist.

/AD03/ Vertrauenswürdigkeit

Der Verzeichnisdienst soll eine Autorisierung der Zielanwendung unterstützen. Umgekehrt sollen nur Komponenten von vertrauenswürdigen Quellen importiert werden.

IV. Anforderungen an die Zielanwendung

Die folgenden Anforderungen **/AZ01/** bis **/AZ04/** der Zielanwendung lassen sich direkt aus [Fel11] ableiten. In dieser Arbeit wird das Konzept einer Zielanwendung zur Integration interaktiver Komponenten vorgestellt und die dafür notwendige Kommunikation zwischen einer interaktiven Komponente und der Zielanwendung beschrieben.

/AZ01/ Integration der Komponenten

Die generierten OSGi-Komponenten aus dem zentralen Repository sollen von einer Zielanwendung integriert werden können. Dabei werden die Komponenten aus dem zentralen Repository geladen.

/AZ02/ Registrierung der Komponenten

Die integrierten Komponenten müssen in der Zielanwendung registriert werden. Die Daten für die Registration werden von der jeweiligen Komponente bereitgestellt. Während der Registrierung wird dann geprüft ob die Komponenten zum Start von der Zielanwendung zu übergebende Eingabeparameter benötigen, die Zielanwendung zusätzliche Plattformfunktionalitäten bereitstellen muss oder die Komponenten einem bestimmten Kontext angehören. Falls die Komponenten Parameter bereitstellen, wird dies in der Zielanwendung registriert.

/AZ03/ Bereitstellung von zusätzlichen Plattformfunktionalitäten

Auf Basis der Registrierungsdaten kann eine interaktive Komponente der Zielanwendung mitteilen, dass diese zusätzliche Plattformfunktionalitäten für den Start benötigt. Diese Funktionalitäten müssen dann in Form von OSGi-Komponenten nachträglich installiert werden, falls diese nicht vorhanden sind und der jeweiligen interaktiven Komponente als Handle übergeben werden. Die Informationen zum Nachinstallieren einer solchen Funktionalität müssen Bestandteil der Registrierungsdaten einer interaktiven Komponente sein, siehe [Fel11], Kapitel 4.

/AZ04/ Ansteuerung der Komponenten

Zur Ansteuerung der erfolgreich registrierten Komponenten wird ein Mechanismus entwickelt, der es ermöglicht, die Komponenten auf Basis der Registrationsdaten (vgl. [Fel11], Kapitel 4) zu starten und wieder zu beenden.

/AZ05/ Kontextadaptierbarkeit

Die Architektur der Zielanwendung muss gewährleisten, dass eine Zielanwendung einen speziellen Anwendungskontext annehmen kann.

V. Übergeordnete Anforderungen

Die übergeordneten Anforderungen resultieren aus der Eingliederung dieser Arbeit in den Gesamtansatz, vgl. 1.2. Das System zur Generierung von interaktiven Komponenten muss sich in die Prozesskette des Gesamtansatzes in Form eines Plattformadapters für Android integrieren lassen.

3. Stand-der-Technik

In diesem Kapitel werden zunächst in einem Grundlagenabschnitt (3.1) die im Rahmen dieser Arbeit eingesetzten Technologien und das verwendete Metamodell zur Beschreibung interaktiver Komponenten vorgestellt. Im Anschluss werden in Abschnitt 3.2 verwandte Ansätze auf dem Gebiet der automatischen Generierung interaktiver dienstbasierter Anwendungen bzw. Ansätze auf dem Gebiet der Integration von UI-Komponenten in einen gemeinsamen Anwendungskontext betrachtet, deren Charakteristika ermittelt und im Anschluss mit denen dieser Arbeit verglichen.

3.1. Grundlagen

In diesem Abschnitt wird zunächst in Unterabschnitt 3.1.1 *OSGi* als Technologie vorgestellt, welche die dynamische Integration interaktiver Komponenten in einen gemeinsamen Anwendungskontext ermöglicht. *Android*, welches im Rahmen dieser Arbeit als Zielplattform eingesetzt wird, wird in Unterabschnitt 3.1.2 näher beschrieben. In Unterabschnitt 3.1.3 wird das verwendete Metamodell, dessen Instanzen den Ausgangspunkt zur Generierung interaktiver Komponenten bilden, vorgestellt.

3.1.1. OSGi

Die OSGi[2] Service Platform, vgl. [OSG07], ist ein dynamisches Modulsystem für Java und ermöglicht die dynamische Integration und das Management von Softwarekomponenten (Bundles) und Diensten (Services). Ursprünglich für eingebettete Systeme konzipiert, wird die OSGi Service Platform heute vielfältig eingesetzt, von Anwendungen für Mobilfunkgeräte über Client-Anwendungen wie der Eclipse IDE bis hin zu Server-Applikationen [WHKL08]. Die OSGi-Plattform setzt dafür eine Java Virtual Machine (JVM) voraus und bietet darauf aufbauend das OSGi-Programmiergerüst, in welchem die Komponente, das *Bundle*, im Vordergrund steht. Java bietet als einzige Umgebung die benötigte Portabilität, um viele verschiedene Plattformen unterstützen zu können. Um eine relativ lose Kopplung der Komponenten zu erreichen und um diese auch verwalten zu können, basiert die Service-Plattform auf einer SOA. Die Bundles sind so in der Lage Services zu registrieren, zu finden, zu binden, die Bindung zu lösen und werden benachrichtigt, sobald ein Service registriert oder beendet wurde. Ein Service ist im OSGi-Umfeld lediglich ein JAVA-Interface, welches für die Registrierung und die Kommunikation der Komponenten

[2] http://www.osgi.org

untereinander benutzt wird. Das Bundle stellt die Implementierung des jeweiligen Services bereit. Es wird somit die Trennung von Spezifizierung und Implementierung erreicht. Ein weiteres wesentliches Merkmal der Service-Plattform ist die Möglichkeit, die Services bzw. Bundles zur Laufzeit einzuspielen, zu aktualisieren und auch wieder zu entfernen. Bundles, welche von entfernten Bundles abhängig waren, werden automatisch gestoppt und erst dann wieder gestartet, sobald alle Abhängigkeiten wieder aufgelöst sind. Vor allem dieses Merkmal führte dazu OSGi, zur dynamischen Integration von generierten Komponenten für diese Arbeit zu verwenden.

3.1.2. Android

Android ist ein Betriebssystem sowie auch eine Software-Plattform für mobile Geräte wie Smartphones, Mobiltelefone und Netbooks, die von der Open Handset Alliance entwickelt wird. Die Architektur von Android baut auf dem Linux-Kernel 2.6 auf. Er ist für Speicherverwaltung, Prozessverwaltung und die Netzwerkkommunikation zuständig. Außerdem bildet es die Hardwareabstraktionsschicht für den Rest der Software und stellt die Gerätetreiber für das System. Ein wesentlicher Hauptbestandteil der Android-Plattform ist die *Dalvik Virtual Machine*, eine eigens für mobile Geräte entwickelte JVM. Im Gegensatz zur herkömmlichen JVM, welche auf einen Kellerautomaten basiert, arbeitet die Dalvik-VM als Registermaschine. Das Modell der Registermaschine ist wesentlich besser geeignet im Hinblick auf den begrenzten Speicher und Leistung mobiler Geräte. Dies hat aber zur Folge, dass herkömmlicher Java-Bytecode auf der Dalvik-VM nicht ausführbar ist. Mit dem Werkzeug *dx*, welches Bestandteil des Android-SDKs ist, kann aber der herkömmliche Java-Bytecode in das dex-Format (Dalvik Executable) überführt werden und so auf der Dalvik-VM zur Ausführung gebracht werden. Seit der Version 2.2 ist ein JIT-Compiler Bestandteil der Dalvik-VM, welcher für eine zwei- bis fünffache Verbesserung der Ausführungszeit sorgt. Die Nutzung von *Apache Felix* als OSGi-Container ist durch leichte Anpassungen möglich [Cle08]. Hierbei müssen Bundles, welche auf einem Android-Gerät deployed werden, neben dem Java-Bytecode, den Bytecode für die Dalvik-VM enthalten.

3.1.3. Meta-Modell zur Beschreibung interaktiver Komponenten

Als Meta-Modell zur Generierung wird das im Rahmen von [Fel11] entstandene Service-based Interactive Component-Model (SBIC) verwendet. Mit diesem Modell lassen sich dienstbasierte interaktive Komponenten beschreiben, welche in eine Rahmenanwendung integriert werden können. Das Generierungsverfahren von [Fel11] kann, mit Hilfe verschiedener Regelsätze, Instanzen erzeugen, die an verschiedene Anforderungen einer Zielapplikation angepasst werden. Die im Rahmen dieser Arbeit verwendeten Instanzen wurden für *Android* als Zielplattform generiert.

Das SBIC gliedert sich in sechs Packages: *component*, *view*, *viewflow*, *globaldata*, *dependency* und *layout*.

- Das *component*-Package bildet das Herzstück und kapselt den Einstiegspunkt für Instanzen des Meta-Modells und stellt die Verbindung zu den jeweiligen anderen Packages her.
- Das *asm*-Package dient zur Beschreibung des zugrundeliegenden Service-Modells und beinhaltet eine abstrakte Service-Beschreibung und die Annotationen mit denen die Dienste annotiert wurden.
- Das *view*-Package dient zur Modellrepräsentation eines User-Interfaces, bestehend aus View-Containern und UI-Elementen.
- Das *viewflow*-Package dient zur Modellrepräsentation des Daten- und Navigationsflusses zwischen den Elementen des *view*-Packages. Dabei ist es möglich, mit einem Navigationsfluss eine Menge von Operationsaufrufen zu assoziieren.
- Das *context*-Package dient der Beschreibung eines Zielkontextes, welcher aus einer Menge von Kontextelementen besteht. Ein Kontextelement besitzt einen Bezeichner und einen Wert. Ist zum Bespiel eine SBIC-Instanz für die Plattform *Android* und die Sprachen *Deutsch* und *Englisch* inferiert, könnte der Zielkontext wie folgt aussehen. $\{(platform, android), (language, german), (language, english)\}$
- Mit Hilfe des *globaldata*-Packages können globale Datenrelationen beschrieben werden. Global bedeutet in diesem Sinne, dass Daten von einer externen Quelle bezogen werden können oder Daten der interaktiven Komponente von außerhalb abgerufen werden können. Anders ausgedrückt, ist es möglich zu beschreiben, dass eine Rückgabe einer Operation oder eine Eingabe einer Operation global verfügbar gemacht wird bzw. dass Daten, welche von einer externen Quelle stammen, als eine Eingabe für einen Operationsaufruf verwendet werden.
- Das *dependency*-Package dient zur Modellierung von externen Abhängigkeiten. Konkret kann so beschrieben werden, dass einzelne Werte für einen Operationsaufruf von externen Funktionen bereitgestellt werden.
- Mit Hilfe des *Layout*-Packages können Layout-Parameter für die Elemente des *view*-Packages beschrieben werden.

Die Relation der einzelnen Packages untereinander wird durch Abbildung 3.1 deutlich, welches das *component*-Package veranschaulicht. Wie bereits beschrieben, bildet die *InteractiveComponent* das Herzstück des SBICs, welche Referenzen auf Elemente der eben vorgestellten Packages enthält. Zusätzlich beinhaltet die *InteractiveComponent* einen Namen und einen eindeutigen Bezeichner in Form eines Universally Unique Identifier (UUID).

Im Folgenden soll das *view*-Package näher betrachtend werden. Zentrale Rolle spielt hier die *View*-Klasse, wobei ein *View* mehreren Service-Operationen zugeordnet sein kann, aber immer eine Operation höchstens einem *View* zugeordnet ist. Da es sich um dienstbasierte Komponenten handelt, welche durch eine SBIC-Instanz repräsentiert werden, sind in einer solchen Instanz Sichten zur Ein- und Ausgabe von Service-Operationen in Form von *View*-Instanzen enthalten. Jede *View*-Instanz enthält hierbei einen Container, welcher eine

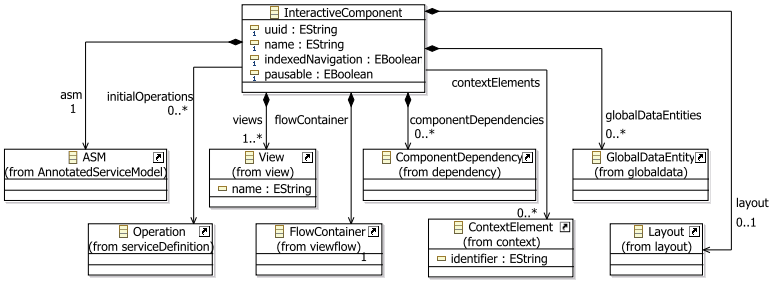

Abbildung 3.1.: SBIC - Component-Package

Menge von *ContentContainern*, *SwitchContentContainern* bzw. *Wizards* enthalten kann, wobei der *ContentContainer* als einziger in der Lage ist UI-Elemente zur Ein- und Ausgabe bzw. Aktivatoren zu gruppieren, da der *SwitchContentContainer* und der *Wizard*, *Container* mit spezifischer Laufzeitsemantik darstellen, die letztendlich selbst wieder *ContentContainer* beinhalten. Die UI-Elemente zur Ein- und Ausgabe werden durch einen *ServiceInteractor* repräsentiert, welcher einen genauen Typ enthält und zusätzlich noch relevante Annotationen beinhalten kann. Mit Hilfe des Typs und den zusätzlichen Annotationen kann die entsprechende Repräsentation des Service-Interaktors für die jeweilige Zielplattform generiert werden, welche in diesem Fall *Android* ist. Das Generierungsverfahren von [Fel11] sorgt hierbei für die Bestimmung des Typs und das Setzen der relevanten Annotationen auf Basis der mit Annotationen versehenen Dienstbeschreibungen. Akivatoren, in einer SBIC-Instanz als *Activator*-Instanzen repräsentiert, sind in der Lage, einen Operationsaufruf auszulösen. Dies geschieht mit Hilfe eines *Flows*, welcher Bestandteil des *viewflow*-Packages ist. Ein Aktivator besitzt ebenfalls einen genauen Typ.

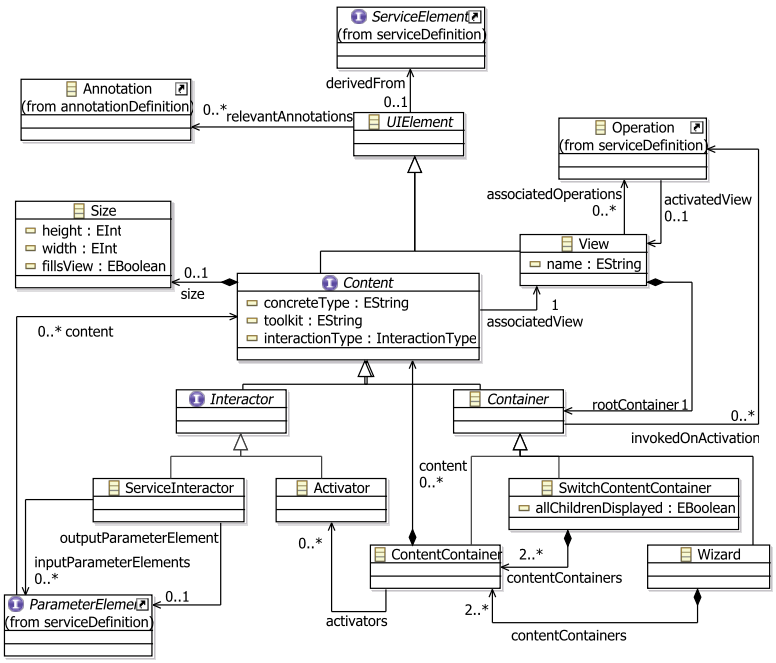

Abbildung 3.2.: SBIC - View-Package

Als letztes soll das *Viewflow*-Package vorgestellt werden. Wie bereits beschrieben, ist ein *Activator* in der Lage einen *Flow* auszulösen. Ein *Flow* kann dabei mehrere *FlowTransitions* enthalten, die zusätzlich noch an eine Bedingung geknüpft sein können, enthält aber mindestens immer eine standardmäßige *FlowTransition*. Eine solche *FlowTransition* repräsentiert einen Navigationsfluss, von einem *View* zu einem anderen oder einen Fluss zwischen zwei *ContentContainern* eines *Views*. Zusätzlich kann eine solche *FlowTransition* eine *DataFlowDefinition* enthalten, um einen Datenfluss von einer Sicht zu einer anderen definieren zu können. Die Navigationsflüsse der SBIC-Instanzen, die im Rahmen dieser Arbeit verwendet werden, resultieren zum einen aus der Servicedefinition, d.h. das auf eine Eingabesicht einer Operation deren Ausgabesicht folgt, und zum anderen können Navigations- und vor allem Datenflüsse in Form von Annotationen beschrieben werden, welche im Rahmen der Arbeit von [Mar10] spezifiziert werden. Das Generierungsverfahren von [Fel11] sorgt hierbei für Paginierung und das Erstellen der jeweiligen Navigations- bzw. Datenflüsse.

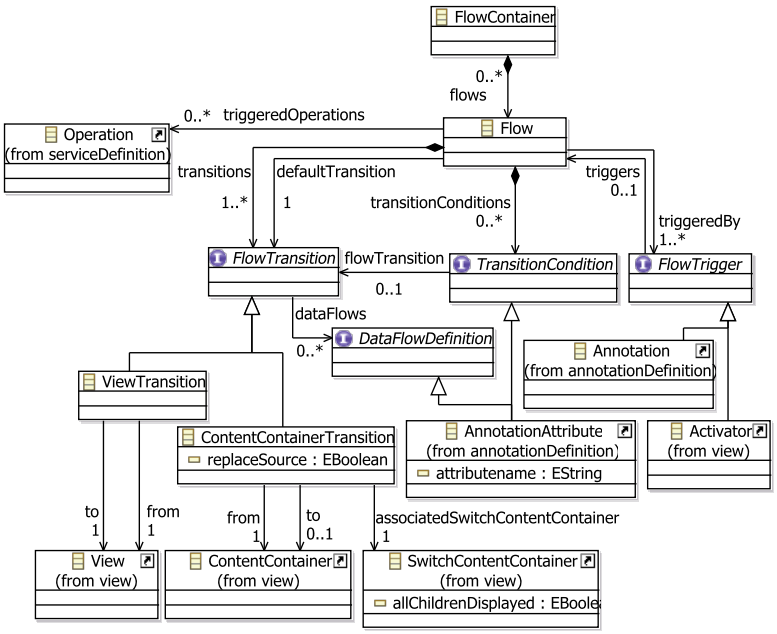

Abbildung 3.3.: SBIC - Viewflow-Package

Auf die hier vorgestellten Teile des SBICs wird im Zuge der Erläuterungen zur Konzeption der M2C-Transformation erneut eingegangen. Eine genaue Definition des SBICs kann in [Fel11], Kapitel 4 nachgelesen werden.

3.2. Verwandte Ansätze

In den folgenden Unterabschnitten werden verwandte Ansätze in Bezug auf diese Arbeit vorgestellt, deren Charakteristika ermittelt und im Anschluss mit denen dieser Arbeit verglichen. Zunächst werden einzelne Ansätze auf dem Gebiet der Benutzerschnittstellengenerierung für Web-Services präsentiert, wobei der Schwerpunkt auf der resultierenden

Präsentationsform liegt, als auf den Beschreibungsformaten bzw. dem Generierungsverfahren an sich. Exemplarisch wird hierbei anhand von Metawidget auf eine andere Domäne eingegangen. Daraufhin werden bereits bestehende Ansätze auf dem Gebiet der Integration von UI-Komponenten in einen gemeinsamen Anwendungskontext betrachtet und bewertet.

3.2.1. WSGUI und Dynvoker

Das Konzept zur Generierung von UIs für Web-Services wurde erstmals von [**?**] an der Stanford University als Projekt *Web Services Graphical User Interface (WSGUI)* vorgestellt. Dieses Projekt war eines der ersten Ansätze in denen zusätzliche UI-Beschreibungen in Form von Annotationen verwendet wurden. Das Konzept wurde in Form des *Dynvokers* [Spi06] später weiterentwickelt. Die verwendeten Annotationen werden in Form von GUI-Deployment-Descriptor (GUIDD)-Dokumenten gemeinsam mit der funktionalen Dienstschnittstellenbeschreibung veröffentlicht und dienen neben der funktionalen Dienstschnittstellenbeschreibung zur Generierung einer XForm-basierten, vgl. [Boy09], Benutzeroberfläche. Ein GUIDD enthält Formular-Komponenten, welche XForm-Elemente auf Message-Instanzen eines WSDL-Dokuments abbilden. Zur Ausgabe können verschiedene *outputTypes* definiert werden, die die Beschreibung von verschiedenen MIME-Typen unterstützen. Weiterhin können zusätzliche Deskriptoren von Service- und Operationselementen definiert werden, um die jeweiligen Elemente mit menschenlesbaren Informationen zu versehen. Auf diese Weise kann auch eine mehrsprachige Anwendung realisiert werden. Eine weitere Charakteristik von GUIDD ist die Möglichkeit zur Angabe von Style-Sheets, welche eine Anpassbarkeit des generierten UI ermöglichen. Um den eigentlichen Service-Aufruf zu ermöglichen, benötigt die WSGUI-Engine das WSDL- und das dazugehörige GUIDD-Dokument als Eingabe. Durch Verwendung eines XSL-Transformation (XSLT)-Prozessors, wird ein Eingabe-Formular generiert, welches dann dem Nutzer bereitgestellt wird. Nach dem Zurücksenden der Nutzereingaben an die Engine, werden diese in eine Web-Service-Request-Message transformiert und an den Web-Service gesendet. Web-Service-Reponse-Messages werden daraufhin in eine aussagekräftige Repräsentation in Form einer Webseite transformiert.

Das Konzept von *WSGUI* bzw. *Dynvoker* benötigt zur Interaktion mit dem eigentlichen Web-Service immer eine Netzinfrastruktur bestehend aus der WSGUI-Engine. Neben diesen Umstand kann immer nur mit einem einzigen Service interagiert werden. Insgesamt entstehen recht einfache und es können keine Navigations- und Datenflüsse modelliert werden.

3.2.2. ServFace-Projekt

Das *ServFace-Project* verfolgt das Ziel der Erstellung einer modellgetriebenen Entwicklungsmethodik für einen integrierten Entwicklungsprozess für dienstbasierte Anwendungen. Dabei wurde ein Annotationsmodell entwickelt, welches eine umfangreiche Sammlung

von Dienstannotationen beinhaltet. Diese Annotationen in Verbindung mit den funktionalen Dienstbeschreibungen bieten die Grundlage für die Generierung von dienstbasierten Anwendungen, was den zweiten Aspekt der entstandenen Entwicklungsmethodik des *ServFace-Projects* darstellt. Bei der Generierung verfolgt das Projekt wiederum zwei verschiedene Ansätze, welche im folgenden getrennt voneinander betrachtet und ausgewertet werden. Der Schwerpunkt liegt hierbei auf den Beschreibungsformaten, welche den Ausgangspunkt zur Generierung von dienstbasierten Anwendungen bilden.

3.2.2.1. ServFace Builder und CAM

Der *ServFace Builder* ist ein Design-Time-Werkzeug zur Erstellung interaktiver dienstbasierter Anwendungen, wobei die verschiedenen Dienste bzw. Teile davon in einer visuellen Komposition zu einer neuen Anwendung zusammengesetzt werden. Um die UI-Elemente zur visuellen Komposition erzeugen zu können, beinhaltet der *ServFace Builder* eine *Inference-Engine*. Die resultierende Anwendung, welche mit Hilfe des Werkzeugs erstellt wurde, wird als Instanz des Composite Application Model (CAM) gespeichert, welches ebenfalls im Rahmen des *ServFace-Projects* entstanden ist. Dieses Modell, dessen Instanzen den Ausgangspunkt für die Generierung dienstbasierter Anwendungen für verschiedene Plattformen bilden, soll im Folgenden näher betrachtet werden. Das CAM ermöglicht die Beschreibung von Seiten, welche abstrakte UI-Elemente beinhalten. Die UI-Elemente, auch als *Interaktoren* bezeichnet, können hierbei in Containern organisiert werden. Der Navigationsfluss der Anwendung wird mittels *FlowTransitions* beschrieben, wobei jede *FlowTransition* eine Start- und eine Zielseite besitzt. Durch eine an die Transition geknüpfte Bedingung kann auf eine alternative Seite verwiesen werden. Ausgelöst wird eine solche Transition durch einen definierten Interaktor, wobei neben der Navigation auch ein Aufruf einer Dienstoperation getätigt werden kann. Weiterhin im CAM enthalten ist das Annotations- als auch ein Service-Modell. Im Hinblick auf das in dieser Arbeit verwendetet Anwendungsmodell fehlt im CAM die Möglichkeit zur Definition von Meta-Informationen, welche benötigt werden, um interaktive Komponenten beschreiben zu können, die in einem gemeinsamen Anwendungskontext verwendet werden. Ein weiterer Punkt ist, dass bei den existierenden M2C-Transformationen im Gegensatz zum SBIC, die Ein- und Ausgabesichten immer auf verschiedene Seiten verteilt sind und verschiedene Formen von Zustandsübergängen nicht möglich sind.

3.2.2.2. MARIAE und MARIA

MARIAE ist ein Werkzeug, welches *Model-based description of Interactive Applications (MARIA)* zur Beschreibung von Benutzerschnittstellen für Webservices verwendet [PSS09]. Neben der Benutzerschnittstellenbeschreibung werden Task-Modelle, in Form von Concur Task Trees (CTT), vgl. [PMM97], zur Beschreibung einer interaktiven Anwendung verwendet. In einem ersten Schritt wird das Task-Modell in Form eines CTT-Modells erzeugt, welches die gesamte Anwendung durch eine Menge von Anwendungsaktivitäten beschreibt.

Daraufhin werden Bindungen zwischen Aktivitäten und Dienstoperationen hergestellt, um Dienstaufrufe mit festgelegten Systemereignissen zu verbinden. Ausgehend davon kann aus den Informationen des Task-Modells eine erste abstrakte Benutzerschnittstelle erstellt werden. Hierbei können bestehende ServFace-Annotationen der verwendeten Dienste berücksichtigt werden. In weiteren Schnitten kann die bereits entstandene Benutzerschnittstelle verfeinert werden. MARIA bietet dabei die Möglichkeit die Benutzerschnittstellen auf einem abstrakten Niveau für plattformunabhängige Elemente und auf einem konkreteren Niveau für plattformabhängige aber implementierungssprachenunabhängige Elemente zu beschreiben. Weitere Merkmale sind die Beschreibung eines Datenmodells und die Beschreibung eines Event-Modells, welches auf verschieden Abstraktionsniveaus definieren kann, wie auf Ereignisse reagiert werden soll. Mit Hilfe der in MARIA beschriebenen Benutzerschnittstelle und den Service-Definitionen, soll es möglich sein, durch Verwendung von M2C-Transformationen, dienstbasierte interaktive Anwendungen zu generieren. Zum gegenwärtigen Zeitpunkt wurde dies noch nicht nachgewiesen. Wie beim CAM fehlt hier die Möglichkeit zur Beschreibung von Meta-Informationen, welche zur Generierung interaktiver Komponenten genutzt werden könnten, die dann in einem gemeinsamen Anwendungskontext veröffentlicht werden könnten. Eine weitere generelle Schwachstelle ist, dass MARIA ein reines UI-Modell ist und die Bindung zum Service-Modell nur über textuelle Bezeichner des UI-Modells hergestellt wird. Dies kann bei einer ungünstigen Konstellation, z.B. bei mehrfacher Verwendung eines Datentyps durch verschiedene Parameter, zu Problemen führen.

3.2.3. Kawash

Kawash hat eine annotationsbasierte Beschreibungssprache entwickelt, mit Hilfe derer man browserbasierte Webanwendungen für unterschiedliche Zielplattformen beschreiben kann [Kaw04]. Zusätzlich hat er ein Verfahren zur dynamischen Generierung von Benutzerschnittstellen für Dienste vorgestellt. Der Fokus liegt hierbei auf der der Beschreibung angepasster Benutzerschnittstellen mit unterschiedlichen Eigenschaften, wobei nach zwei Kriterien unterschieden wird: nach *UI Functionality* und nach *UI Element Attributes*.

Die UI Functionality wird in Form eines erweiterten endlichen Automaten beschrieben, welcher ein Tripel von *Zuständen*, *Transitionen* und *Relationen zwischen UI-Elementen und Dienstoperationen* definiert.

Die UI Element Attributes dienen der Variation der in der *UI Functionality* definierten UI-Elemente. Das bedeutet, dass z.B. für ein mobiles Endgerät mit einer komfortableren Displaygröße dem Gerät entsprechende Eingabe-Elemente verwendet werden können. Neben dieser Variation können Bezeichner für Eingabefelder und Schaltflächen festgelegt werden und Eingabefelder als Pflichtfelder markiert werden.

Das Verfahren von Kawash ermöglicht das Beschreiben einer Dienstanwendung mittels einer Zustandsmaschine und zusätzlichen Benutzerschnittstellenbeschreibung. Diese Informationen können interpretativ zur Laufzeit auswertet werden, um die jeweilige Benutzer-

schnittstelle des aktuellen Zustands der Anwendung zu visualisieren. In der momentanen Konzeption ist aber nur die Integration eines einzelnen Dienstes möglich, was einen fundamentalen Nachteil darstellt. Wie auch bei den bereits vorgestellten Ansätzen, eignet sich das Beschreibungsformat von Kawash nicht zur Beschreibung von interaktiven servicebasierten Komponenten.

3.2.4. Broll

Im Rahmen der Arbeit von Broll [BSR+07] wurde ein Framework entwickelt, welches die dynamische Generierung von abstrakten Benutzerschnittstellen für Webservices ermöglicht. Auf Basis der abstrakten Benutzerschnittstellenbeschreibungen können konkrete Benutzerschnittstellen im HTML-Format, als auch Benutzerschnittstellen für einen JavaME-Client erzeugt werden. Die verwendeten Web-Services werden hierbei um Service-Annotationen erweitert. Die Basis hierfür bildet die Web Ontology Language for Web Services (OWL-S), vgl. [BHL+04], welche um die *Service User Interface Annotation* Ontologie erweitert wurde, um Service-Elemente mit Informationen, zur automatischen Erzeugung von Benutzerschnittstellen, versehen zu können. Herzstück des Frameworks bildet der *Interaction Proxy*, welcher die Informationen der Servicebeschreibungen mit Hilfe einer integrierten *Apache Coocoon*-Instanz[3] in eine abstrakte Benutzerschnittstellenbeschreibung transformiert. Diese abstrakte UI-Beschreibung kann direkt in einem JavaME-Client interpretiert und visualisiert werden oder in einem weiteren Transformationsschritt in eine XHTML-Beschreibung zur Ausgabe in einem Browser umgewandelt werden.

Ähnlich wie beim Ansatz des Projekts *Dynocation*, werden Benutzschnittstellen für annotierte Dienste mittels XSLT-Prozessoren, welche Bestandteil des *Interaction-Proxys* sind, automatisch erzeugt. Der Proxy muss zusätzlich immer Bestandteil der Netzinfrastruktur sein. Neben diesen Umstand kann immer nur mit einem einzigen Service interagiert werden und es fehlt die Möglichkeit der Integration der Anwendungen in einen gemeinsamen Anwendungskontext.

3.2.5. Metawidget

Metawidget ist in der Lage, bestehende Objekte der Backend-Domain zur Laufzeit zu analysierten und daraus native UI-Komponenten für eine Anzahl von Fontend-Frameworks zu erstellen [KL10]. Hierbei werden nicht alle möglichen Fälle abgedeckt, sondern nur die Fälle in denen standardisierte Formulare zum Einsatz kommen. Metawidget beschreibt sich selbst mit fünf Charakteristika: Inspektion von bestehenden heterogenen Backend-Architekturen, Unterstützung von verschiedenen Arten der Aufbereitung der Ergebnisse der Inspektion, Unterstützung verschiedener UI-Bibliotheken, Unterstützung von verschiedenen Verhaltensweisen verschiedener UI-Elemente und Unterstützung verschiedener

[3] http://cocoon.apache.org/

Layouts. Ausgehend von diesen Charakteristika wurde eine Architektur entwickelt, welchen diesen gerecht wird. Metawidget bildet somit eine Brücke zwischen verschiedenen Frontend-Technologien, wie z.B. *Swing*, *Java Server Faces* bis hin zu *Android*, und verschiedenen Backend-Technologien, wie z.B. XML-Dateien, Java-Annotationen oder Bean-Validation. Der volle Umfang unterstützter Fontend- und Backend-Technologien kann auf [Pro09] nachgelesen werden. Obwohl Metawidget Android als Frontend-Technologie unterstützt [Rum09], ist der Beschreibungsumfang aller unterstützten Backend-Technologien, insbesondere die unterstützten Java-Annotationen, zu dem in dieser Arbeit verwendeten Annotationsmodell sehr gering, zumal nur Ein- oder Ausgabemasken für einzelne Objekte erzeugt werden können und keine UIs für Webservices, da es an der Möglichkeit fehlt die dafür benötigten Navigationsflüsse zu definieren.

3.2.6. Portlets

Portlets sind beliebig kombinierbare Komponenten einer Benutzeroberfläche, die von einem Portalserver, dem Portlet-Container, angezeigt und verwaltet werden, und Fragmente von HTML-Code erzeugen und in einer Portalseite zusammengefügt werden können. Verabschiedete Standards sollen die Erstellung von Portlets ermöglichen, die in jedes Portal integriert werden können, welches die Standards erfüllt. Die Java Portlet Spezifikationen JSR168 [JSRa] und JSR286 [JSRc] ermöglichen die Interoperabilität zwischen verschiedenen Portlets und definiert eine Menge von APIs für die Interaktion zwischen den Portlets und den Portlet-Containern im Hinblick auf Präsentation und Sicherheit.

Die Beziehung zwischen *Portlet* und *Portlet-Container* kann mit der Beziehung zwischen *Interaktiver Komponente* und *Zielanwendung*, wie sie in der Terminologie dieser Arbeit verwendet werden, verglichen werden. Theoretisch wäre somit eine Integration kleiner Anwendungen eines gemeinsamen Kontexts in eine einheitliche Anwendung möglich, jedoch müssten die jeweiligen Portlets immer noch manuell erstellt werden. Ein Ansatz zur Generierung von Portlets zur Benutzerschnittstellenrepräsentation eines Webservices ist derzeit nicht bekannt.

3.2.7. Projekt CRUISe

Composition of Rich User Interface Services (CRUISe) verfolgt das Ziel der serviceorientierten Distribution webbasierter Benutzerschnittstellen, wobei Anwendungslogik und Präsentation vollständig entkoppelt werden [PVRM09]. Die Benutzerschnittstelle setzt sich hierbei aus der Komposition austauschbarer und wiederverwendbarer User Interface Service (UIS) zusammen. Somit werden neben den funktionalen Webservices auch in der Präsentationsschicht Dienste eingesetzt. Diese UISs kapseln reichhaltige, webbasierte UI-Bestandteile mit integrierter UI-Logik und stellen diese über eine Service-Schnittstelle zur Verfügung. Die Gesamt-UI ergibt sich durch die Integration bzw. Komposition solcher Dienste, deren Konfiguration, Data-Binding und gegenseitige Interaktion mit Hilfe einer Kompositionssprache beschrieben wird [CRU10].

So gesehen könnte nach dem CRUISe-Ansatz, jede interaktive Komponente als ein UIS veröffentlicht werden, um dann in einer gemeinsamen Anwendung integriert werden zu können. Die UI-Komponenten, welche ein solcher UIS ausliefern würde, müssten jedoch von Hand implementiert werden, was einen erheblichen Mehraufwand darstellt. Hier muss aber erwähnt werden, dass ein Widget, welches von einem solchen Service ausgeliefert werden kann, der Komplexität einer interaktiven Komponente im Sinne des SBICs nicht gerecht wäre. Ein weiterer Nachteil ist, dass die Zielplattform, in welche die UISs eingebunden werden können, immer auf eine Webanwendung beschränkt ist.

3.2.8. Zusammenfassung

Die vorgestellten Ansätze auf dem Gebiet der Benutzerschnittstellengenerierung unter Einbeziehung von zusätzlichen UI-Beschreibungen und deren bewerteten Charakteristika werden in Tabelle 3.1 zusammengefasst dargestellt.

	Op. in UI	**IC beschreibbar**	**vollautomatisch**	**Ziel UI**
WSGUI / Dynvoker	1	nein	ja	XForms
ServFace Builder	>1	nein	nein	CAM
MARIAE	>1	nein	nein	MARIA
Kawash	1	nein	ja	UIFS
Broll	1	nein	ja	abstractUI
Metawidget	-	nein	ja	Swing, SWT, Android, ...

Tabelle 3.1.: Charakteristika der Ansätze zur UI-Generierung für Webservices

Bei einer zusammenfassenden Betrachtung werden folgende Einschränkungen deutlich:

- Keiner der betrachteten Ansätze bietet die Möglichkeit zur Beschreibung einer interaktiven Anwendung, welche im Sinne einer interaktiven Komponente in einen gemeinsamen Anwendungskontekt integriert werden könnte.
- Lediglich die Ansätze des ServFace-Builders und der MARIAE können mehrere Operationen in einer Ansicht zusammenfassen, wohingegen alle anderen betrachten Ansätze immer nur eine Ein- und Ausgabemaske einer einzelnen Operation darstellen können bzw. im Fall von Metawidget überhaupt keine Ein- oder Ausgabemasken für Dienstoperationen darstellen können.
- Die eben erwähnten Ansätze des ServFace-Builders und der MARIAE bieten jedoch im Gegensatz zu den anderen betrachteten Ansätzen zur Benutzerschnittstellengenerierungen, keine vollautomatische Generierung der Ziel-UI. Beim ServFace-Builder, welcher das CAM als Zielbeschreibung nutzt, muss erwähnt werden, dass eine vollautomatische ad-hoc Erzeugung einer CAM-Instanz auf Basis von ServFace-Annoationen und funktionaler Dienstbeschreibungen in [Ber09] nachgewiesen wurde.

- Der Ansatz von Kawash bietet als einziger die Möglichkeit zur Beschreibung eines Navigationsflusses mit Hilfe von Annotationen, was eine vollautomatische Generierung ermöglicht. In den Ansätzen vom ServFace-Builder und MARIAE können zwar Anwendungen mit spezifischen Navigations- und Datenflüssen modelliert werden, was aber wiederum eine vollautomatische Generierung ausschließt. In allen anderen Ansätzen ist Navigationsstruktur fast oder explizit vorgegeben.

Im Folgenden werden die vorgestellten Ansätze auf dem Gebiet der Integration von UI-Komponenten in einen gemeinsamen Anwendungskontext betrachtet und deren bewerteten Charakteristika in Tabelle 3.2 dargestellt.

	Integration in ZA	**vollautomatische Generierung**	**UI-Format**
Portlets	ja	nein	JSR 168 / JSR 286
Projekt CRUISe	ja	nein	UIS / UIC

Tabelle 3.2.: Charakteristika der Ansätze zur Integration von UI-Komponenten

Bei einer zusammenfassenden Betrachtung werden folgende Einschränkungen deutlich:

- Obwohl die betrachteten Ansätze eine Integration von UI-Komponenten in eine Zielanwendung ermöglichen, werden die Beschreibungen der UI-Komponenten nicht den Charakteristika einer interaktiven Komponente gerecht.
- Beide betrachteten Ansätze sind jeweils auf die Integration von UI-Komponenten in Web-Anwendungen beschränkt und bieten keine Lösung für eine Integration von UI-Komponenten in eine mobile Anwendung.
- Die UI-Komponenten beider betrachteten Ansätze zur Integration von UI-Komponenten in eine Zielanwendung, werden der Komplexität einer interaktiven Komponente, wie sie im Rahmen dieser Arbeit zugrundegelegt wird, nicht gerecht.
- Wie es den zuvor betrachteten Ansätzen zur automatischen Generierung von Benutzerschnittstellen an der Möglichkeit gefehlt hat, die generierten Anwendungen in einen gemeinsamen Anwendungskontext zu integrieren, fehlt es den Ansätzen zur Integration von UI-Komponenten in eine Zielanwendung an der Möglichkeit, die zu integrierenden UI-Komponenten automatisch zu generieren.

Zusammenfassend kann gesagt werden, dass keiner der betrachteten Ansätze ein vollautomatische Generierung von interaktiven Komponenten auf Basis von annotierten funktionalen Dienstbeschreibung unterstützen, welche dynamisch in eine Zielanwendung integriert werden können. Der Punkt der automatischen Generierung von interaktiven Komponenten wird in der Arbeit von [Fel11] adressiert, indem die Lösung darin besteht, die Anwendungsbeschreibung einer interaktiven dienstbasierten Anwendung ebenfalls mit Hilfe von Dienstannotationen auszudrücken, wobei eine werkzeuggestütze Unterstützung solch einer Anwendungserstellung in [Mar10] beschrieben wird. Diese Beschreibung kann im Rahmen

der Arbeit von [Fel11] dazu genutzt werden, eine Instanz eines Meta-Modells zur Beschreibung einer interaktiven dienstbasierten Anwendung zu generieren, die die notwendigen Informationen zur Integration einer Komponente in eine Zielanwendung enthält, welche auf Basis einer solchen Instanz generiert werden kann. Die Generierung einer solchen interaktiven Komponente und die Konzipierung der dazugehörigen Zielanwendung für Android werden im Laufe dieser Arbeit vorgestellt.

4. Konzeption und Umsetzung

In diesem Kapitel wird zunächst das Konzept des Gesamtsystems zur Generierung, der Veröffentlichung und der Integration der generierten Komponenten in die Zielanwendung vorgestellt, ausgehend von den ermittelten zentralen Anforderungen aus Kapitel 2.2, wobei die Dreiteilung *Generierungsverfahren*, *Distribution* und *Zielanwendung* beibehalten wird. In einem zweiten Teil wird die Umsetzung des aufgestellten Konzeptes diskutiert.

4.1. Konzeption

Das gesamte System zur Generierung von interaktiven Komponenten lässt sich in drei Software-Bestandteile untergliedern. Zum ersten ist das der *Android-Adapter*, welche als Server-Anwendung konzipiert wurde und die Generierung der interaktiven Komponenten, welche wiederum den zweiten Bestandteil darstellen, realisiert und diese anschließend in einem zentralen Repository veröffentlicht. Die *Zielanwendung*, welche den dritten Software-Bestandteil darstellt, integriert die generierten Komponenten und stellt diese dem Endnutzer zur Verfügung. Abbildung 4.1 bietet einen Überblick über das gesamte System.

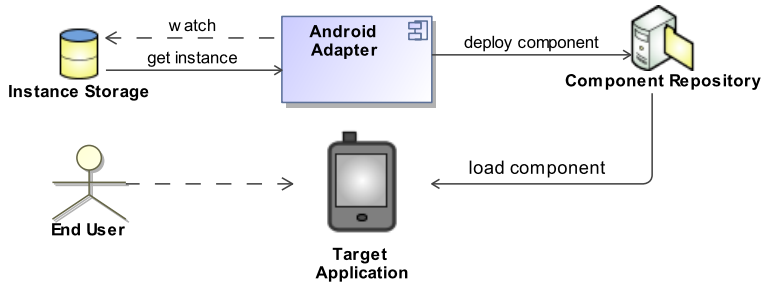

Abbildung 4.1.: System-Übersicht

In den folgenden Abschnitten werden die Konzepte des Generierungsverfahrens (4.1.1), das Konzept zur Distribution interaktiver Komponenten (4.1.2) und die Konzeption der Zielanwendung (4.1.3) vorgestellt.

4.1.1. Konzeption der Generierung interaktiver Komponenten

Das Konzept zur Generierung interaktiver Komponenten lässt sich in drei Teilbereiche gliedern:

1. Die *Server-Komponente*, welche ein Verzeichnis überwacht und die Generierung anstößt, sobald in diesem Verzeichnis eine neue Modellinstanz vorliegt.

2. Die M2C-Transformation, welche die eigentliche Generierung der interaktiven Komponenten realisiert.

3. Die interaktiven Komponenten als Resultat der M2C-Transformation.

Die Konzeption der *Server-Komponente* wird in Unterabschnitt 4.1.1.1 näher beschrieben. In Unterabschnitt 4.1.1.2 wird die Konzeption der M2C-Transformation vorgestellt. Abschließend werden einige der entwickelten Konzepte der interaktiven Komponente werden in Unterabschnitt 4.1.1.3 näher behandelt.

4.1.1.1. Server-Komponente

Wie bereits beschrieben, steuert die *Server-Komponente* die automatische Generierung interaktiver Komponenten, die anschließende Kompilierung, das Paketieren und die abschließende Veröffentlichung der generierten Komponenten. Die *Server-Komponente* lässt sich wiederum in drei Teilkomponenten unterteilen, dem *Generator*, dem *Directory-Watcher* und der *Workflow-Engine*. Die *Generator-Komponente* bildet hierbei die eigentliche Anwendung, welches mittels der *Directory-Scanner-Komponente* ein Verzeichnis überwacht und die Generierung und Veröffentlichung anstößt, sobald eine neue Modellinstanz im überwachten Verzeichnis vorliegt. Die als *Workflow-Engine* bezeichnete Teilkomponente ist für die Generierung, die Kompilierung, die Paketierung und die Veröffentlichung der interaktiven Komponenten zuständig. Jede dieser Teilaufgaben wird von der *Generator-Komponente* beim Vorliegen einer neuen Modellinstanz initiiert. Abbildung 4.2 verschafft einen Überblick über die *Server-Komponente* und das Zusammenspiel der einzelnen Teilkomponenten.

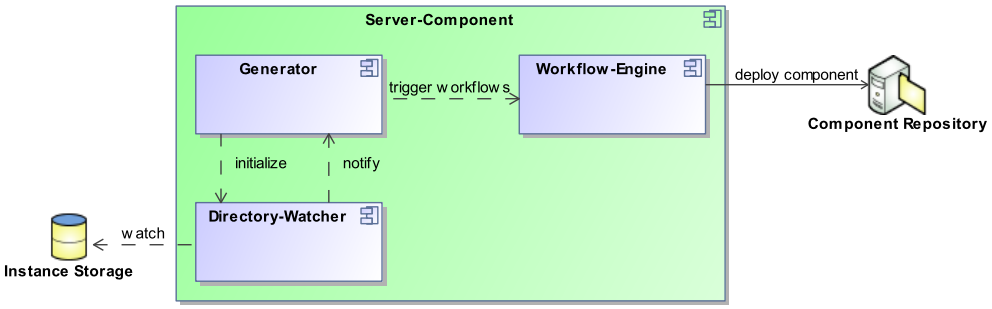

Abbildung 4.2.: Architektur der Server-Komponente

4.1.1.2. Modell-zu-Code-Transformation

Im Folgenden wird die Konzeption der M2C-Transformation zur Generierung interaktiver Komponenten vorgestellt, wobei eine implementierungsnahe Darstellung gewählt wurde,

um einen Mehrwert für den Abschnitt der Umsetzung der M2C-Transformation zu schaffen, durch Vermeidung von Dopplungen bezüglich der Diskussionen der betreffenden Abschnitte. Im Vorfeld der Konzeption der M2C-Transformation wurde zunächst die Menge fester unveränderlichen Bestandteile einer interaktiven Komponente ermittelt. Diese Bestandteile resultieren in einer Bibliothek, welche für jede generierte interaktive Komponente wiederverwendet wird. Durch diese Vorüberlegung wird die Komplexität der M2C-Transformation und die Redundanz des generieren Codes verringert. In der Mengendarstellung der Abbildung 4.3 werden die *Common Library* und beispielhaft zwei generierte interaktive Komponenten veranschaulicht.

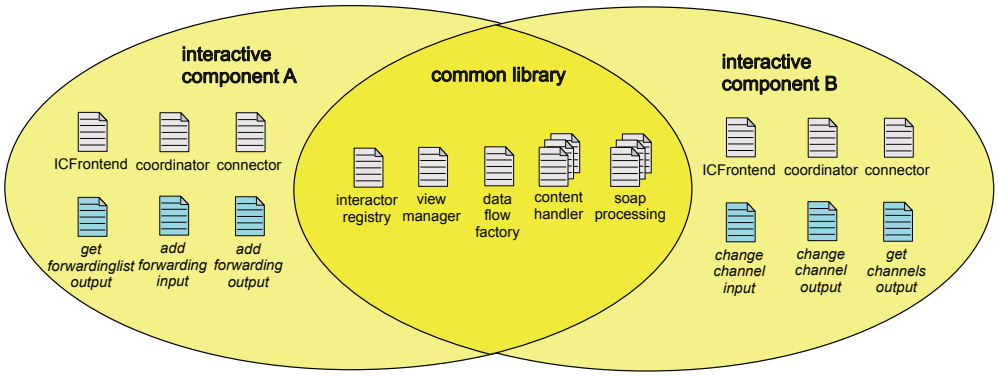

Abbildung 4.3.: Gemeinsam genutzter Code generierter Komponenten

Generierte Komponenten variieren in den einzelnen *Views*, die sie enthalten, und in den generierten Service-Implementierungen der Teilkomponenten einer interaktiven Komponente. Das Konzept des zugrundeliegenden Komponentenmodells wird in Unterabschnitt 4.1.1.3 näher erklärt. Ebenso werden in diesem Unterabschnitt die Bestandteile der *Common Library* näher erläutert. Die Struktur der eigentlichen M2C-Transformation resultiert aus der Struktur des zugrundeliegenden Meta-Modells, welches als Ausgangspunkt der Generierung verwendet wird. Dieses Meta-Modell wurde bereits im Grundlagenkapitel (3.1.3) vorgestellt. Als Einstiegspunkt der M2C-Transformation dient die *InteractiveComponent*. Die *componentDependencies*, die *globalDataEntities* und die *contextElements* dienen in Verbindung mit der *uuid* und des Names der *InteractiveComponent* zur Generierung der Daten zur Registrierung der interaktiven Komponente in der Zielanwendung. Das Konzept zur Registrierung wird im Unterabschnitt 4.1.3.2 detailliert beschrieben. Ausgehend von der *InteractiveComponent* und den ermittelten Registrierungsdaten wird die Schnittstelle zur Interaktion mit der Zielanwendung generiert. Daraufhin werden die einzelnen *views* der *InteractiveComponent* durchlaufen, um deren Code-Repräsentation zu erzeugen. Das Annotated Serive Model (ASM) dient der Generierung der Klassen, welche zum Service-Aufruf benötigt werden.

Im Folgenden soll das Konzept der M2C-Transformation anhand des Algorithmus 4.1 veranschaulicht werden. Die Generierung beginnt mit dem Einstiegspunkt CreateInteractiveComponent, wobei zunächst die *Connector-Komponente*, danach die *Coordinator-Komponente* und zum Schluss das *ICFrontend (Interactive Component Frontend)* generiert

Algorithmus 4.1 M2C-Transformation

```
1: procedure CreateInteractiveComponent(IC)
2:     call CreateConnector(IC_asm)
3:     call CreateCoordinator(IC_asm)
4:     call CreateICFrontend(IC)
5: end procedure
6:
7: procedure CreateICFrontend(IC)
8:     writeRegistrationData(IC)
9:     for each view in IC_views do
10:        call CreateView(IC)
11:    end for
12: end procedure
13:
14: procedure CreateView(view)
15:    for each c in GetContents(view) do
16:        writeContentFactoryMethod(c)
17:        if typeof(c) = Activator then
18:            writeActivatorListener(c)
19:        end if
20:    end for
21: end procedure
22:
23: function GetContents(c)
24:    childs ← ∅
25:    if typeof(c) = View then
26:        return c_rootContainer ∪ GetContents(c_rootContainer)
27:    else if typeof(c) = ContentContainer then
28:        for each nestedContent in c_content do
29:            childs ← childs ∪ GetContents(nestedContent)
30:        end for
31:        return c_content ∪ childs
32:    else if typeof(c) = SwitchContentContainer then
33:        for each nestedContent in c_contentContainers do
34:            childs ← childs ∪ GetContents(nestedContent)
35:        end for
36:        return c_content ∪ childs
37:    else if typeof(c) = Wizard then
38:        for each nestedContent in c_contentContainers do
39:            childs ← childs ∪ GetContents(nestedContent)
40:        end for
41:        return c_content ∪ childs
42:    else
43:        return c
44:    end if
45: end function
```

wird, wobei die konkrete Reihenfolge keine Rolle spielt. Das Komponentenmodell der interaktiven Komponente wird in Abschnitt 4.1.1.3 detailliert beschrieben. Während der Generierung des *ICFrontends*, werden zunächst die Registrierungsdaten geschrieben. Daraufhin wird über alle assoziierten *Views* der *InteractiveComponent* iteriert und für jeden *View* eine entsprechende Code-Repräsentation mittels CreateView erstellt. Bei dieser Erstellung werden zunächst die Menge aller *Content*-Elemente eines *Views* rekursiv ermittelt und anschließend über diese Menge iteriert. Dabei wird für jeden *Content* eine Factory-Methode generiert, welche die entsprechende Code-Repräsentation erzeugt. Neben der eigentlichen Code-Repräsentation des *Contents* werden zugleich *Content-Handler* erstellt, deren Funktionsweise im Abschnitt zur Konzeption der interaktiven Komponente (4.1.1.3) präsentiert wird. Zum Bestimmen der genauen Repräsentation dient der Typ eines *Contents* und dessen relevanten Annotationen.

Für jeden *Activator* wird zusätzlich ein Listener zur Ereignisbehandlung generiert. Hierbei wird der dazugehörige Navigations- und der Datenfluss, falls vorhanden, in eine entsprechende Code-Repräsentation umgesetzt.

Ein weiteres Konzept der M2C-Transformation resultiert aus der Anforderung /AG02/, der Erweiterbarkeit der M2C-Transformation, welche auf Seite 9 beschrieben wurde. Die Transformation soll hierbei zu einem späteren Zeitpunkt um Teile erweitert werden, welche spezielle UI-Elemente in Form von Widgets generieren. Zum einen muss hierbei für eine einheitliche Schnittstelle der Transformationserweiterungen gesorgt werden und zum anderen für eine einheitliche Schnittstelle der generierten Widgets, damit diese in den Code, dessen M2C-Transformation bekannt ist, integriert werden können.

4.1.1.3. Konzeption der Interaktiven Komponente

Komponentenmodell Die generierten interaktiven Komponenten gliedern sich jeweils in vier Teilkomponenten, welche in Abbildung 4.4 veranschaulicht werden. Zum ersten ist das das *InteractiveComponentFrontend*, welches das UI enthält und die Schnittstelle zur Interaktion mit der Zielanwendung bereitstellt. Kommt es dann zu einem Service-Aufruf, wird dieser an den *Coordinator* weitergeleitet. Dieser kann dann den Service-Aufruf um zusätzliche Parameter ergänzen, welche nicht vom Nutzer eingegeben werden, wie zum Beispiel die aktuelle Position oder einen von der Zielanwendung bereitgestellten Session-Key zur Authentifizierung. Der ergänzte Service-Aufruf wird dann an den *Connector* weitergereicht, welcher diesen wiederum an den *SoapService* weiter gibt, der dann den eigentlichen Service-Aufruf tätigt und das Ergebnis zurückliefert. Der *Connector* übermittelt dieses Ergebnis daraufhin dem *Coordinator*, der diesen wiederum an die *InteractiveComponent-Frontend* weiter reicht.

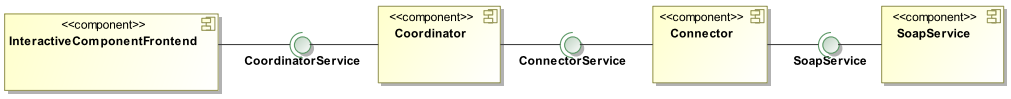

Abbildung 4.4.: Interactive Component - Komponenten-Diagramm

Der *Connector* fungiert somit als Proxy, welcher die Service-Aufrufe annimmt und an den *SoapService* weiterreicht. Mit dieser Architektur könnte auch eine persistente Speicherung von Daten, bei nicht vorhandenen Internetverbindung, realisiert werden, indem der *Connector* bei nicht vorhandener Internetverbindung die Operationsaufrufe, entgegen nimmt und in einer lokalen Datenbank zwischenspeichert. Zu einem späteren Zeitpunkt, bei bestehender Internetverbindung, würden dann die CRUD-Operationen, an den *SoapService* übergeben werden, um die lokale Datenbank mit der eigentlichen entfernten Datenbank zu synchronisieren.

Datenmodell Ein weiterer wichtiger Punkt in der Konzeption der interaktiven Komponenten ist das zugrundeliegende Datenmodell, wobei eine generische Lösung favorisiert wurde. Dies bedeutet, dass der *SoapService* ungetypte Objekte zurückliefert, welche dann mittels eines *ObjectResolvers* abgefragt werden können. Umgekehrt können mit Hilfe eines *ObjectCreators* Objekte für einen SOAP-Aufruf erzeugt werden. Zu jeder konkreten *SoapService*-Implementierung muss somit der dazugehörige *ObjectResolver* bzw. der dazugehörige *ObjectCreator* implementiert werden. Als Abfragesprache wurde eine Sprache namens *SelectionPath* spezifiziert, welche es ermöglicht Kind-Elemente in beliebiger Baumtiefe abzufragen. Erzeugt wird eine Instanz eines *SelectionPaths* von der *SelectionPathFactory*. Neben dem Erzeugen und Abfragen von SOAP-Objekten, wird eine *SelectionPath*-Instanz in Verbindung mit einem eindeutigen Bezeichner einer Service-Operation zur Registrierung von Ein- bzw. Ausgabeelementen verwendet, um diese an das jeweilige Service-Element zu binden. Die Registrierung wird hierbei von der *Interactor Registry*, welche noch näher erläutert wird, vorgenommen. Die Syntax der *SelectionPath*-Abfragesprace wird in Abbildung 4.5 verdeutlicht. In Listing 4.1 werden beispielhaft drei Instanzen der *SelectionPath*-Abfragesprace vorgestellt.

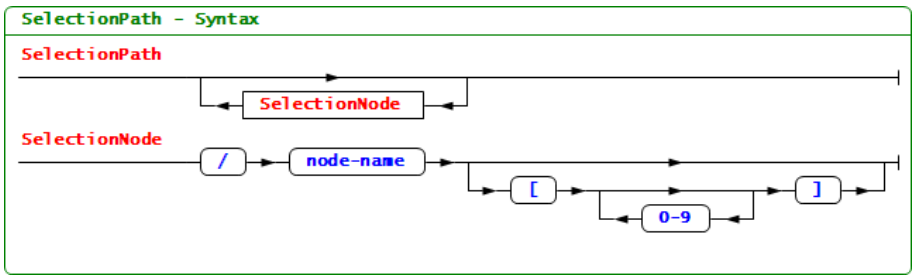

Abbildung 4.5.: SelectionPath - Syntax-Diagramm

```
1 /Customer
2 /Customer/Orders[]
3 /Customer/Orders[5]/Price
```

Listing 4.1: SelectionPath - Beispiele

Die erste Instanz zeigt lediglich auf das Kopfelement *Customer*, die zweite Instanz zeigt auf alle Bestellungen des Kunden und die dritte Instanz zeigt auf den Preis der fünften Bestellung des Kunden. Die Verwendung des generischen Datenmodells bringt einige Vorteile

mit sich. Zum einen entfällt die Generierung der Daten-Objekte auf Basis des Service-Modells, was einen wesentlichen Mehraufwand bedeuten würde. Da die Verwendung der Daten-Objekte im Vorhinein bekannt ist, genügt die Generierung der Abfrage-Pfade an den jeweiligen Stellen im Programm, welche sich relativ leicht aus dem Service-Modell erstellen lassen. Der wesentliche Nachteil dieser Lösung besteht bezüglich der Typsicherheit. Diese wird aber vom Generierungsverfahren gewährleistet, da zum Zeitpunkt der Generierung das Service-Modell, welches die Typinformation enthält, noch bekannt ist.

Content-Handler Für jedes UI-Element, das während der M2C-Transformation erzeugt wird, werden *Content-Handler* erzeugt, welche als *Adapter* zum jeweiligen nativen UI-Element verstanden werden können. Jeder *Content-Handler* konkretisiert hierbei, je nach Typ des nativen UI-Elements, eine abstrakte Klasse. Diese abstrakte Klasse stellt somit für alle nativen UI-Elemente eine einheitliche Schnittstelle zur Verfügung. Bestandteil dieser Schnittstelle sind Methoden zum Ermitteln oder Setzen eines Wertes plus zusätzliche Methoden, die die Handhabung verschiedener nativer UI-Elemente vereinheitlichen. Die *Content-Handler* können somit als zusätzliche Abstraktionsschicht über den nativen UI-Elementen verstanden werden, was vor allem die Komplexität der M2C-Transformation verringert. Ein weiterer Vorteil ist, dass geforderte Funktionalitäten, resultierend aus Annotationen, direkt in dieser Schicht implementiert werden können, ohne die nativen UI-Elemente anpassen zu müssen oder die M2C-Transformation unnötig zu erweitern. Um zum Beispiel die Funktionalität resultierend aus einer Annotation, welche für alle Eingabefelder gültig ist, wie das *MandatoryField* (vgl. [Ser09], Seite 37) zu implementieren, muss nur die abstrakte *Content-Handler*-Klasse erweitert werden.

Interactor-Registry Eine *Interactor-Registry* dient der Registrierung von UI-Elementen eines *Views*, welche der Ein- bzw. Ausgabe einer Service-Operation dienen. Für die Registrierung wird ein eindeutiger Bezeichner der Service-Operation, der *SelectionPath* des dazugehörigen Parameterelements und der jeweilige *Content-Handler* verwendet. Die Registrierung geschieht hierbei während der Initialisierung des jeweiligen *Views*. Kommt es zum Auslösen eines Operationsaufrufes, können die Werte der Eingabeparameter für den entsprechenden Operationsaufruf relativ leicht ermittelt werden. Auf der andern Seite können die UI-Elemente, welche der Ausgabe einer Operation dienen, leicht aktualisiert werden.

4.1.2. Konzeption der Distribution interaktiver Komponenten

Um der Zielanwendung das Laden und Installieren der generierten Komponenten zu ermöglichen, werden diese in einem zentralen Repository veröffentlicht. Die aufgestellten zentralen Anforderungen, die sich an die Distribution der Komponenten richten, sind zum einem Vertrauenswürdigkeit und zum anderen die Versionierung der veröffentlichten Komponenten. Vertrauenswürdigkeit hinsichtlich der Klienten, welche aus der *Server-Komponente*

beim Veröffentlichen und aus der Zielanwendung beim Laden der Komponenten bestehen, wird durch eine Authentifizierung durch Nutzername und Passwort erreicht. Vertrauenswürdigkeit hinsichtlich des Servers wird durch Nutzung des sicheren Hypertext-Übertragungsprotokolls erreicht, wobei nur Zertifikate von vertrauenswürdigen Quellen akzeptiert werden. Die Versionierung der veröffentlichten Komponenten wird mit Hilfe von Revisionsnummern des Repositories realisiert, wobei jede Revisionsnummer eindeutig einer Version einer Komponente zugeordnet werden kann. Die Zielanwendung kann die Revisionsnummern der veröffentlichten Komponenten ermitteln und mit denen der bereits installierten Komponenten vergleichen, da diese beim Herunterladen in einer lokalen Kopie des Repositories gespeichert bzw. aktualisiert werden. Abbildung 4.6 verdeutlicht den Ablauf beim Herunterladen der Komponenten nach einer erfolgreichen Authentifizierung durch die Zielanwendung.

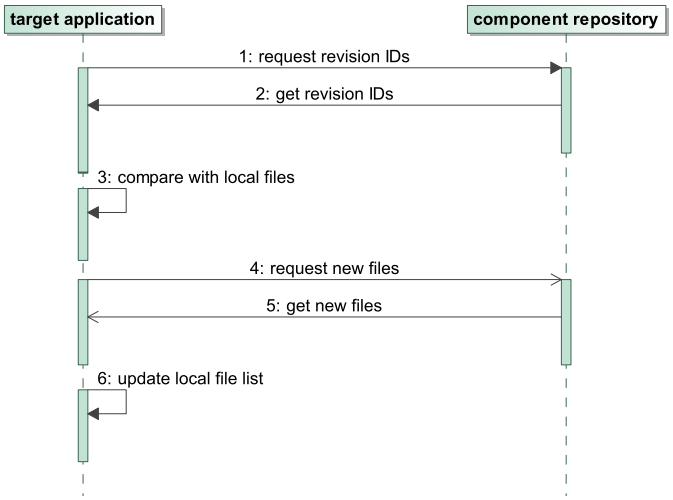

Abbildung 4.6.: Component Repository - Interaktion mit Zielanwendung

4.1.3. Konzeption der Zielanwendung

Die Zielanwendung bildet die Rahmenanwendung, für die eine transparente Integration der generierten Komponenten realisiert wird. Transparente Integration bedeutet, dass sich die integrierten interaktiven Komponenten nahtlos in die Rahmenanwendung einfügen sollen, ohne einen zu großen Bruch des User-Interfaces zu verursachen und dass der Vorgang vollkommen automatisch abläuft. Neben der Sicht zur Integration der Komponenten stellt die Zielanwendung zusätzlich verschiedene Konfigurationssichten bereit. Ein Konzept der Benutzerschnittstelle einer solchen Anwendung wird in Abbildung 4.7 veranschaulicht.

Als Beispiel für das UI-Konzept wurde eine Anwendung für das in Kapitel 2.1 vorgestellte Szenario zur Heimautomatisierung gewählt. Sicht (a) repräsentiert eine vordefinierte Authentifizierung, welche der Nutzer zu Beginn der Anwendung durchführen muss. Nach erfolgreicher Authentifizierung gelangt der Nutzer direkt zur Hauptsicht der Anwendung, Sicht (b), welche die integrierten interaktiven Komponenten visualisiert. Dargestellt sind hier drei interaktive Komponenten und deren Einstiegspunkte. Sicht (c) und Sicht (d)

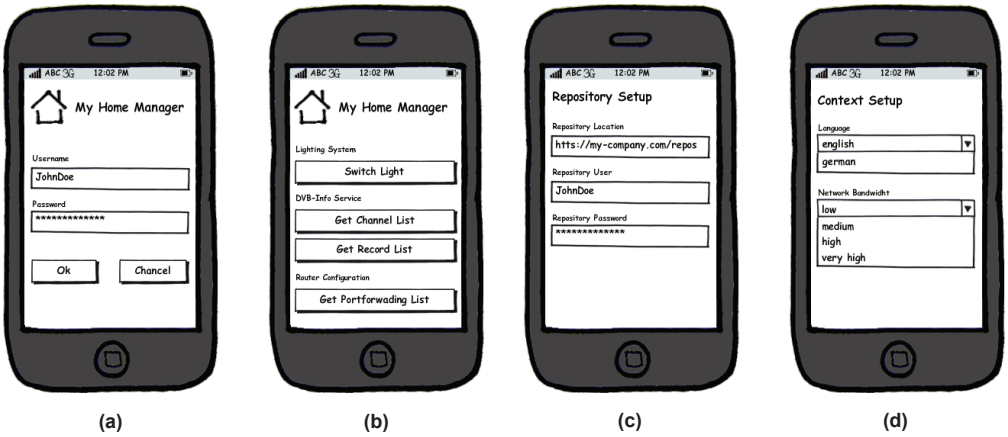

Abbildung 4.7.: Konzept der Benutzerschnittstelle der Zielanwendung

können über das Menü erreicht werden und dienen der Konfiguration der Zugangsparameter des Component-Repositories und der Konfiguration des momentanen Kontextes der Zielanwendung.

4.1.3.1. Architektur der Zielanwendung

Die Architektur der Zielanwendung wurde so konzipiert, das eine konkrete Zielanwendung, wie zum Beispiel die Anwendung für das Szenario zur Heimautomatisierung, welche auf ein bestimmtes Anwendungsszenario zugeschnitten ist, nur wenige Klassen der Kern-Bibliothek instanziieren bzw. konkretisieren muss. Die Kern-Bibliothek besteht wiederum aus einzelnen Komponenten, welche in Abbildung 4.8 veranschaulicht werden.

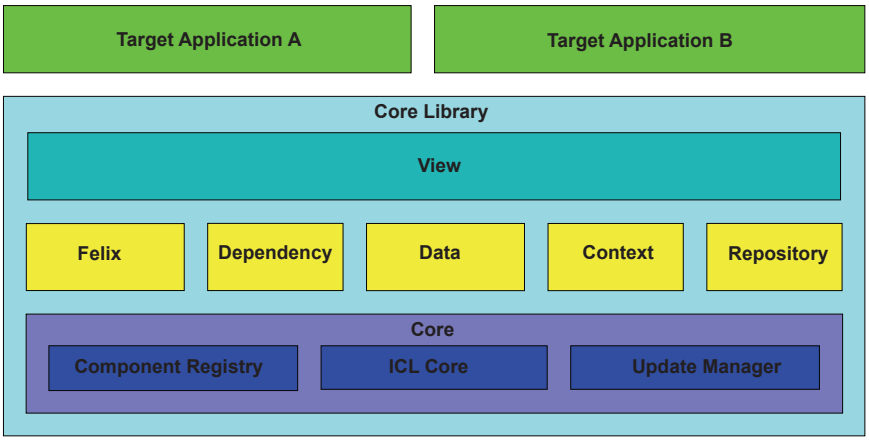

Abbildung 4.8.: Architektur einer Zielanwendung

Die Core-Komponente beinhaltet die Klassen, welche für die Basisfunktionalitäten der Zielanwendung zuständig sind. Das Herzstück bildet hierbei der *ICLCore*, welcher die Verwaltung, Steuerung und die Kommunikation der einzelnen Teilkomponenten untereinander regelt. Ein weiterer wichtiger Bestandteil der *Core-Komponente* ist die *Component Registry*, welche die einzelnen geladenen interaktiven Komponenten in der Zielanwendung

registriert. Der Prozess der Registrierung wird in Unterabschnitt 4.1.3.2 näher beschrieben. Der *Update Manager*, als dritter Bestandteil der *Core-Komponente*, verwaltet die geladenen Komponenten und übernimmt deren Versionierung. Das Konzept zur Versionierung wurde bereits in Abschnitt 4.1.2 vorgestellt.

Die weiteren Komponenten der Kern-Bibliothek separieren jeweils eine Funktionalität der Zielanwendung.

Die Felix-Komponente regelt das Laden der OSGi-Bundles und stellt Klassen und Schnittstellen zur Ereignisbehandlung bereit, welche es z.B. anderen Komponenten ermöglicht, auf neu installierte Bundles zu reagieren.

Die Dependency-Komponente dient zur Verwaltung von Plattformfunktionalitäten, welche nachträglich installiert werden können. Während des Registrierungsprozesses kann es vorkommen, dass eine interaktive Komponente der Zielanwendung mitteilt, dass diese eine zusätzliche Plattformfunktionalität benötigt. Die Mitteilung erfolgt in Form eines voll qualifizierten Namens. Eine solche Funktionalität könnte zum Beispiel aus dem Bereitstellen der aktuellen geografischen Position in Form von GPS-Koordinaten[4] bestehen. Falls diese Funktionalität in der Zielanwendung noch nicht verfügbar ist, wird diese in Form von OSGi-Bundles nachgeladen, installiert und der interaktiven Komponenten bekannt gemacht. Den Ort von dem aus die Bundles installiert werden können, werden der Zielanwendung ebenfalls während der Registrierung von der interaktiven Komponenten mitgeteilt. Die *Dependency-Komponente* übernimmt hierbei die Verwaltung solcher zusätzlicher Funktionalitäten bzw. initiiert deren Nachladen bei Nichtvorhandensein und übernimmt die Benachrichtigung bei erfolgreicher Installation.

Die Data-Komponente übernimmt die Verwaltung von einzelnen Data-Entities, welche global in der Zielanwendung verfügbar sind. Ein solches Data-Entity kann von einer interaktiven Komponenten zum Start als Eingabeparameter benötigt werden. Umgekehrt können interaktive Komponenten Data-Entities der Zielanwendung bekannt machen, die so wiederum anderen Komponenten als Eingabeparameter dienen können. Neben interaktiven Komponenten können aber auch Klassen der Zielanwendung Data-Entities bereitstellen. So kann zum Beispiel eine vordefinierte Authentifizierung als fester Bestandteil der Zielanwendung implementiert werden, welche bei erfolgreicher Authentifizierung, den zurückgelieferten Session-Key als Data-Entity bekannt macht. Als eindeutiger Bezeichner für die einzelnen Data-Entities wird eine ID bestehend aus dem URI der Service-Beschreibung, dem Servicenamen, dem Operationsnamen und dem Namen des Rückgabeparameters der aufgerufenen Operation verwendet. Die *Data-Komponente* regelt hierbei die Speicherung der einzelnen Data-Entities und stellt Klassen zur Bereitstellung von Data-Entities seitens der Zielanwendung zur Verfügung.

Die Context-Komponente dient zur Verwaltung der in der Zielanwendung verfügbaren Kontexte. Ein Kontext kann zum Beispiel die zugrundeliegende Plattform sein, wobei der

[4] Global Positioning System, offiziell NAVSTAR GPS, ist ein globales Navigationssatellitensystem zur Positionsbestimmung und Zeitmessung.

Wert dieses Kontexts in diesem Fall immer *Android* lauten würde. Ein anderes Beispiel wäre die in der Zielanwendung momentan eingestellte Sprache. Eine registrierte interaktive Komponente wird erst dann aktiviert und in der Zielanwendung visualisiert, wenn alle Kontexte mit den geforderten Werten verfügbar sind und alle anderen Abhängigkeiten aufgelöst sind. Ändert sich der Wert eines Kontexts, welcher von einer interaktiven Komponente gefordert wurde, wird diese wieder deaktiviert und in der Zielanwendung nicht mehr visualisiert. So kann zum Beispiel Mehrsprachigkeit erreicht werden, falls eine interaktive Komponente für verschiedene Sprachen generiert und installiert wurde. Die *Context-Komponente* übernimmt hierbei die Verwaltung der jeweiligen Kontexte und deren aktuellen Zustände und stellt zusätzlich Schnittstellen zur Spezifizierung spezieller Kontexte in Form von Kontext-Providern bereit. Ein solcher Kontext-Provider kann von einer Zielanwendung implementiert werden, um diese an das jeweilige Anwendungsumfeld anzupassen. Zum Beispiel könnte ein ortsbezogener Kontext-Provider implementiert werden, der je nach konfigurierbarem Ort, einen anderen Wert annimmt, basierend auf der momentan geografischen Position des Endgerätes.

Die Repository-Komponente dient zur lokalen Speicherung und Versionierung der interaktiven Komponenten. Verwaltet und angesteuert wird diese Komponente vom bereits erwähnten *Update Manager* der *Core-Komponente*.

Die View-Komponente, welche in Abbildung 4.8 veranschaulicht wurde, ist nicht als Software-Komponente zu verstehen, sondern sollte eher als Präsentationsschicht der Kern-Bibliothek betrachtet werden. Diese Schicht enthält somit alle Klassen, welche der Visualisierung dienen, wodurch die Trennung der Benutzerschnittstelle von der Kontroll-Logik bzw. vom verwendeten Datenmodell erreicht wurde. Enthalten sind die Klassen, welche das UI zur Visualisierung integrierter Komponenten bereitstellen, die Klassen zur Verwaltung der vom Nutzer konfigurierbaren Kontext-Provider, die Klassen zur Konfiguration des Komponenten-Repositories und die Klassen, welche benötigt werden, um eine wie bereits erwähnte vordefinierte Authentifizierung zu realisieren. Im Wesentlichen konkretisiert bzw. parametrisiert eine konkrete Zielanwendung Klassen aus dieser Schicht.

Eine konkrete Zielanwendung welche in Abbildung 4.8 als *Target Application A* bzw. als *Target Application B* veranschaulicht ist, muss wie bereits erwähnt nur Klassen der Kern-Bibliothek instanziieren, um eine vollständige Anwendung zu realisieren, welche in der Lage ist, interaktive Komponenten transparent integrieren zu können. Kontextadaptierbarkeit, als zentrale Anforderung der Zielanwendung, kann durch Bereitstellung spezieller Kontext-Provider bzw. spezieller Provider für Data-Entities erreicht werden. Somit kann mit relativ geringem Aufwand die Zielanwendung an ein spezielles Anwendungsumfeld angepasst werden kann.

4.1.3.2. Registrierung interaktiver Komponenten

Wie bereits schon erwähnt, werden die interaktiven Komponenten nach dem Laden in der Zielanwendung registriert. Die Registrierung wird hierbei von der *Component Registry* der

Core-Komponente übernommen. Die für den Registrierungsprozess benötigten Informationen müssen der Zielanwendung von der interaktiven Komponente bereitgestellt werden. Die einzelnen Registrierungsinformation werden von [Fel11] in Kapitel 4 wie folgt beschrieben.

1. einen *eindeutigen Identifikator* der interaktiven Komponente;
2. den *Zielkontext* für den die interaktive Komponente generiert wurde;
   Bedingung für Angabe: interaktive Komponente ist kontextabhängig
3. eine Menge von Tupeln bestehend aus *vollqualifizierten Operationsnamen*, die zur direkten Operationsansteuerung verwendet werden können und einer Menge mit den Operationsnamen assoziierter *Zusatzinformationen*;
   Bedingung für Angabe: interaktive Komponente unterstützt keine indexierte Navigation, sondern eine direkte Operationsansteuerung
4. Identifikatoren für die Elemente der Menge von der interaktiven Komponente *benötigter Eingabeparameter*;
   Bedingung für Angabe: interaktive Komponente benötigt von der Zielanwendung Eingabeparameter
5. Identifikatoren für die Elemente der Menge von der interaktiven Komponente der Zielanwendung *bereitgestellter Ausgabeparameter*;
   Bedingung für Angabe: interaktive Komponente liefert Ausgabeparameter zurück
6. eine Menge über eindeutige Bezeichner identifizierbarer Funktionalitäten, die die Zielanwendung bereitstellen muss;;
   Bedingung für Angabe: interaktive Komponente besitzt Abhängigkeiten zu lokalen Funktionalitäten der Zielanwendung

Minimal ist für eine erfolgreiche Registrierung nur der unter Punkt 1 angegebene eindeutige Identifikator notwendig. Wird ein Zielkontext, bestehend aus einzelnen Kontexten und deren jeweiligen Werten angegeben, ist die interaktive Komponente kontextabhängig. Das bedeutet, die Komponente wurde für einen speziellen Zielkontext generiert, welcher der Zielanwendung während der Registrierung übermittelt wird. Die Komponente wird in diesem Fall erst dann aktiviert, sobald der Kontext mit dem geforderter Wert in der Zielanwendung verfügbar ist, wie bereits in der Beschreibung der *Context-Komponente* erwähnt wurde. Wird eine wie unter Punkt 3 aufgelistete Menge von Tupeln bestehend aus vollqualifizierten Operationsnamen und eine Menge mit dem Operationsnamen assoziierter Zusatzinformationen angegeben, unterstützt die interaktive Komponente eine direkte Operationsansteuerung. Anderenfalls muss diese eine indexierte Navigation unterstützen, wobei in der Zielanwendung lediglich nur ein einziger Einstiegspunkt visualisiert wird. Die interaktive Komponente muss dann die Sicht zur Ansteuerung der einzelnen Operationen selbst bereitstellen. Die assoziierten Zusatzinformationen dienen der Zielanwendung zur Aufbereitung der Darstellung ansteuerbarer Operationen für den Nutzer. Diese bestehen in der Regel aus menschenlesbaren Bezeichnern oder Piktogrammen, die von der

Zielanwendung visualisiert werden können. Wird der Zielanwendung eine unter Punkt 4 angegebene Menge von Eingabeparametern übermittelt, kann diese nur aktiviert und visualisiert werden, sobald diese Eingabeparameter als Data-Entity in der Zielanwendung verfügbar sind. Die Verwaltung solcher Entities wurde bereits in der Beschreibung der *Data-Komponente* erwähnt. Analog dazu kann eine interaktive Komponente Parameter in Form von Data-Entities der Zielanwendung bekannt machen, wenn deren eindeutigen Bezeichner in der unter Punkt 5 angegebene Menge enthalten sind. Wird eine unter Punkt 6 angebende Menge von zusätzlichen Plattformfunktionalitäten der Zielanwendung während der Registrierung übergeben, kann die interaktive Komponente nur dann gestartet werden, sofern die geforderten Funktionalitäten verfügbar sind bzw. nachinstalliert werden konnten. In der Beschreibung der *Dependency-Komponente* wurden solche Plattformfunktionalitäten bereits näher beschrieben.

Für die Steuerung der Registrierung und Aktivierung der generierten Komponenten wurde ein Registrierungsmechanismus konzipiert. Die Komponenten können hierbei vier verschiedene Zustände annehmen: *loaded*, *inactive*, *active started* und *paused*.

- Den Zustand *loaded* nimmt eine interaktive Komponente automatisch an, sobald diese physisch verfügbar ist und erfolgreich in der Service-Plattform registriert und gestartet wurde.
- Im Zustand *inactive* verweilt die Komponente, falls geforderte Bedingungen, bestehend aus einem Zielkontext und eventuell benötigter Eingabeparameter, nicht erfüllt sind.
- Den Zustand *activateable* nimmt die Komponente an, sobald alle geforderten Bedingungen erfüllt sind. Die Komponente wird in diesem Zustand in der Zielanwendung visualisiert und ist startbar.
- Den Zustand *activated* nimmt eine interaktive Komponente an, sobald diese gestartet wurde. Sie verweilt in diesem Zustand bis zu ihrer Beendigung. Zur selben Zeit kann nur eine einzelne Komponente diesen Zustand annehmen, da aufgrund der Eigenschaften einer mobilen Anwendung die Aktivierung von mehreren Komponenten keinen Sinn ergeben würde, prinzipiell aber möglich wäre.
- Den Zustand *paused* kann eine interaktive Komponente nur annehmen, sofern diese zuvor gestartet wurde. Die Komponente ist in diesem Zustand pausiert bis eine Fortsetzung seitens der Zielanwendung erfolgt und die Komponente an der Stelle fortgesetzt wird, an der sie pausiert wurde.

Wie bereits beschrieben, nehmen die interaktiven Komponenten den Zustand *loaded* an, sobald diese physisch verfügbar sind. Können geforderte Plattformfunktionalitäten nicht nachinstalliert werden, geht die jeweilige Komponente in den Endzustand über und wird physisch gelöscht. Anderenfalls nehmen die jeweiligen Komponenten zunächst den Zustand *inactive* an, in dem sie verweilen, bis die geforderten Bedingungen aufgelöst werden

können. Sind alle Anforderungen erfüllt, nimmt die jeweilige Komponente den Zustand *activateable* an und wird in der Zielanwendung visualisiert. Wird die Komponente daraufhin gestartet, nimmt die Komponente den Zustand *activated* an, bis diese wieder gestoppt wird und die Komponente wieder in den Zustand *activateable* übergeht oder die Komponente pausiert wird, wobei diese den Zustand *paused* annimmt. Vom Zustand *activated* aus kann die Komponente auch wieder den Zustand *inactive* annehmen, wenn z.B. ein geforderter Kontext nicht mehr verfügbar ist. Die Komponente wird dann nicht mehr visualisiert und kann auch nicht mehr gestartet werden. Abbildung 4.9 verdeutlicht noch einmal die Übergänge der einzelnen Zustände, welche eine Komponente annehmen kann.

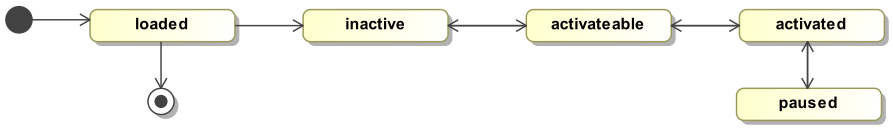

Abbildung 4.9.: Komponenten-Zustände

4.1.3.3. Algorithmus zur Aktivierung interaktiver Komponenten

Um die Zustände bzw. die Zustandsübergänge geladener interaktiver Komponenten, welche in Abschnitt 4.1.3.2 beschrieben wurden, steuern zu können, wurde ein Algorithmus konzipiert, der im Folgenden näher diskutiert wird. Grundlage hierfür bilden die Registrierungsdaten der jeweiligen interaktiven Komponente, welche ebenfalls in Abschnitt 4.1.3.2 beschrieben wurden. Um eine Komponente aktivieren zu können, müssen alle Bedingungen bezüglich geforderter Plattformfunktionalitäten erfüllt sein, alle benötigten Eingabeparameter müssen verfügbar sein und der geforderte Zielkontext muss dem der Zielanwendung entsprechen. In Algorithmus 4.2 wird die Prüfung auf eventuelle Plattformfunktionalitäten veranschaulicht. Zu Beginn ist der aktuelle Zustand der Komponente $LOADED$.

Algorithmus 4.2 Prüfung auf Komponenten-Abhängigkeiten

Require: IC with $IC_{Reg} = (Dependencies, InputParameter, TargetContext)$
1: **function** PROOFCOMPONENTDEPENDENCIES(IC)
2: $dependenciesFulfilled \leftarrow true$
3: **for each** $icDependency$ **in** $Dependencies_{IC}$ **do**
4: **if not** $IsServiceAvailable(icDependency)$ **then**
5: $dependenciesFulfilled \leftarrow false$
6: $RequestComponentDependency(icDependency)$
7: **break**
8: **end if**
9: **end for**
10: **if** $dependenciesFulfilled$ **then**
11: $STATE_{IC} \leftarrow INACTIVE$
12: **call** PROOFREQUIREDINPUTPARAMETER(IC)
13: **end if**
14: **end function**

Im Wesentlichen überprüft dieser Algorithmus für jede geforderte Plattformfunktionalität, ob diese in der Zielanwendung verfügbar ist. Ist dies nicht der Fall, wird die geforderte Plattformfunktionalität mittels *RequestComponentDependency* angefordert. Die Prüfung bricht daraufhin ab. Sobald eine Plattformfunktionalität erfolgreich installiert wurde, wird der Algorithmus nochmals gestartet. Die Prüfung sollte dann erfolgreich bestanden werden, woraufhin die Prüfung auf benötigte Eingabeparameter gestartet wird, welche in Algorithmus 4.3 verdeutlicht wird. Nicht verdeutlicht ist der Fall des Fehlschlagens des Nachladens einer Plattformfunktionalität, wobei die jeweilige interaktive Komponente physisch gelöscht wird und somit den Endzustand annimmt.

Algorithmus 4.3 Prüfung auf benötigte Eingabeparameter

Require: IC with $IC_{Reg} = (Dependencies, InputParameter, TargetContext)$

```
 1: function PROOFREQUIREDINPUTPARAMETER(IC)
 2:     requiredParameterAvailable ← true
 3:     for each icInputParameter in InputParameter_IC do
 4:         if not IsDataEntityAvailable(icInputParameter) then
 5:             requiredParameterAvailable ← false
 6:             break
 7:         end if
 8:     end for
 9:     if requiredParameterAvailable then
10:         call PROOFCONTEXT(IC)
11:     else
12:         STATE_IC ← INACTIVE
13:     end if
14: end function
```

Der Algorithmus zur Prüfung auf benötigte Eingabeparameter verhält sich analog zum eben beschriebenen Algorithmus. Es wird für jeden geforderten Eingabeparameter geprüft, ob dieser in der Zielanwendung verfügbar ist. Sobald eine dieser Prüfungen fehlschlägt, wird die Komponente in den Zustand $INACTIVE$ versetzt. Werden hingegen alle Prüfungen bestanden, wird die Prüfung des geforderten Zielkontextes gestartet, welche in Algorithmus 4.4 veranschaulicht wird.

Bei der Prüfung auf den verfügbaren Zielkontext verhält es sich ein wenig anders. Er werden in Zeile 3 zunächst alle Typen der einzelnen Kontexte der interaktiven Komponente ermittelt. Eine interaktive Komponente kann für zwei verschiedene Werte desselben Kontexttyps generiert wurden sein. Zum Beispiel wäre ein Kontext vom Typ *Network-Bandwidth* mit den Werten *low* und *medium* denkbar. Der Kontext wäre dann zweimal im Targetkontext enthalten, mit gleichem Typ, aber unterschiedlichen Werten. Die Komponente wäre somit für beide Werte gültig. Jeder ermittelte Typ aus Zeile 3 wird in der Schleife von Zeile 5 gegen die jeweiligen in der Zielanwendung verfügbaren Werte geprüft. Sind alle Werte geprüft, wird der nächste Typ in der nächsten Iteration der Schleife von Zeile 3 geprüft. Wird die Prüfung erfolgreich bestanden, wird die interaktive Komponente in der Zielanwendung aktiviert, anderenfalls wird sie deaktiviert, indem sie den Status $INACTIVE$ annimmt.

Algorithmus 4.4 Prüfung auf verfügbaren Zielkontext

Require: IC with $IC_{Reg} = (Dependencies, InputParameter, TargetContext)$

function PROOFCONTEXT(IC)
  $targetContextAvailable \leftarrow true$
  for each $ContextType$ **in** $TargetContext \mid TargetContext_{Type_i} \neq TargetContext_{Type_j}; i = 0..n-1, j = i..n$ **do**
    $contextAvailable \leftarrow false$
    for each $Context$ **in** $TargetContext | ContextType = TargetContext_{Type_i}; i = 0..n$ **do**
      $contextAvailable \leftarrow contextAvailable \vee IsContextAvailable(Context_{Type}, Context_{Value})$
    end for
    $targetContextAvailable \leftarrow targetContextAvailable \wedge contextAvailable$
  end for
  if $requiredParameterAvailable$ **then**
    $STATE_{IC} \leftarrow ACTIVATEABLE$
  else
    $STATE_{IC} \leftarrow INACTIVE$
  end if
end function

Der Mechanismus zur Aktivierung bzw. Deaktivierung der interaktiven Komponenten besteht aus den veranschaulichten Algorithmen 4.2, 4.3 und 4.4. Werden alle Prüfungen bestanden, kann die Komponente aktiviert werden. Nicht veranschaulicht wurde, dass es zu einer erneuten Prüfung des Zielkontextes kommt, sobald sich dieser ändert. Dieses Prinzip gilt auch bei Änderung in der Zielanwendung verfügbarer Data-Entities, welche als Eingabeparameter benötigt werden. Ebenso nicht veranschaulicht wurde das Ereignismodell zum Reagieren auf die jeweiligen Zustandsübergänge. Bei jedem ausgelösten Zustandsübergang muss eine Ereignis auslöst werden, um zum Beispiel die jeweilige Komponente zu visualisieren oder diese wieder aus dem User-Interface der Zielanwendung zu entfernen.

4.2. Umsetzung

Nachdem im ersten Teil die wesentlichen Konzepte des gesamten Systems vorgestellt wurden, werden in den folgenden Abschnitten die jeweiligen Konzepte verfeinert dargestellt und detaillierter Lösungsvorschlag präsentiert. Beginnend mit der Umsetzung des Generierungsverfahrens (4.2.1) wird im Anschluss die Umsetzung der Distribution interaktiver Komponenten (4.2.2) vorgestellt. Abschließend werden die Umsetzungen der Konzepte der Zielanwendung (4.2.3) betrachtet.

4.2.1. Umsetzung des Generierungsverfahrens

In diesem Abschnitt wird die Umsetzung der im Abschnitt 4.1.1 vorgestellten Konzepte des Generierungsverfahrens präsentiert. Dabei wird zunächst die Umsetzung der *Server-*

Komponente in Unterabschnitt 4.2.1.1 vorgestellt, darauf folgend werden einzelne Aspekte der Umsetzung der M2C-Transformation in Unterabschnitt 4.2.1.2 präsentiert. Zum Schluss wird in Unterabschnitt 4.2.1.3 die Realisierung der interaktiven Komponenten als Resultat der M2C-Transformation vorgestellt.

4.2.1.1. Umsetzung der Server - Komponente

In diesem Unterabschnitt wird ein Lösungsvorschlag für das in Abschnitt 4.1.1.1 vorgestellte Konzept der *Server-Komponente* vorgestellt. Die realisierte *Server-Komponente* kann als ein Plattformadapter für Android im Sinne des Gesamtansatzes (siehe 1.2) verstanden werden. Realisiert wurde die *Server-Komponente* als Kommandozeilen-Anwendung, wobei der Anwendung beim Start der Ort des zu überwachenden Verzeichnisses und die Zugangsdaten für das Repository der generierten Komponenten als Kommandozeilenparameter übergeben werden. Es ist somit möglich, mehrere Instanzen der Server-Komponente für unterschiedliche Einsatzgebiete auf einem Server parallel auszuführen. Die Architektur der Anwendung, bestehend aus den drei Teilkomponenten, *Generator*, *Directory-Watcher* und *Workflow-Engine*, wird in Abbildung 4.10 detailliert dargestellt.

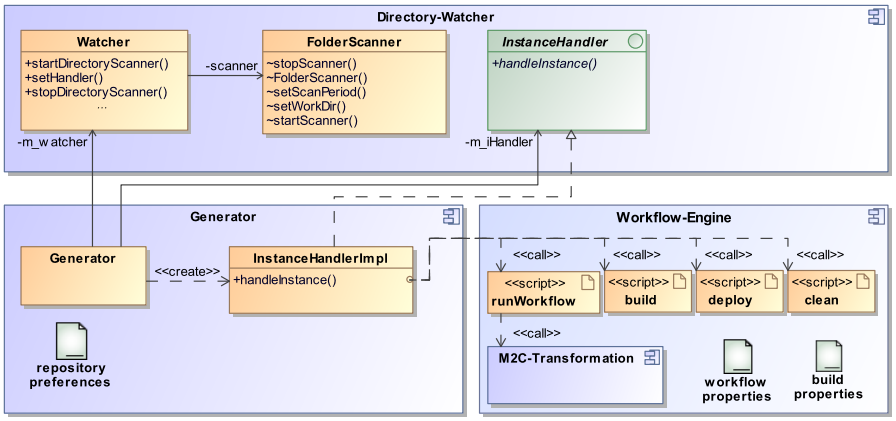

Abbildung 4.10.: Server-Komponente - System-Architektur

Die Überwachung des Verzeichnisses wird durch die *Directory-Watcher - Komponente* übernommen, welche im Rahmen der Arbeit von [Fel11] entwickelt wurde. Diese Komponente ermöglicht die Überwachung eines Verzeichnisses und stellt eine Schnittstelle zur Ereignisbehandlung bei neu vorhandenen Dateien bereit. Konkret besteht die *Directory-Watcher - Komponente* aus einem *DirectoryWatcher* und einem *FolderScanner* und stellt das Interface *InstanceHandler* bereit. Der *FolderScanner*, welcher vom *DirectoryWatcher* initialisiert wird, sucht in regelmäßigen Abständen nach neuen Modellinstanzen. Sobald eine neue Instanz vorliegt, stößt der *DirectoryWatcher* die Generierung an, indem die *handleInstance*-Methode der *InstanceHandler*-Implementierung aufgerufen wird. Diese Implementierung ist Bestandteil der *Generator-Komponente* und steuert die eigentliche Generierung, die Paketierung und die Veröffentlichung der generierten Komponenten mit Hilfe der *Workflow-Engine*. Initialisiert wird diese *InstanceHandlerImpl*-Klasse vom *Generator*,

welcher die eigentliche Anwendung realisiert. Zum Verarbeiten der übergebenen Kommandozeilenparameter wurde die *Apache Commons CLI*-Bibliothek[5] verwendet.

Durch die festgelegte Dreiteilung in *Generator*, *Directory-Watcher* und *Workflow-Engine*, erhöht sich die Wiederverwendbarkeit der *Server-Komponente*. Um einen Plattformadapter für eine weitere Plattform bereitzustellen, müsste nur die *Workflow-Engine* und die *InstanceHandler*-Implementierung ausgetauscht werden und der Generator ggf. leicht angepasst werden, wohingegen der *Directory-Watcher* ohne Änderungen übernommen werden kann.

Zur Steuerung der Workflows zur Paketierung und zum Veröffentlichen generierter Komponenten, wurde *Apache Ant*[6] verwendet, wobei für jede Aufgabe ein Ant-Skript angelegt wurde. Die einzelnen Skripte sind in Anhang A.2 dargestellt. Im Folgenden sollten nur einzelne Teilaspekte einzelner Ant-Tasks beleuchtet werden. Um den Generierungsprozess starten zu können, wurde im Task zum Starten der M2C-Transformation (siehe Listing A.2) der *WorkflowRunner*[7] der *Eclipse Modeling Workflow Engine* verwendet. Dieser startet die M2C-Transformation, deren Umsetzung detailliert in Unterabschnitt 4.2.1.2 vorgestellt wird. Nachdem die Generierung der interaktiven Komponenten abgeschlossen ist, werden die einzelnen Teilkomponenten kompiliert und anschließend zu einem OSGi-Archiv zusammengefasst, siehe Listing A.4. Besonderheit hierbei ist, dass OSGi-Komponenten, welche auf *Android* veröffentlicht werden, neben dem Java-Bytecode das Format zur Ausführung auf der *DalvikVM* (dx - Dalvik Executeable) beinhalten müssen. Dieses Format wird mittels des Werkzeugs *dx*[8] erstellt, welches Bestandteil des *Android SDKs* ist und lediglich den Java-Bytecode als Eingabe benötigt. Zum Veröffentlichen der interaktiven Komponenten (siehe Listing A.5) wurde die *SvnAnt*-Bibliothek[9] verwendet.

4.2.1.2. Umsetzung der Modell-zu-Code - Transformation

In der Realisierung der M2C-Transformation wurde der Abschnitt 4.1.1.2 vorgestellte Algorithmus (Algorithmus 4.1) unter Verwendung der Werkzeuge des *Eclipse Modeling Projects*[10] umgesetzt. Zur Steuerung des Workflows zur M2C-Transformation wurde die Modeling Workflow Engine (MWE) genutzt. Als Sprache für die eigentliche M2C-Transformation wurde Xpand[11] verwendet. Xpand ist eine statisch typisierte Templatesprache mit speziellen, für die Codegenerierung wichtigen Features.

Im Folgenden soll die Umsetzung der Anforderung /AG02/, der Erweiterbarkeit der M2C-Transformation, vorgestellt werden. Wie bereits beschrieben, soll hierbei die Transformation um Teile erweitert werden, welche zur Generierung spezieller UI-Elemente in Form von Widgets dienen. Diese Widgets müssen hierbei die abstrakte Klasse *CustomWidget*

[5] http://commons.apache.org/cli/
[6] http://ant.apache.org
[7] `http://wiki.eclipse.org/MWE_oaw4.3_doc#Starting_the_WorkflowRunner`
[8] http://developer.android.com/guide/developing/tools/othertools.htmldx
[9] http://subclipse.tigris.org/svnant.html
[10] http://www.eclipse.org/modeling
[11] http://wiki.eclipse.org/Xpand

konkretisieren. Das *PreviewWidget*, als konkretes Widget des *DVD-Info-Services* des Heimautomatisierungsszenarios, und die abstrakte *CustomWidget*-Klasse werden in Abbildung 4.11 veranschaulicht.

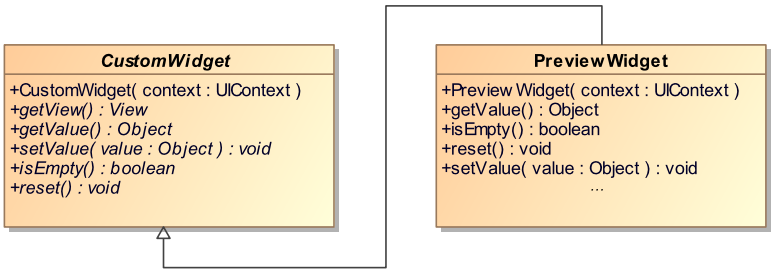

Abbildung 4.11.: CustomWidget-Schnittstelle

Da jede *CustomWidget*-Konkretisierung eine Ein- oder Ausgabe repräsentieren kann, definiert die abstrakte Klasse die abstrakten Methoden *getValue* und *setValue*, zum Erhalten oder Setzen des jeweiligen Wertes. Die abstrakte Methode *reset* dient zum Zurücksetzen des Widgets in den initialen Zustand. Die *isEmpty*-Methode wird vom jeweiligen *Content-Handler* des Widgets benötigt, deren Konzept in Abschnitt 4.1.1.3 bereits vorgestellt wurde.

Der Aufruf eines dynamisch integrierten Xpand-Templates zur Generierung eines konkreten *Custom-Widgets* wird durch einen Xtend-Aufruf, innerhalb des Xpand-Templates zur Generierung der einzelnen UI-Elemente, realisiert und wird in Listing 4.2 veranschaulicht.

```
1  <<IF this.concreteType=="CustomWidget">>
2    <<LET ((ServiceInteractor)this).relevantAnnotations.selectFirst(e|e.type=="CustomWidget") AS cw>>
3      <<IF getStaticAttributeValue((Annotation)cw, "type")=="XPAND">>
4        <<invokeTemplate("dynamicinvoke::templates::"+getWidgetNameByURI(getStaticAttributeValue((
             Annotation)cw, "URI"))+"::createWidget", (ServiceInteractor)this)>>
5      <<ENDIF>>
6      <<getStaticAttributeValue((Annotation)cw, "qualifiedName")>> cw = new <<getStaticAttributeValue
           ((Annotation)cw, "qualifiedName")>>(activity);
7      <<type>> c = cw.getView();
8      <<getContentHandlerName()>> = new rn.icl.common.view.handler.CustomWidgetHandler(cw);
9      <<storeCustomWidgetNamespace(getWidgetNamespace(getStaticAttributeValue((Annotation)cw, "
           qualifiedName")))>>
10   <<ENDLET>>
11 <<ELSEIF type="...">>
```

Listing 4.2: Aufruf von dynamisch zur Generierungszeit integrierten Templates

Während der Erstellung der einzelnen UI-Elemente wird geprüft, ob ein Interaktor vom Typ *„CustomWidget“* ist. Ist dies der Fall, wird, falls die *CustomWidget*-Annotation auf ein Xpand-Template verweist, das dazugehörige Template mittels des Xtend-Aufrufs *invokeTemplate* aufgerufen. Im Falle des *Preview-Widgets*, würde dem *invokeTemplate*-Aufruf *„dynamicinvoke::templates::PreviewWidget::createWidget“* und der dazugehörige Service-Interaktor übergeben. Die dazugehörige Xtend-Erweiterung leitet diesen Aufruf an

den *TemplateInvoker* weiter, welcher die Ausführung des dynamisch integrierten Xpand-Templates veranlasst und in Abbildung 4.12 dargestellt wird.

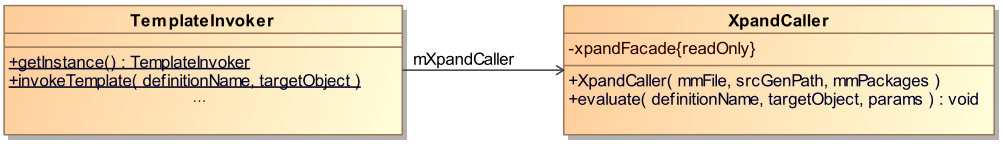

Abbildung 4.12.: Xpand-Template - Erweiterung

Der *TemplateInvoker* wurde hierbei als *Singleton* (vgl. [GHJV95]) realisiert, welcher einen *XpandCaller* initialisiert, der dann den eigentlichen Template-Aufruf tätig. Zur Initialisierung des *XpandCallers* muss diesem das zugrundeliegende Metamodell, der Pfad, in welchem die generierten Dateien gespeichert werden sollen und die verwendeten Packages des Metamodells übergeben werden. Mit Hilfe dieser Informationen kann der *XpandCaller* einen *XpandExecutionContext*[12] erzeugen, der mit Hilfe einer *XpandFacade*[13] dazu verwendet werden kann, Xpand-Templates zur Laufzeit der Generierung dynamisch unter Verwendung von Informationen der verwendeten Modellinstanz aufzurufen. Die detaillierte Implementierung des *XpandCallers* ist im Anhang unter Listing A.6 veranschaulicht.

4.2.1.3. Umsetzung der Interaktiven Komponente

In diesem Abschnitt werden einzelne Aspekte zur Umsetzung der in Abschnitt 4.1.1.3 vorgestellte Konzepte vorgestellt.

Datenmodell Wie bereits beschrieben, wurde in der Konzeption der interaktiven Komponente ein generisches Datenmodell verwendet, wobei Instanzen von *SelectionPath*-Objekten zur Referenzierung einzelner Datenobjekte dienen. Abbildung 4.13 veranschaulicht die Umsetzung der in Absatz 4.1.1.3 vorgestellten Konzepte des Datenmodells.

Ein *SelectionPath* wird mit Hilfe der *SelectionPathFactory* erstellt. Diese instanziiert für jeden Knoten eines *Seclection-Paths* einen *SelcectionNode* und fügt diese in einem *SelectionPath*-Objekt zusammen. Zum Abfragen einen Datenobjekts wurde die Schnittstelle *ObjectResolver* spezifiziert. Zum Erstellen eines Datenobjekts wurde die Schnittstelle *ObjectCreator* spezifiziert. Die Konkretisierungen dieser Schnittstellen hängen von der verwendeten Bibliothek zur Webservice-Kommunikation ab. Da im Rahmen dieser Arbeit *kSOAP 2*[14] verwendet wurde, wurde ein *KsoapObjectResolver* und ein *KsoapObjectCreator* umgesetzt.

[12] `http://download.eclipse.org/modeling/m2t/xpand/javadoc/0.7.0/org/eclipse/xpand2/XpandExecutionContext.html`
[13] `http://download.eclipse.org/modeling/m2t/xpand/javadoc/0.7.0/org/eclipse/xpand2/XpandFacade.html`
[14] http://ksoap2.sourceforge.net/

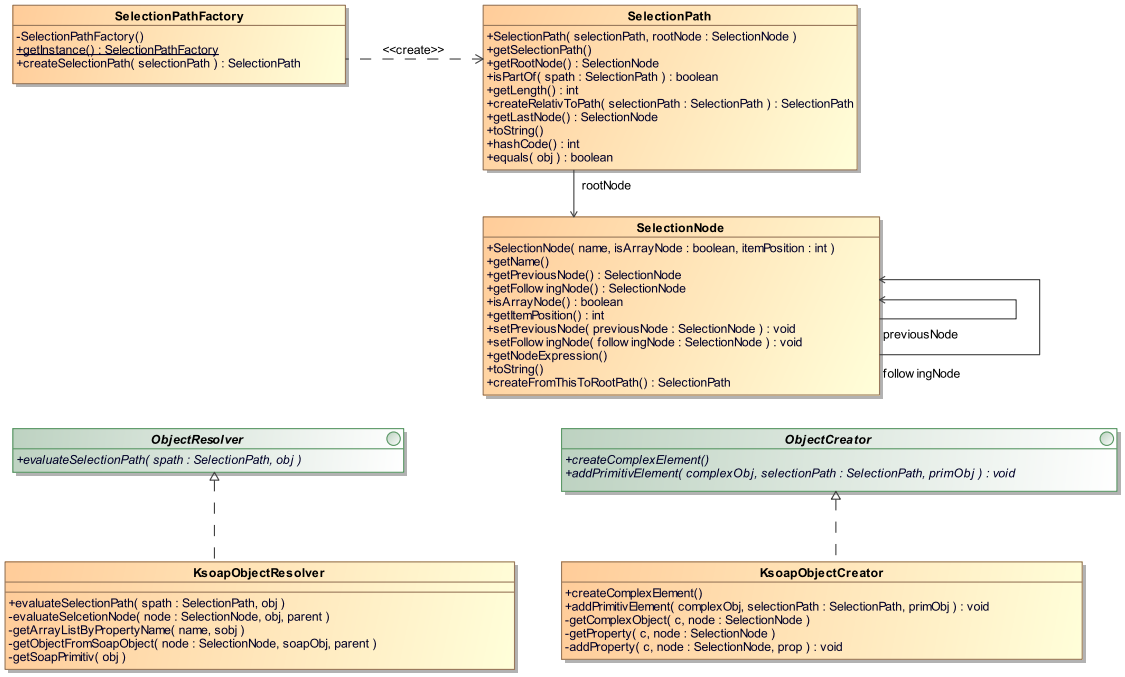

Abbildung 4.13.: Umsetzung des generischen Datenmodells

Content-Handler Abblildung 4.14 veranschaulicht die bereits im Konzept der interaktiven Komponenten vorgestellten *Content-Handler*. Jedem nativen UI-Element wird mit einem *ContentHandler* bzw. mit einem *ActivatorHandler* versehen, der dann als Adapter zum eigentlichen Element fungiert. Hierbei werden alle UI-Elemente, welche in der urspünglichen SBIC-Instanz als *Activator* repräsentiert wurden, mit einem Handler versehen, welcher die abstrakte Klasse *ActivatorHandler* konkretisiert. In der momentanen Umsetzung ist dies ausschließlich die Klasse *ButtonHandler*. Alle anderen Handler, welche in der SBIC-Instanz als *Service-Interaktoren* repräsentiert wurden, müssen die abstrakte Klasse *ContentHandler* konkretisieren.

Dadurch, dass alle Elemente zur Ein- und Ausgabe von einer abstrakten Klassen erben, ist die Realisierung von Funktionalitäten, welche aus Annotationen resultieren und somit für beliebige Interaktoren gültig sind, einfach möglich. Zum Beispiel wird für jedes UI-Element eines Interaktors, welcher mit der *Mandatory*-Annotation versehen wurde, der dazugehörige *ContentHandler* beim *ActivatorHandler* des Aktivators zum dazugehörigen Operationsaufruf als *madatoryInput* registriert. Das gleiche Prinzip wurde für *Validation*-Annotation angewendet. Die konkreten *ContentHandler* müssen hierbei die Methoden *isEmpty* und *isValid* implementieren.

4.2.2. Umsetzung der Distribution interaktiver Komponenten

Das Konzept zur Distribution interaktiver Komponenten wurde in Unterabschnitt 4.1.2 bereits vorgestellt. In diesem Unterabschnitt werden einzelne Details zur Umsetzung des

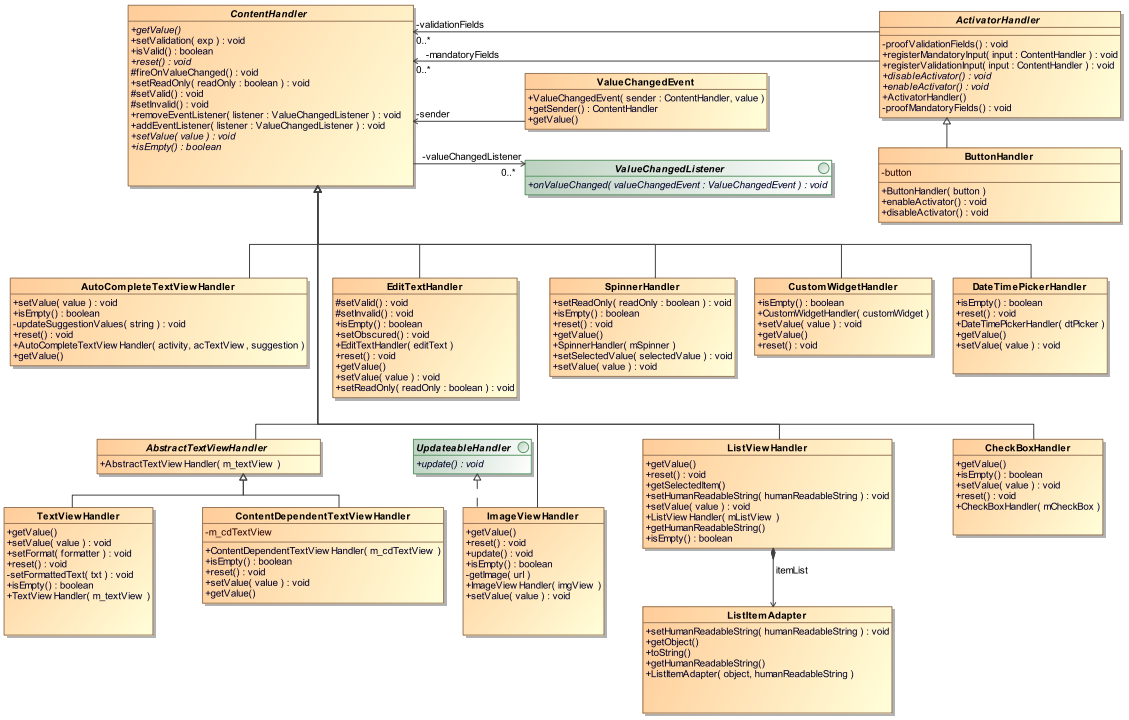

Abbildung 4.14.: Umsetzung der Content-Handler

beschriebenen Konzeptes präsentiert. Wie bereits diskutiert, soll das Repository für jede beinhaltende Komponente die dazugehörige Revisionsnummer zurückliefern. Umgesetzt wurde das Bundle-Repository mit Hilfe von Apache Subversion (SVN)[15]. SVN ist eine freie Software zur Versionsverwaltung von Dateien und Verzeichnissen. Auf ein SVN-Repository kann hierbei über verschiedene Protokolle zugegriffen werden. Für die Realisierung des Bundle-Repositories wurde jedoch ein Modul für den *Apache-Webserver*[16] verwendet, das es ermöglicht, die Dateien mittels des WebDAV-Protokolls übertragen zu können, welches eine Erweiterung des HTTP/HTTPS-Protokolls darstellt [Dus07]. Zentrale Rolle spielt hierbei die *Propfind*-Methode des WebDAV-Protokolls, welche Eigenschaften einer angefragten Ressource zurückliefert. Ein Bestandteil dieser Eigenschaften ist die Revisonsnummer einer angefragten Datei bzw. die Revisionnummern aller Dateien eines angefragten Verzeichnisses. Es ist somit durch einen einzigen WebDAV-Request möglich, die Revisionsnummern aller verfügbaren Dateien zu erhalten. Der Entity-Body eines solchen Requests ist in Listing 4.3 veranschaulicht.

Die Response-Message des Propfind-Request wird in der clientseitigen Implementierung mit Hilfe des *PropfindResponseParsers* ausgewertet, welcher Bestandteil der Klasse *BundleRepository* ist, die wiederum Bestandteil der *Repository-Komponente* der Zielanwendung ist und in Abbildung 4.15 veranschaulicht wird. Das *BundleRepository* verwaltet hierbei eine Liste der lokal verfügbaren Bundles und deren assoziierten Revisionsnummern. Wird

[15] http://subversion.apache.org
[16] http://httpd.apache.org

```
<?xml version="1.0"?>"
<a:propfind xmlns:a="DAV:">
  <a:prop>
    <a:version-name/>
  </a:prop>
</a:propfind>
```

Listing 4.3: Propfind Request zum Ermitteln der Revisionsnummern

ein *BundleRepository* für ein lokales Bundleverzeichnis initial angelegt, wird zunächst eine lokale Liste ohne Einträge erstellt.

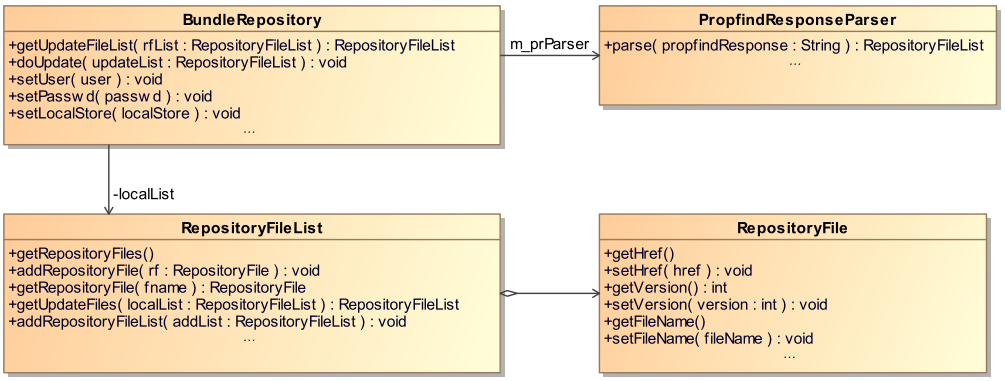

Abbildung 4.15.: Komponenten-Repository

Die Methode *getUpdateFileList* der *BundleRepository*-Klasse dient zum Ermitteln der Dateien, welche neu vorhanden sind oder von denen eine neuere Versionen verfügbar ist. Dabei wird zunächst die Liste der verfügbaren Dateien aus dem zentralen Repository ermittelt und anschließend mit der lokalen Liste verglichen. Mit Hilfe der Methode *doUpdate* können diese Dateien dann heruntergeladen werden, wobei gleichzeitig die lokalen Meta-Daten aktualisiert werden, welche aus dem Dateinamen und der Dateiversion bestehen. Organisiert werden diese Meta-Daten mit Hilfe der Klasse *RepositoryFileList*, welche eine Liste von Dateien mit assoziierten Zusatzinformationen repräsentiert und Methoden anbietet, die den Vergleich zweier Repository-Dateilisten vereinfacht. Eine einzelne versionierte Datei wird durch die Klasse *RepositoryFile* repräsentiert, welche aus dem Ort der Datei im zentralen Repository, der Version der Datei und dem Dateinamen besteht.

Durch die Verwendung eines SVN-Repository in Kombination eines Apache-Webservers, war es möglich eine leichtgewichtige clientseitige Lösung zu implementieren, die sich für die Verwendung in einer mobilen Umgebung sehr gut eignet.

4.2.3. Umsetzung der Zielanwendung

In den folgenden Abschnitten wird die Umsetzung, der aufgestellten Konzepte der Zielanwendung aus Abschnitt 4.1.3, präsentiert. In Unterabschnitt 4.2.3.1 werden zunächst die einzelnen Umsetzungen der Teilkomponenten der Kern-Bibliothek vorgestellt, welche im

Abschnitt zur Architektur der Zielanwendung (4.1.3.1) bereits vorgestellt wurden. Daraufhin wird die Lösungsvorschlag für die Registrierung interaktiver Komponenten in der Zielanwendung in Unterabschnitt 4.2.3.2 detailliert präsentiert. Abschließend wird in Unterabschnitt 4.2.3.3 eine Lösung zur Ansteuerung interaktiver Komponenten seitens der Zielanwendung vorgestellt.

4.2.3.1. Umsetzung der Teilkomponenten der Kern-Bibliothek

Wie bereits im Abschnitt zur Architektur (4.1.3.1) erwähnt, besteht eine konkrete Zielanwendung minimal aus einzelnen Klassen, welche Klassen der Kern-Bibliothek parametrisieren oder konkretisieren. Dies wird ersichtlich in Abbildung 4.16, welche einen Überblick über die Kern-Bibliothek verschafft. Um die Übersichtlichkeit zu wahren, enthält die Abbildung nicht alle Klassen und Schnittstellen.

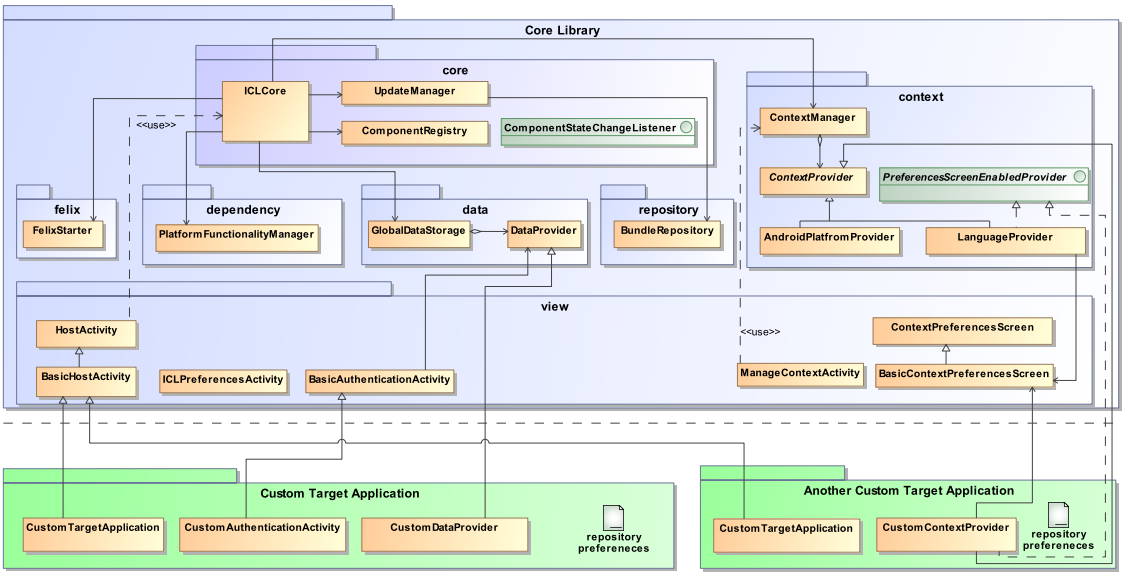

Abbildung 4.16.: Architektur der Kern-Bibliothek

Die Core-Komponente bildet das Herzstück der Kern-Bibliothek. Dies wird in Abbildung 4.16 deutlich. Eine zentrale Rolle nimmt hierbei der *ICLCore* an. Der *ICLCore* wurde als *Singleton* umgesetzt. Er hält die einzelnen Referenzen zu den jeweils anderen Komponenten und sorgt für deren Zusammenwirken. Der *UpdateManger*, ebenfalls als *Singleton* realisiert, steuert mit Hilfe der *Repository-Komponente* das Laden interaktiver Komponenten aus dem entfernten Repository. Die *ComponentRegistry* der *Core-Komponente* regelt die Registrierung interaktiver Komponenten. Auf das Registrierungsverfahren wird im Detail in Unterabschnitt 4.2.3.2 eingegangen. Zustandsänderungen der installierten interaktiven Komponenten können mit Hilfe der *ComponentStateChangeListener*-Schnittstelle überwacht werden. *ComponentStateChangeListener*-Implementierungen fungieren hierbei als *Observer* (vgl. [GHJV95]), wofür sie sich bei der *ComponentRegistry*, dem *Subject*, zuvor registrieren müssen. Sobald ein interaktive Komponente geladen, entfernt, aktiviert bzw.

deaktiviert wird, werden alle registrierten *ComponentStateChangeListener*-Realisierungen benachrichtigt.

Die Felix-Komponente regelt, wie bereits beschrieben, das Laden und die Verwaltung der OSGi-Bundles. Zur Ereignisbehandlung wurden die Schnittstellen *ServiceChangedListener* und *ServiceChangedEvent* eingeführt, welche allerdings in Abbildung 4.16 aus Platzgründen nicht veranschaulicht wurden. Der *ServiceChangedListener* fungiert hierbei als *Observer*, dessen Realisierungen vom *ICLCore* der *Core-Komponente* und von der *Dependency-Komponente* verwendet werden. Der *ICLCore* und die *Dependency-Komponente* werden somit immer benachrichtigt, sobald ein OSGi-Bundle hinzugefügt, geändert oder entfernt wurde. Im *ICLCore* werden diese Informationen gefiltert, handelt es sich bei einem solchen Bundle um die interaktive Komponente, wird dieses Ereignis an die *ComponentRegistry* weitergereicht, damit diese die Registrierung der Komponente vornehmen kann. Die *Dependency-Komponente* realisiert einen *ServiceChangedListener*, um überwachen zu können, ob eine angeforderte zusätzliche Plattformfunktionalität in Form eines OSGi-Bundles, erfolgreich installiert wurde.

Die Dependency-Komponente steuert wie eben beschrieben die Bereitstellung von zusätzlichen Plattformfunktionalitäten. Die einzige Klasse dieser Komponente ist der *PlatformFunctionalityManager*, welcher als *Singleton* umgesetzt wurde. Der *PlatformFunctionalityManager* stellt drei Methoden zur Verfügung, *isServiceAvailable*, *requestPlatformFunctionality* und *createPFHandle*, deren Zusammenwirken in Abbildung 4.17 deutlich wird.

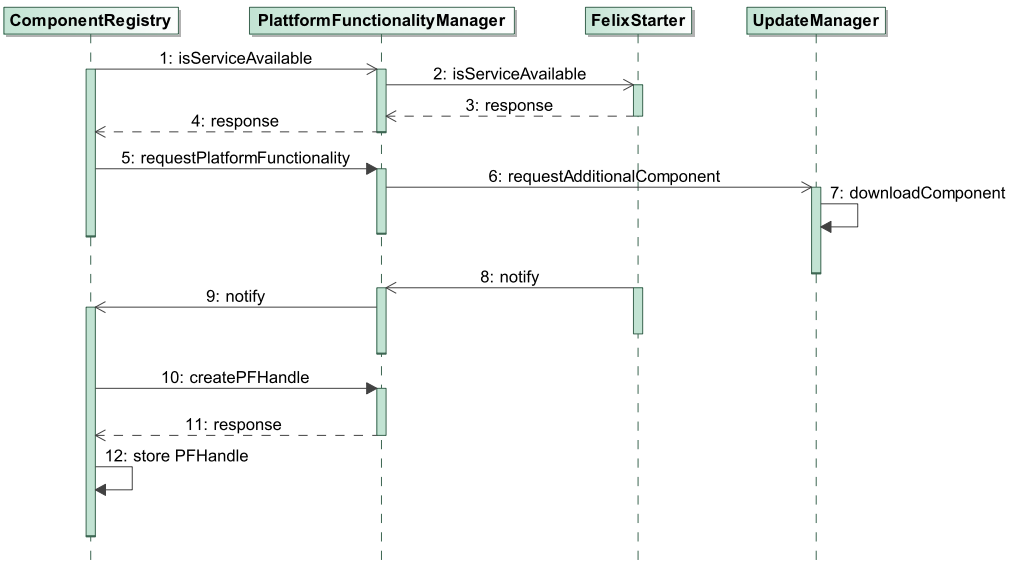

Abbildung 4.17.: Installieren einer Plattformfunktionalität bei Nichtverfügbarkeit

Muss die *ComponentRegistry* während der Registrierung einer interaktiven Komponente feststellen, ob eine Plattformfunktionalität verfügbar ist, erfragt sie dies beim *PlatformFunctionalityManager* mittels der *isServiceAvailable*-Methode. Der *PlatformFunctionalityManager* erfragt daraufhin die Verfügbarkeit der Funktionalität beim *FelixStarter* und gibt

das Ergebnis an die *ComponentRegistry* weiter. In Abbildung 4.17 ist der Fall der Nichtverfügbarkeit dargestellt. Im Falle der Verfügbarkeit würden die Punkte 5 bis 9 übersprungen werden und sofort mit Punkt 10 weiter verfahren werden. Da die Funktionalität aber nicht verfügbar ist, fordert die *ComponentRegistry* diese beim *PlatformFunctionalityManager* mittels der *requestPlatformFunctionality*-Methode an, welcher wiederum die Funktionalität in Form einer OSGi-Komponente beim *UpdateManager* anfordert. Dieser lädt die Komponente dann herunter. Wie bereits beschrieben, wird der *PlatformFunctionalityManager* von *FelixStarter* informiert, sobald eine neue Komponente verfügbar ist. Ist eine solche neue Komponente eine der beim *UpdateManager* angeforderten, wird die *ComponentRegistry* über deren Verfügbarkeit informiert. Die *ComponentRegistry* lässt sich dann, sofern alle geforderten Plattformfunktionalitäten verfügbar sind, ein Handle erzeugen, mit welchem die von der jeweiligen interaktiven Komponente geforderten Funktionalitäten über eine einheitliche Schnittstelle aufgerufen werden können. Einer interaktiven Komponente wird somit nur der Zugang zu den explizit geforderten Funktionalitäten gewährt. Eine detaillierte Beschreibung zu diesem Handle erfolgt im Unterabschnitt zur Ansteuerung interaktiver Komponenten (4.2.3.3). Zum Schluss wird das erstellte Handle in der *ComponentRegistry* gespeichert, um dieses zum Starten der jeweiligen Komponente verwenden zu können.

Die Data-Komponente stellt, wie bereits beschrieben, Data-Entities global in der Zielanwendung zur Verfügung. Die Bereitstellung solcher Data-Entities erfolgt mit Hilfe der Klasse *DataProvider* der *Data-Komponente*, die jeder Bereitsteller instanziieren muss. Um Data-Entities global in der Zielanwendung verfügbar zu machen, muss der *DataProvider* mittels der Methode *registerDataProvider* im *GlobalDataStorage* registriert werden. Der *GlobalDataStorage* wird dadurch beim Hinzufügen, beim Updaten oder beim Entfernen eines Data-Entities informiert und kann mittels der Methode *isValueAvailable* mitteilen, ob ein Data-Entity verfügbar ist bzw. mittels der Methode *getValue* den aktuellen Wert eines Data-Entities zurückliefern. Die Kommunikation zwischen den *DataProvider*-Instanzen und dem *GlobalDataStorage* wird mit Hilfe der *DataChangeListener*-Schnittstelle realisiert, wobei der *GlobalDataStorage* die Rolle des *Observers* einnimmt und die *DataProvider*-Instanzen die Rolle der jeweiligen *Subjects* annehmen. In Abbildung 4.16 ist ersichtlich, wie eine Authentifizierung mit Hilfe einer *DataProvider*-Instanz realisiert werden kann, allerdings wurde die *DataChangeListener*-Schnittstelle aus Platzgründen vernachlässigt. Die *Custom Target Application* konkretisiert im gezeigten Beispiel die *BasicAuthenticationActivity* der Kern-Bibliothek, welche bei erfolgreicher Authentifizierung den erhaltenen Session-Key mittels eines Data-Providers für andere Komponenten verfügbar macht. Die *Data-Komponente* bietet somit die Möglichkeit, Daten in die Zielanwendung global verfügbar zu machen, welche wiederum zum Start einer interaktiven Komponente verwendet werden können.

4.2.3.2. Umsetzung des Registrierungsverfahrens

Die Daten, welche im Unterabschnitt 4.1.3.2 für die Registrierung interaktiver Komponenten in der Zielanwendung benötigt werden, wurden mit Hilfe spezifischen Datenstruktur spezifiziert. Das Generierungsverfahren stellt dabei sicher, dass die interaktive Komponente Instanzen der spezifizierten Datenstruktur erzeugt und diese der Zielanwendung bereitstellt. Die Registrierungsdaten sind somit Teil des Protokolls, welches die Kommunikation zwischen der Zielanwendung und den interaktiven Komponenten regelt. In Abbildung 4.18 wird die Datenstruktur der Registrierungsdaten veranschaulicht.

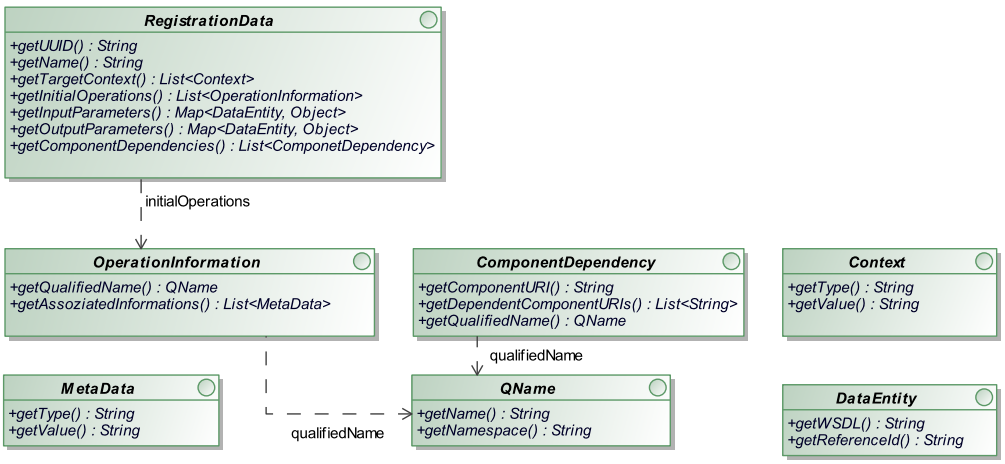

Abbildung 4.18.: Registrierungsdaten

Minimal muss die Interaktive Komponente für die Registrierung in der Zielanwendung einen eindeutigen Bezeichner übermitteln. Dies wird über die *getUUID*-Methode realisiert, wobei die Implementierung einen eindeutigen Bezeichner in Form eines Strings zurückliefern muss. Über die *getName*-Methode wird sichergestellt, dass die Implementierung einen Namen der interaktiven Komponente übermittelt, welcher zur Visualisierung der Komponente verwendet werden kann. Der Zielkontext wird der Zielanwendung mit Hilfe der *getTargetContext*-Methode bekannt gemacht. Diese liefert eine Menge von *Context*-Implementierungen zurück, welche aus einem Kontext-Typ und dessen Wert bestehen. Unterstützt die interaktive Komponente eine direkte Operationsansteuerung, werden die hierfür benötigten Information mit Hilfe der *getInitialOperations*-Methode ermittelt. Es wird eine Menge von *OperationInformation*-Implementierungen zurückgeliefert, bestehend aus einem vollqualifizierten Namen zur Operationsansteuerung und mit der Operation assoziierter Zusatzinformationen in Form von *MetaData*-Implementierungen, welche der Visualisierung in der Zielanwendung dienen. Soll ein menschenlesbarer Bezeichner übermittelt werden, würde eine *MetaData*-Instanz zum Beispiel einen Typ mit dem Wert *Text-Feedback* und mittels der *getValue*-Methode einen Bezeichner mit dem Wert *Show Channel List* zurück liefern. Benötigte Eingabe oder bereitgestellte Ausgabeparameter kann die Zielanwendung mit Hilfe der *getInputParameters*- bzw. der *getOutputParameter*-Methode ermitteln. Die eindeutigen Bezeichner für die jeweiligen Parameter wurden durch die *DataEntity*-Schnittstelle realisiert. Mittels der *getComponentDependencies*-Methode kann die

Zielanwendung eventuelle Abhängigkeiten ermitteln, die ggf. noch nachinstalliert werden müssen. Die spezifizierten Registrierungsdaten in Form einer Schnittstellenbeschreibung definieren die notwendigen Daten zur Registrierung einer interaktiven Komponente in der Zielanwendung.

4.2.3.3. Ansteuerung interaktiver Komponenten

In Anschnitt 4.2.3.2 wurde erwähnt, dass die Registrierungsdaten Bestandteil des Protokolls zur Kommunikation zwischen der Zielanwendung und der interaktiven Komponente sind. Das Konzept für das gesamte Protokoll zur Ansteuerung interaktiver Komponenten wurde wie das Konzept der Registrierungsdaten in [Fel11] beschrieben. Umgesetzt wurde das Protokoll in Form einer Datenstruktur, welche in Abbildung 4.19 veranschaulicht wird.

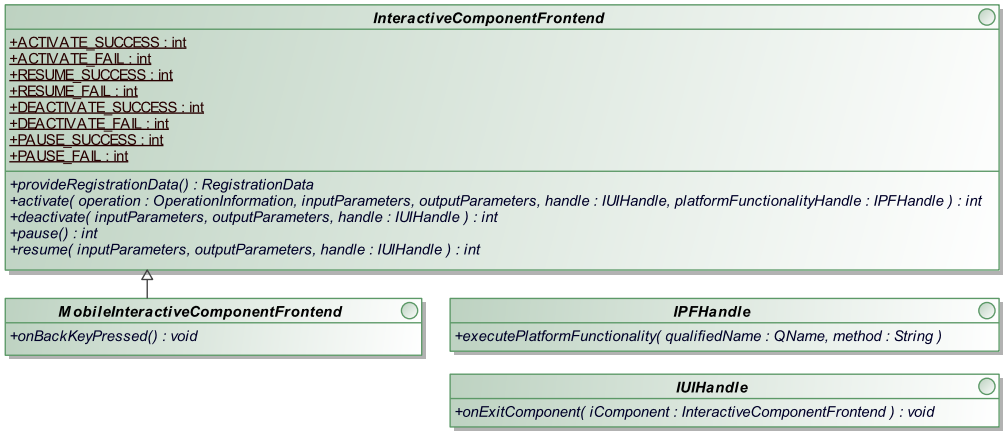

Abbildung 4.19.: Datenstruktur zur Ansteuerung interaktiver Komponenten

Über der Datenstruktur *InteractiveComponentFrontend*, welche direkt aus der Spezifikation von [Fel11] resultiert, macht sich eine interaktive Komponente gegenüber der Zielanwendung bekannt. Diese Schnittstelle wurde um die Methode *onBackKeyPressed* erweitert, um der Komponente das Ereignis weiterreichen zu können, welches beim Drücken der Zurück-Taste auf dem mobilen Endgerät eintritt. Mit Hilfe der Methode *provideRegistrationData* ermittelt die Zielanwendung die Daten zur Registrierung, welche detailliert in Abschnitt 4.2.3.2 behandelt wurden.

Mittels der Methode *activate*, kann die interaktive Komponente gestartet werden. Unterstützt diese eine direkte Operationsansteuerung, muss der *operation*-Parameter die jeweilige *OperationInformation* enthalten. Die Parameter *inputParameter* und *outputParameter* dienen der Übergabe von Eingabeparametern bzw. zum Ermitteln von Rückgabeparametern. Mittels des *handle*-Parameters wird der Komponente ein Handle für das UI übergeben. Die Komponente kann der Zielanwendung mit Hilfe der *onExitComponent*-Methode dieses Handles mitteilen, dass die Komponente beendet werden soll. Der *platformFunctionalityHandle*-Parameter dient zur Übergabe eines Handles zur Ansteuerung der geforderten Plattformfunktionalitäten resultierend aus den Registrierungsdaten der interakti-

ven Komponente. Mittels der *executePlatformFunctionality*-Methode kann eine spezielle Plattformfunktionalität aufgerufen werden, welche mit Hilfe eines vollqualifierten Namens und dem Methodennamen der Funktionalität identifiziert wird. Die Implementierung dieses Handles wurde hierbei mittels Reflection realisiert und wird bei der Registrierung einer Komponente speziell für jede interaktive Komponente, welche Plattformfunktionalitäten anfordert, erstellt. Dabei kann jeweilige interaktive Komponente nur auf geforderte Plattformfunktionalitäten zugreifen. Erfolgt eine erfolgreiche Aktivierung der Komponente, liefert die *activate*-Methode den Status $ACTIVATE\_SUCCESS$ zurück, anderenfalls wird $ACTIVATE\_FAIL$ zurückgeliefert.

Die *deactivate*-Methode des *InteractiveComponentFrontends* dient zum Stoppen einer gestarteten interaktiven Komponente. Die Parameter *inputParameter* und *outputParameter* entsprechen der Semantik der selbigen Parameter der eben beschriebenen *activate*-Methode. Kann eine Komponente erfolgreich beendet werden, liefert die *deactivate*-Methode den Status $DEACTIVATE\_SUCCESS$ zurück, andernfalls analog zur *activate*-Methode wird als Status $DEACTIVATE\_FAIL$ zurückgeliefert.

Mittels der Methoden *pause* und *resume* kann eine interaktive Komponente pausiert bzw. wieder fortgesetzt werden. Beim Pausieren sichert die interaktive Komponente ihren internen Zustand, welcher beim Fortsetzen der Komponente, wiederhergestellt wird. Analog zur *activate*- und *deactivate*-Methode liefern die Methoden zum Pausieren und Fortsetzen den Status über den Erfolg der jeweiligen Operation zurück.

4.3. Zusammenfassung

In diesem Kapitel wurde zunächst die Konzeption eines Systems zur vollautomatischen Generierung und Distribution von interaktiven dienstbasierten Komponenten und die Konzeption der Anwendung, welche die veröffentlichten Komponenten integriert, vorgestellt. Im Anschluss wurden Lösungen der vorgestellten Konzepte im Abschnitt zur Umsetzung (siehe 4.2) präsentiert. Dabei ist ein komplexes System entstanden, wobei alle aufgestellten Konzepte realisiert werden konnten. Der manuell erstellte Code aller entstanden Software-Komponenten umfasst ca. 8.700 Lines of Code (LOC). Eine generierte interaktive Komponente eines Dienstes des verwendeten Szenarios umfasst im Durchschnitt ca. 1.300 LOC. Interessant wird die Betrachtung der Zielanwendung. Der Kern einer Zielanwendung, welcher für alle konkreten Zielanwendungen identisch ist, umfasst ca. 2.700 LOC, wohingegen der variable Teil bzw. der Code einer konkreten Zielanwendung, wie sie in dieser Arbeit umgesetzt wurde, nur lediglich 50 LOC umfasst. Hier wird noch einmal deutlich, dass eine Zielanwendung mit einem relativ geringen Aufwand an einen speziellen Anwendungskontext angepasst werden kann. Zum Erstellen der Server-Komponente als lauffähige Anwendung, wurde ein Ant-Script erstellt, welches alle benötigen Projekte erstellt, für das Vorhandensein der notwendigen Bibliotheken sorgt und die Server-Komponente als ZIP-Archiv in einem vordefinierten Verzeichnis ablegt. Dieses Archiv kann dann leicht auf den

Server übertragen werden, auf dem die Server-Komponente in Form eines Android-Adapter installiert und gestartet werden soll.

5. Evaluation

In diesem Kapitel wird das entwickelte Konzept und der in Kapitel 4 vorgestellte Prototyp validiert. In Abschnitt 5.1 wird zunächst der entstandene Prototyp anhand des vorgestellten Szenarios zur Heimautomatisierung (siehe 2.1) evaluiert. Daraufhin folgend wird der Prototyp unter Verwendung des Szenarios von [Mar10] validiert, wobei auf einzelne Aspekte eingegangen wird. Im Anschluss wird in Abschnitt 5.5 eine Validierung der aufgestellten Anforderungen (siehe 2.2) durchgeführt. Die Validierung der Anforderungen bezüglich der Benutzbarkeit wird in Abschnitt 5.3 gesondert betrachtet. Zum Abschluss wird das in der Einleitung vorgestellte Gesamtkonzept (siehe **??**), bestehend aus den Interactive Component Editor (ICE) von [Mar10], der Inferenz-Engine von [Fel11] und dem im Rahmen dieser Arbeit entstandenen Generator bzw. Prototyp der Zielanwendung, validiert.

5.1. Evaluierung anhand des eingeführten Szenarios

Das im Rahmen dieser Arbeit verwendete Anwendungsszenario (siehe 2.1), beschreibt eine Anwendung zur mobilen Verwaltung von heterogen verteilten Geräten eines Heimnetzwerks. Jedes Gerät stellt hierbei seine Funktionalitäten in Form eines Webservices zur Verfügung. Im Folgenden soll das entstandene System zur Generierung interaktiver Komponenten und der entstandene Prototyp zur Integration interaktiver Komponenten anhand des Heimautomatisierungsszenarios evaluiert werden. Ausgangspunkt bildet hierbei die Server-Komponente, welche ein Verzeichnis überwacht und für jede Modellinstanz, welche in diesem Verzeichnis hinterlegt wird, eine interaktive Komponente generiert und diese anschließend in einem zentralen Komponenten-Repository veröffentlicht. Beim Start der Server-Komponente, müssen dieser das zu überwachende Verzeichnis und die Zugangsparameter des Komponenten-Repositories mittgeteilt werden. Abbildung 5.1 veranschaulicht den Start der Server-Komponente.

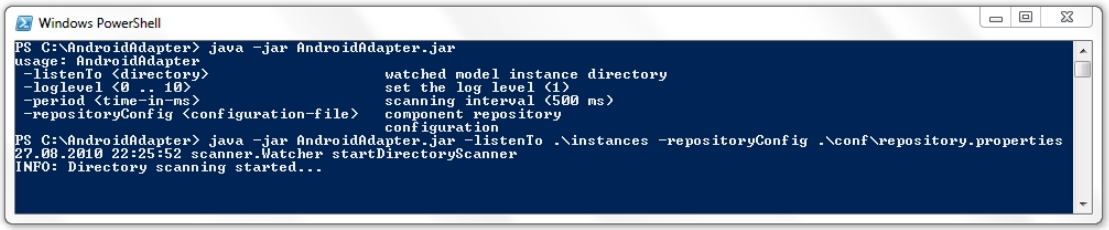

Abbildung 5.1.: Server-Komponente - Initialer Zustand

Der erste Aufruf der Server-Komponente zeigt eine Auflistung der benötigten Eingabeparameter. Wie bereits beschrieben muss das zu überwachende Verzeichnis und Zugangsparameter des Komponenten-Repositories übergeben werden. Zusätzlich kann ein Intervall angegeben werden, in dessen Abstand das zu überwachende Verzeichnis durchsucht wird und ein Log-Level der Ausgabe angegeben werden. Die Zugangsparameter werden in Form einer Konfiguartionsdatei angegeben. Listing 5.1 veranschaulicht exemplarisch eine solche Konfiguration.

```
#component repository location
repository.dir = /ICGenerator/repositories/BundleRepositoryHome
repository.iComponentFolder = InteractiveComponents

#component repository authentication
repository.user = berndt
repository.pass = avb43w7tvabrg
```

Listing 5.1: Komponenten-Repository - Konfiguration

Die Konfiguration des Komponenten-Repositories besteht aus dem Ort des lokalen Archivs und den Zugangsdaten bestehend aus Benutzername und Passwort. Das zu überwachende Verzeichnis, der zweite benötigte Parameter, wird der Server-Komponente mit dem Parameter *listenTo* übergeben. Wird in diesem Verzeichnis dann eine Modellinstanz abgelegt, startet die automatische Generierung und die anschließende Veröffentlichung der jeweiligen interaktiven Komponente. In Abbildung 5.2 wird die Ausgabe der Server-Komponente nach der Generierung und der Veröffentlichung der interaktiven Komponente des *DVBInfo-Services* veranschaulicht.

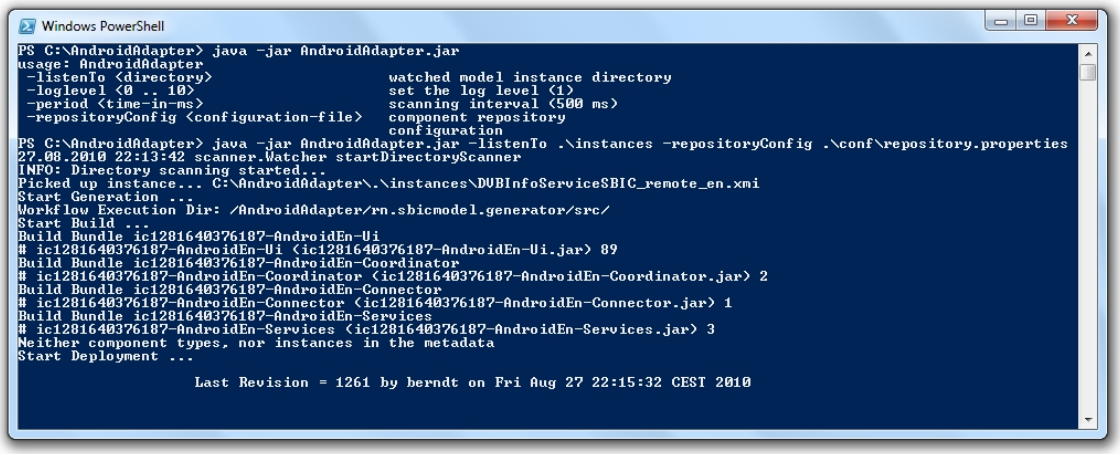

Abbildung 5.2.: Server-Komponente - Ausgabe der Generierung und Veröffentlichung

Die Ausgabe nach der Veröffentlichung einer interaktiven Komponente endet mit der Angabe der aktuellen Revisionsnummer des Komponenten-Repositories. Ab diesem Zeitpunkt ist die interaktive Komponente für die Zielanwendung verfügbar und kann heruntergeladen und installiert bzw. aktualisiert werden. Sicht (a) der Abbildung 5.3 veranschaulicht

die Sicht nach dem Starten des *My Home Managers*, der Zielanwendung des Heimautomatisierungsszenarios. Nach einer erfolgreichen Authentifizierung wird der erhaltene Session-Key als global verfügbares Data-Entity gespeichert und die verfügbaren Komponenten werden visualisiert. Sicht (b) der Abbildung 5.3 zeigt den Zustand vor der Generierung und Veröffentlichung der interaktiven Komponente des *DVDInfo-Services*. Die bereits vorhandenen Komponenten des *Alarm-System-Services*, des *Lighting-Services* und des *Router-Configuration-Services* wurden im Vorfeld bereits generiert, veröffentlicht und in die Zielanwendung integriert. Der Zustand der aktualisierten Komponenten nach der Generierung und Veröffentlichung der interaktiven Komponente des *DVDInfo-Services* wird in Sicht (c) veranschaulicht, wobei zu erkennen ist, dass sich die Komponente in die Liste der verfügbaren interaktiven Komponenten eingereiht hat. Bei der Visualisierung der integrierten interaktiven Komponenten werden jeweils die initialen Operationen der dazugehörigen Komponente veranschaulicht. Die Information für die Beschriftung der UI-Elemente zum Ansteuern der Operationen werden der Zielanwendung von der interaktiven Komponente mitgeteilt. Sicht (b) und Sicht (c) der Abbildung 5.3 zeigen jeweils die Hauptansicht der Zielanwendung unter Verwendung von *Englisch* als Sprachkontext. Sicht (d) zeigt die entsprechende Sicht zu Sicht (c) unter Verwendung von *Deutsch* als Sprachkontext. Die jeweils äquivalenten interaktiven Komponenten übermitteln hierbei die deutschen Bezeichner zur Visualisierung der Einstiegspunkte der jeweiligen Komponente.

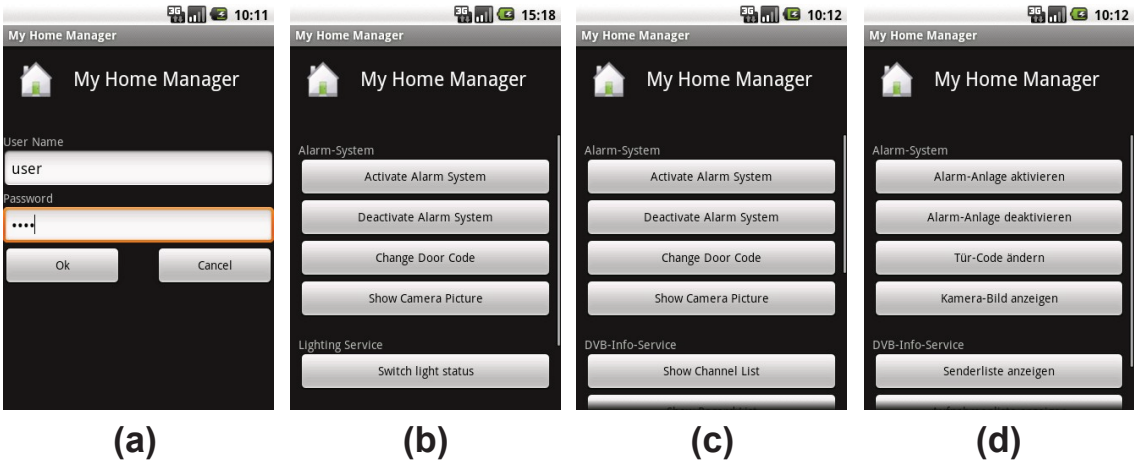

Abbildung 5.3.: My Home Manager - Authentifizierung und Übersicht der Komponenten

Im Folgenden soll die interaktive Komponente des *DVBInfo-Services* näher betrachtet werden. Sicht (a) der Abbildung 5.4 veranschaulicht die Ausgabe der interaktiven Komponenten nach dem Aufruf der Option *Show Record List* in der Hauptansicht der Zielanwendung. In dieser Sicht werden die verfügbaren Aufnahmen der *DVBInfo-Services* in einer Liste angezeigt. Zur Visualisierung der Einträge dienen die Informationen der *HumanReadableString*-Annotation (vgl. [Mar10], Seite 32), mit welcher die Rückgabe der Operation *getRecordList* annotiert wurde. Die Steuerelemente *New Record*, *Remove Record* und *Show Preview* resultieren aus der *NavigationChoice*-Annotation, vgl. [Mar10] Seite 28, wobei *Remove Record* und *Show Preview* einen Datenfluss auf die nächste Seite enthalten und *New*

Record ein reiner Navigationslink ist, und deshalb gesondert visualisiert wird. Sicht (b) der Abbildung 5.4 veranschaulicht die Eingabe einer neuen Aufnahme. Hierbei wurde für die Eingabe der Aufnahmezeit ein spezieller Interaktor eingefügt, welcher in einem *DateTimePicker* resultiert. Da in der gezeigten Darstellung noch nicht alle Eingabefelder ausgefüllt sind, deren Eingabeparameter aber mit der ServFace Annotation *MandatoryField* annotiert wurden (vgl. [Ser09], Seite 37), ist das Steuerelement zum Anlegen der neuen Aufnahme noch nicht aktiviert. Nach erfolgreichem Anlegen einer neuen Aufnahme wird diese in der Liste der angelegten Aufnahmen mit angezeigt, was durch Sicht (c) der Abbildung 5.4 veranschaulicht wird. Sicht (d) der Abbildung 5.4 zeigt die Sicht, welche über das *Show Preview*-Steuerelement erreicht wird und visualisiert eine Vorschau einer angelegten Aufnahme. Hierbei wurde die Rückgabe der *getPreview*-Annotation mit der *CustomWidget*-Annotation (vgl. [Mar10], Seite 31) annotiert, welche durch den Inferenz-Mechanismus von [Fel11] als Interaktor vom Typ *CustomWidget* abgeleitet wird, der aber eine Referenz auf ein Xpand-Template enthält, welches dann für die Generierung dieses Widgets sorgt.

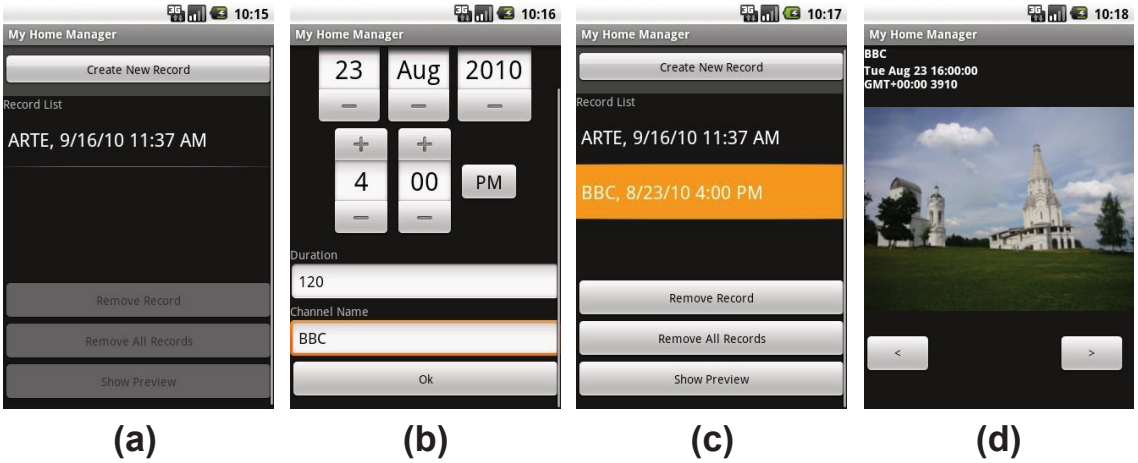

Abbildung 5.4.: My Home Manager - Interaktive Komponente des DVBInfo-Services

5.2. Evaluierung anhand eines weiteren Szenarios

Neben dem in dieser Arbeit verwendeten Szenario zur Heimautomatisierung, wurde das System anhand eines zusätzlichen Szenarios, welches im Rahmen der Arbeit von [Fel11] entstanden ist, evaluiert. Die entstandene Service-Infrastruktur und die daraus resultierende Anwendung soll die Arbeit eines Landarztes in stark entlegenen Regionen unterstützen. Dabei ist das Erfassen medizinischer Patientendaten und das Bestellen von Medikamenten bzw. medizinischen Verbrauchsgütern möglich. Sicht (a) der Abbildung 5.5 zeigt die Hauptansicht der entstandenen Anwendung. Diese Zielanwendung unterscheidet sich nur geringfügig von der des Heimautomatisierungsszenarios. Konkret musste nur der Zielkontext, die Konfiguration des Komponenten-Repositories und die Rahmenelemente der Hauptansicht angepasst werden. Die Server-Komponente zum Veröffentlichen der interaktiven

Komponenten für das Landarztszenario musste dabei nur mit den dem Szenario entsprechenden Parametern gestartet werden. Eine Besonderheit dieses Szenarios gegenüber dem Heimautomatisierungsszenario ist, dass im Patientenverwaltungsdienst ein Kontext vom Typ *Location* verwendet wurde. Hierbei kann die interaktive Komponente zur Patientenverwaltung im mobilen Einsatz für den Hausbesuch und zum Einsatz in der Praxis des Landarztes abgeleitet werden. Im Detail wurde die Operation zum Anlegen bzw. Aktualisieren eines Patienten so annotiert, dass die Ortsinformation eines Patienten bei einem Hausbesuch direkt vom Gerät ermittelt und beim Einsatz in der Praxis nicht verändert wird. Erreicht wurde dies mit den Annotationen *PlatformProvidedEntitiy* (vgl. [Mar10], Seite 35) und *DisabledParameter* (vgl. [Mar10], Seite 34). Die *PlatformProvidedEntitiy*-Annotation führt dazu, dass kein Eingabefeld für die jeweilige Eingabe inferiert wird. Der Wert für diesen Parameter ermittelt die interaktive Komponente mit Hilfe einer nachinstallierten Plattformfunktionalität, dessen Handle von der Zielanwendung bereitgestellt wird. Die *DisabledParameter*-Annotation führt dazu, dass ebenfalls kein Eingabefeld inferiert wird, der Wert des Parameter bei einer Aktualisierung unverändert bleibt bzw. bei einem Anlegen nicht initialisiert wird. Sicht (b) der Abbildung 5.5 zeigt die Ansicht beim Anlegen eines Patienten bei einem Hausbesuch. Das Resultat des Anlegens kann in Tabelle 5.1 betrachtet werden. Die Ortsinformation wurde hierbei direkt vom Gerät bezogen. Sicht (c) der Abbildung 5.5 zeigt die Ansicht der Aktualisierung desselben Patienten in der Praxis des Landarztes. Das Resultat der Aktualisierung kann in Tabelle 5.2 betrachtet werden. Wie zu erkennen ist, wurde die Ortsinformation des Patienten unverändert übernommen, das Gewicht hingegen wurde aktualisiert.

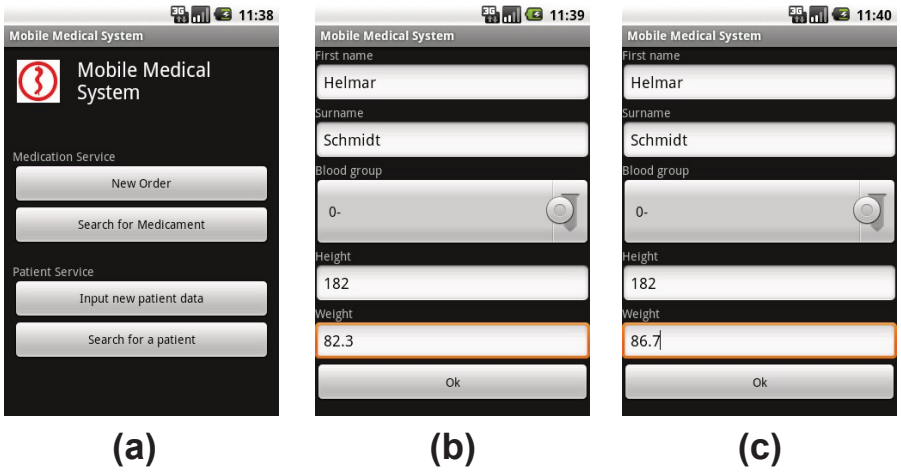

Abbildung 5.5.: Mobile Medical System

ID	Name	Surname	Bloodgr.	Height	Weight	Longitude	Latitude
0	Bruno	Berndt	B+	86.0	10.3	14.73307	51.03957
1	Johanna	Meier	AB-	176.5	68.6	12.45071	52.15557
2	Helmar	Schmidt	0-	182.5	82.3	13.54507	50.08852

Tabelle 5.1.: Patientendaten nach Anlegen eines Patienten (ID=2) bei einem Hausbesuch

ID	Name	Surname	Bloodgr.	Height	Weight	Longitude	Latitude
0	Bruno	Berndt	B+	86.0	10.3	14.73307	51.03957
1	Johanna	Meier	AB-	176.5	68.6	12.45071	52.15557
2	Helmar	Schmidt	0-	182.5	86.7	13.54507	50.08852

Tabelle 5.2.: Patientendaten nach Aktualisierung eines Patienten (ID=2) in der Praxis

5.3. Evaluierung der Benutzbarkeit

Um die Benutzbarkeit der generierten interaktiven Komponenten nachweisen zu können bzw. diese bewerten zu können und daraufhin Entscheidungen zur Verbesserung des Generierungsverfahren treffen zu können, wurde eine Usability-Studie durchgeführt. Die durchgeführte Studie gliedert sich in zwei Teile. Der erste Teil wurde unter Verwendung der Zielanwendung des Landarzt-Szenarios von [Fel11] durchgeführt, im zweiten Teil wurde die Zielanwendung des Heimautomatisierungsszenarios verwendet. Jeder Teil beinhaltet anfänglich einige Fragen und einzelne Aufgaben, die von den Probanden beantwortet bzw. gelöst werden müssen, wobei zur Lösung die jeweilige Zielanwendung bzw. die integrierten Komponenten verwendet werden müssen. Zum Beispiel lautet eine solche Frage des ersten Teils: *„What is the PZN-Number of the medicament Buscopan Plus?“*, und eine Aufgabe: *Update the weight of Mr. Schulz to 83.6.*. Es wird so gewährleistet, dass alle Probanden die Anwendung kennenlernen und möglichst alle Bereiche der Anwendung betreten bzw. verwendet haben. Im Anschluss daran findet dann die eigentliche Evaluation in Form eines Fragebogens statt. Der zweite Teil ist analog zum ersten Teil aufgebaut, wobei der Fragebogen des zweiten Teils Aspekte der Zielanwendung und die Integration der interaktiven Komponenten adressiert, wohingegen der Fragebogen des ersten Teils Aspekte der interaktiven Komponenten adressiert. In der Aufgabenstellung beider Teile wird darauf hingewiesen, dass bei Problemen beim Lösen der anfänglichen Fragen und Aufgaben, diese in wenigen Sätzen notiert werden sollen. Die gewonnenen Informationen sind neben der Auswertung der Fragebögen ein wichtiges Mittel zur Verbesserung der Anwendung. Am Ende der gesamten Usability-Studie schließen sich noch drei Fragen an, welche in freier Textform beantwortet werden sollen, in denen die Probanden ein allgemeines Feedback geben sollen. Die Usability-Studie ist im Anhang A.1 zu finden.

Bei der Evaluierung der Benutzbarkeit wurde iterativ vorgegangen, wobei die Studie im Laufe dieser Arbeit insgesamt zweimal durchgeführt wurde. An der ersten Studie hat eine Gruppe von acht Probanden teilgenommen. An der zweiten Studie hat eine Gruppe von fünf Probanden teilgenommen. Die Erkenntnisse der ersten Studie wurden zur Weiterentwicklung und Verbesserung des Systems genutzt. Die zweite Studie, fortlaufend als Kontrollstudie bezeichnet, dient als Kontrolle der einzelnen Verbesserung, welche am System durchgeführt wurden. Im Gegensatz zur Kontrollstudie haben alle Probanden der ersten Studie im Vorfeld an einer Usability-Studie des ICEs teilgenommen, wobei die Aufgabe darin bestand die Anwendung des Landsarzt-Szenarios zu modellieren und im Anschluss das Programm bezüglich der Benutzbarkeit zu bewerten, siehe [Mar10], Kapitel 6.2. Im di-

rekten Anschluss wurde die erste Usability-Studie durchgeführt, welche Bestandteil dieser Arbeit ist.

Die erste Fragengruppe des Teil 1 der Studie zielt auf die Erwartungshaltung gegenüber der modellierten Anwendung des Landarzt-Szenarios ab. Diese Fragengruppe war nur Bestandteil der ersten Studie, deren Ergebnisse in Abbildung 5.6 veranschaulicht sind.

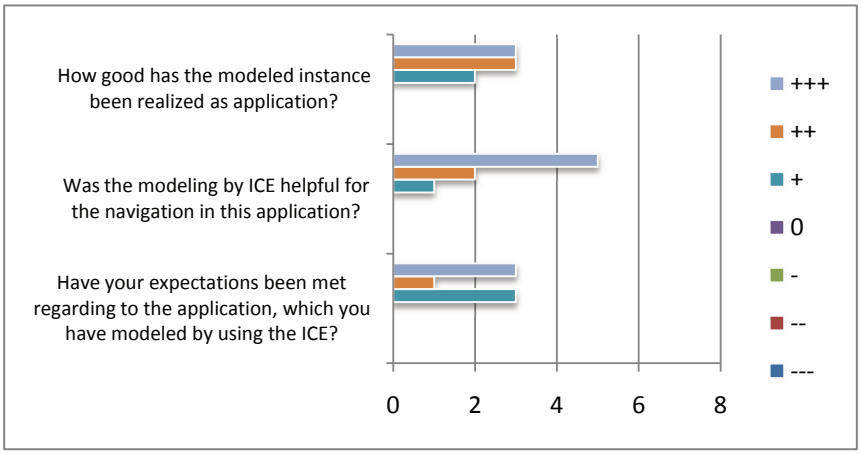

Abbildung 5.6.: Erwartungshaltung gegenüber der im ICE modellierten Anwendung

Alle Probanden haben überwiegend angegeben, dass die Erwartungen gegenüber der modellierten Anwendung des ICEs erfüllt worden sind. Die letzte Frage stellt hierbei eine Kontrolle der ersten Frage dar, da für denselben Sachverhalt eine andere Formulierung gewählt worden ist. Ein Proband hat bei der letzten Frage keine Angaben gemacht, die anderen Probanden bestätigen im Wesentlichen die Angaben der ersten Frage.

Wie bereits beschrieben, führten die Ergebnisse der ersten Studie zu einem Erkenntnisgewinn, welcher maßgeblich zur Verbesserung des entstandenen Systems geführt hat. Eine Auswahl der signifikantesten Unterschiede bezüglich der Bewertung ist in Abbildung 5.7 dargestellt, wobei die Anzahl der Teilnehmer der jeweiligen Studien normiert wurde, um diese miteinander vergleichen zu können.

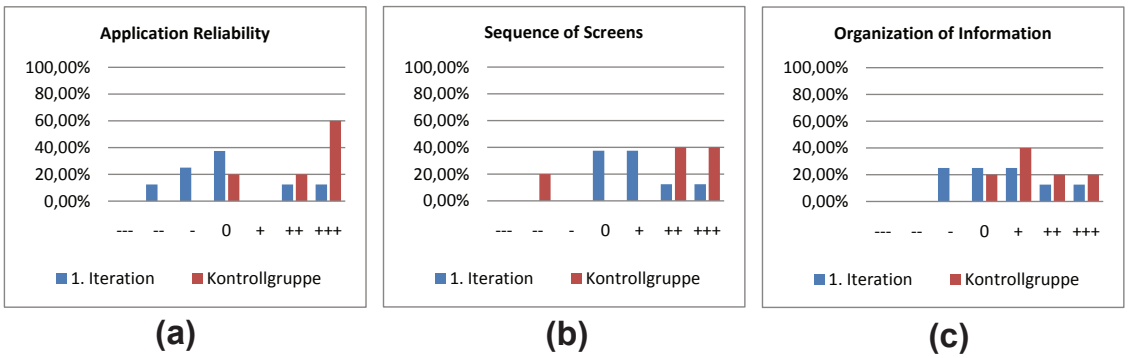

Abbildung 5.7.: Unterschiede zwischen der 1. Studie und der Kontrollstudie

Wie zu erkennen ist, bewertet die Kontrollgruppe das entstandene System wesentlich besser als die Probanden der ersten Usability-Studie. Allgemein kann man sagen, dass alle Fragen von der Kontrollgruppe besser oder mindestens gleich gut bewertet wurden. Diagramm (a) der Abbildung 5.7 stellt das Ergebnis zur Frage der Betriebszuverlässigkeit des entstanden Systems dar, wobei die Kontrollgruppe dieses als sehr zuverlässig bewertet, wohingegen in der ersten Evaluation das System als nicht sehr zuverlässig bewertet wurde. Eine Schwachstelle konnte in der Service-Umgebung des Heimautomatisierungsszenarios gefunden und beseitigt werden. In Folge eines zu kurz gewählten Session-Timeouts, lieferten einige Serivce-Operationen bei einer ungünstigen Reihenfolge der Nutzung der einzelnen interaktiven Komponenten keine nutzbaren Werte zurück. Neben dieser Änderung wurden auch weitere kleinere Verbesserungen durchgeführt. Durch Änderungen einiger Annotation, welche einen Navigations- und Datenfluss beschreiben und einer dazugehörigen Erweiterung des Generierungsverfahrens seitens [Fel11] konnte die Sequenz der einzelnen Ansichten verbessert werden. Konkret wurden überflüssige Eingabesichten entfernt, welche aus einem Datenfluss resultierten, indem direkt zur jeweiligen Ausgabesicht navigiert wird. Diese Verbesserung erklärt zum einen die Unterschiede im Diagramm (b) der Abbildung 5.7. Zum anderen muss erwähnt werden, dass der größte Teil der Teilnehmer der Kontrollgruppe Erfahrungen im Bereich mobiler dienstbasierter Anwendungen besitzt, was wahrscheinlich maßgeblich zu einem so großen Unterschied geführt hat. Neben der Bewertung der einzelnen Fragen wurden wie bereits erwähnt ebenfalls die Antworten der Probanden, welche in freier Textform formuliert wurden, ausgewertet. Teilweise wurde hier im Detail auf eine bestimmte Ansicht einer interaktiven Komponente eingegangen, teilweise wurden hingegen generelle Verbeserungswünsche notiert. Die Auswertung der frei formulierten Antworten der ersten Usability-Studie hat vor allem dazu geführt, dass alle Dienste der jeweiligen Szenarien um zusätzliche Annotationen erweitert bzw. vervollständigt wurden. Dies wird im Wesentlichen der Grund für eine deutlich bessere Bewertung der Organisation der Information durch die Kontrollgruppe gegenüber der Gruppe der ersten Studie sein. Die Ergebnisse zur Bewertung der Organisation der Information wird in Diagramm (c) der Abbildung 5.7 veranschaulicht.

Als letzten ausgewählten Punkt der Ergebnisse der durchgeführten Studien soll im Folgenden die Integration der interaktiven Komponenten in die Zielanwendung betrachten werden. In beiden durchgeführten Studien sollten die Teilnehmer die Einstellungen für das Komponenten-Repository vornehmen, auf neu verfügbare Komponenten prüfen und diese installieren. Zusätzlich wurde gefragt, wie gut die Integration der einzelnen interaktiven Komponenten in die Zielanwendung realisiert wurde. Die Ergebnisse der Befragung zur Integration werden in den Diagrammen der Abbildung 5.8 veranschaulicht, wobei die Anzahl der Teilnehmer normiert wurde.

Wie in den zuvor präsentierten Ergebnissen, bewertet die Kontrollgruppe vor allem die Konfiguration der Komponenten-Repositories (siehe Abbildung 5.8 (a)) und das Update der interaktiven Komponenten (siehe Abbildung 5.8 (b)) deutlich besser als die Teilnehmer der ersten Usability-Studie. Hierbei sei angemerkt, dass an den Mechanismen zum Update und zur Integration der Komponenten nichts verändert wurde und somit die zu

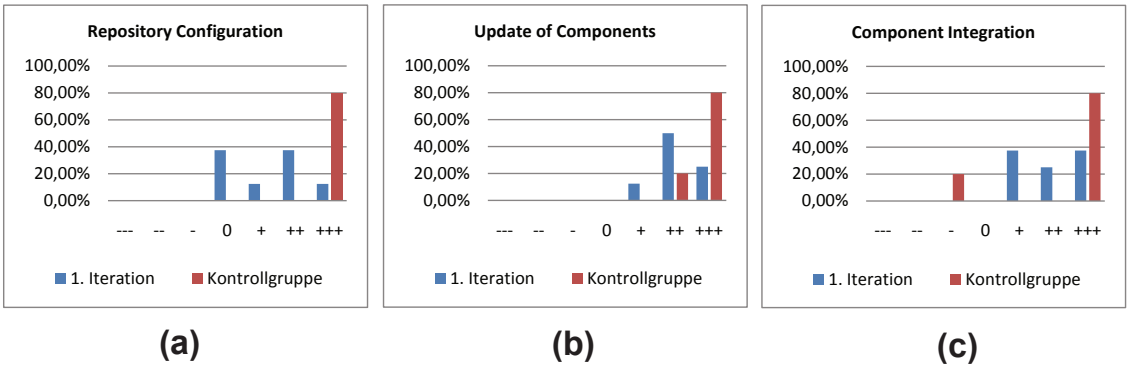

Abbildung 5.8.: Auswertung zur Realisierung der Integration interaktiver Komponenten

bewertenden Teile des Systems zum Zeitpunkt beider Studien auf demselben Entwicklungsstand waren. Die bessere Bewertung der Kontrollgruppe, könnte sich wieder dadurch erklären, dass der größte Teil dieser Gruppe Erfahrung im Umgang mobiler Anwendungen bzw. Android-Anwendungen hat. Insgesamt wird von den Teilnehmern beider Studien die Realisierung der Integration als gut bewertet (siehe Abbildung 5.8 (c)). Das bedeutet, dass die Anwendung als Ganzes wahrgenommen wurde und sich die integrierten interaktiven Komponenten nahtlos in die Anwendung einfügen ohne einen zu großen Bruch im UI zu verursachen. In durchgeführten Tiefeninterviews zur Konsistenz des gesamten UIs konnte diese Aussage nochmals bestätigt werden. Der Grund für die Konsistenz der UIs der einzelnen integrierten Komponenten beruht auf dem einheitlichen Generierungsverfahren, durch welches jeder dieser Komponenten entstanden ist. Nur ein Teilnehmer der Kontrollgruppe hat die Integration schlecht bewertet. Aus den frei formulierten Antworten des Teilnehmers konnte die Schlussfolgerung gezogen werden, dass dieser Teilnehmer generelle Probleme mit der Bedienung des Gerätes hatte und somit keine Erfahrungen mit Android-Anwendungen besitzt.

Abschließend kann gesagt werden, dass die Ergebnisse der ersten Usability-Studie maßgeblich zur Verbesserung des gesamten Systems beigetragen haben. In beiden Studien wurde bestätigt, dass die verschiedenen, in eine Zielanwendung integrierten, interaktiven Komponenten als eine einheitliche Anwendung verstanden werden und die Integration als gelungen bewertet wurde. Die Ergebnisse der Kontrollstudie können zusätzlich noch genutzt werden, um das System noch weiter zu verbessern. Es hat sich z.B. herausgestellt, dass viele den Zurück-Button, den alle Android-Geräte als Hardkey besitzen, nicht nutzen, sondern ein in der Benutzerschnittstelle visualisierten Zurück-Button bevorzugen würden bzw. diesen gesucht haben.

5.4. Evaluierung des Gesamtkonzepts

Wie bereits in der Einleitung beschrieben, bildet das Resultat dieser Arbeit das letzte Glied in der Prozesskette zur Generierung interaktiver Komponenten aus annotierten funktio-

nalen Dienstschnittstellenbeschreibungen (siehe 1.2), in Form eines konkreten Plattformadapters für *Android*. In diesem Abschnitt soll der Gesamtansatz bzw. die Eingliederung der entstandenen Resultate dieser Arbeit in den Gesamtansatz evaluiert werden. Abbildung 5.9 gibt einen Überblick über die im Rahmen des Gesamtansatzes entwickelten Systemkomponenten.

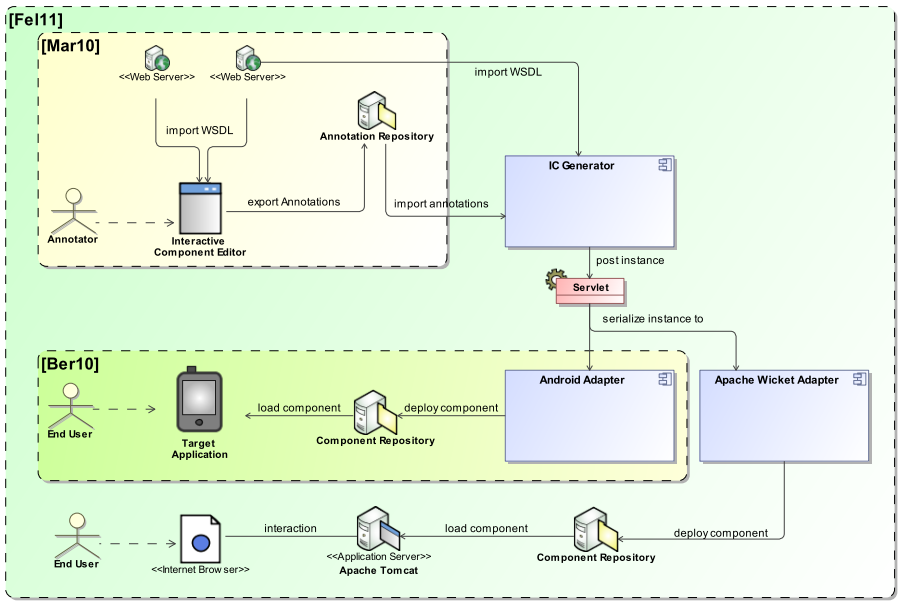

Abbildung 5.9.: Überblick über das Gesamtsystem

Zu Beginn der Prozesskette des Gesamtansatzes steht das Desingtime-Werkzeug zum visuellen Erstellen von Annotation zur Beschreibung von interaktiven dienstbasierten Anwendungen, welche die Grundlage für die Generierung von interaktiven Komponenten bilden. Dieses Werkzeug ist in Form des ICE im Rahmen der Arbeit von [Mar10] als Eclipse Rich Client Platform (RCP)-Anwendung entstanden. Zu Beginn der Anwendungsmodellierung wird im ICE zunächst eine neue Anwendung angelegt, indem eine initiale funktionale Dienstbeschreibung in Form einer WSDL-Datei geladen wird, wobei zu späteren Zeitpunkten weitere funktionale Dienstbeschreibungen importiert werden können. Nachdem die neue Anwendung angelegt wurde, wird in der Dienstübersicht der importierte Dienst mit den verfügbaren Dienstoperationen in einer Baumansicht dargestellt. Über das Kontextmenü können in den nächsten Schritten Anwendungsseiten für die verfügbaren Dienstoperationen angelegt werden. Für diese Schritte bietet sich die Arbeit im *Overview-Editor* des ICE an, welcher in Abbildung 5.10 veranschaulicht wird.

Das Hinzufügen von Annotationen geschieht dann im *Navigation-Editor* des ICEs, wobei immer zwei Anwendungsseiten gegenüber gestellt werden. Zur visuellen Modellierung von Navigations- bzw. Datenflüssen bietet der *Navigation-Editor* geeignete Werkzeuge. Zusätzlich kann die Anwendung mit weiteren Annotationen, welche im Rahmen der Arbeit von [Mar10] diskutiert werden, und einer Teilmenge von ServFace-Annotation versehen werden. Abbildung 5.11 veranschaulicht den *Navigation-Editor* des ICE.

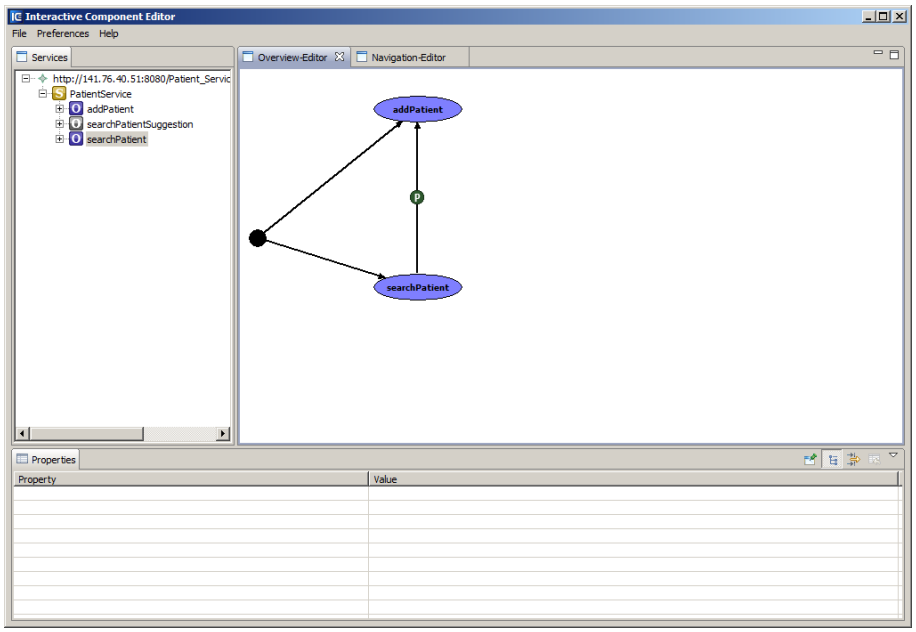

Abbildung 5.10.: ICE - Overview-Editor

Zum Abschluss wird die modellierte Anwendung in Form einer Annotationsdatei in einem *Annotation-Repository* hinterlegt. Die nächsten Schritte, bis hin zur lauffähigen Anwendung, können ab diesem Moment vollautomatisch geschehen.

In der zweiten großen Phase des Gesamtansatzes, in Abbildung 5.9 mit [Fel11] gekennzeichnet, dient die Anwendungsbeschreibung des *Annotation-Repositories* dem *IC Generator* als Grundlage zur Generierung einer Instanz, zur Beschreibung einer interaktiven Komponente in Form einer SBIC-Instanz. Hierbei werden mehrere M2M-Transformation durchgeführt, wobei die Paginierung und das Ableiten der konkreten Interaktoren für verschiedene Zielplattformen, nach speziell für die Plattform definierten Regeln erfolgt. Die plattformspezifischen SBIC-Instanzen werden dann dem jeweiligen Plattformadapter zur Verfügung gestellt.

Die Plattformadapter überführen die jeweils plattformspezifische SBIC-Instanz in eine ausführbare interaktive Komponente, welche auf der jeweiligen Plattform integriert werden kann. Im Rahmen dieser Arbeit wurde ein konkreter Plattformadapter für *Android* realisiert, dessen Arbeitsweise am Beispiel in Abschnitt 5.1 bereits vorgestellt wurde. Neben diesem Plattformadapter, wurde im Rahmen der Arbeit [Fel11] prototypisch ein Adapter umgesetzt, der interaktive Komponenten in Form von *Apache Wicket*-Komponenten[17] generiert. Die generierten interaktiven Komponenten können in eine Zielanwendung integriert werden, welche dem Endnutzer als Web-Anwendung zur Verfügung gestellt wird.

Die Integration des entstanden Plattformadapters für *Android* in den Gesamtansatz wurde in einem Praxistest nachgewiesen. Hierbei wurde die Anwendung zur Patientenverwaltung (siehe 5.2) unter Verwendung des ICEs modelliert und durch die anschließenden Gene-

[17] http://wicket.apache.org/

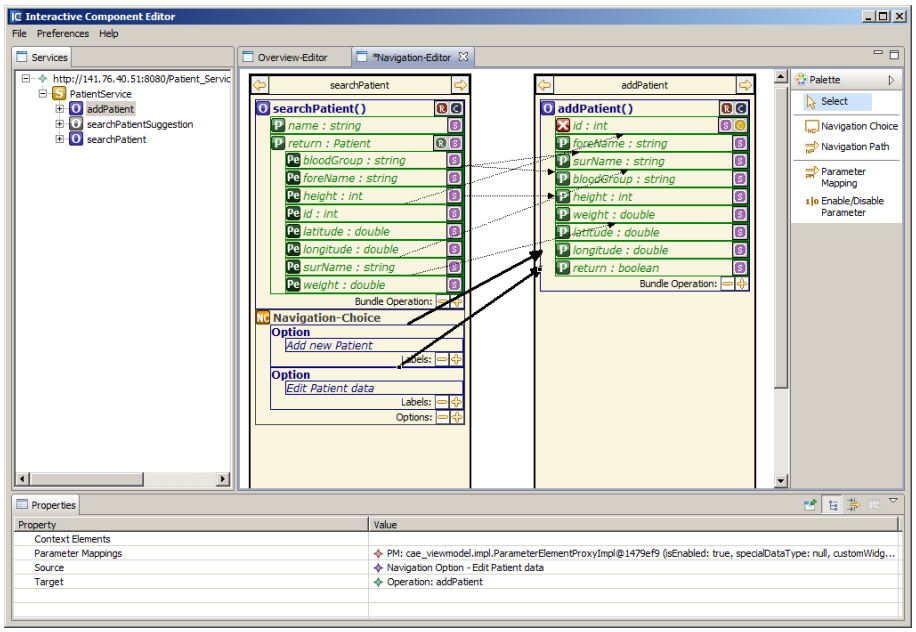

Abbildung 5.11.: ICE - Navigation-Editor

rierungsschritte in eine lauffähige interaktive Komponente überführt. Das Resultat genau dieses Tests wurde bereits in Abbildung 5.5 präsentiert.

5.5. Überprüfung der Anforderungen

In diesem Abschnitt wird das entstandene System zur Generierung interaktiver Komponenten und die damit verbundenen Zielanwendungen anhand der aufgestellten Anforderungen des Analysekapitels (siehe 2.2) evaluiert. Dabei wird die eingeführte Klassifizierung der Anforderungen in die Basisanforderungen, die Anforderungen, die sich an das allgemeine Generierungsverfahren richten, die Anforderungen bezüglich der Distribution der generierter interaktiver Komponenten, die Anforderungen, welche die Zielanwendung betreffen und die übergeordneten Anforderungen beibehalten.

I. Basisanforderungen

Die Basisanforderungen, welche direkt aus der Themenstellung dieser Arbeit resultieren, wurden durch die Verwendung von *OSGi* zur Entwicklung und Integration interaktiver Komponenten und durch die Verwendung von *Android* als Zielplattform für die Zielanwendung erfüllt.

II. Anforderungen an das Generierungsverfahren

/AG01/ M2C-Transformation zur Generierung von OSGi-Komponenten

Die entwickelte M2C-Transformation der *Workflow-Engine* generiert eine Code-Repräsentation interaktiver Komponenten aus SBIC-Instanzen. Diese Code-Repräsentation wird in einem Build- und Paketierungsprozess, der Bestandteil der *Workflow-Engine* ist, in OSGi-Bundles überführt, welche zusammen eine interaktive Komponente repräsentieren.

/AG02/ Erweiterbarkeit der M2C-Transformation

Die M2C-Transformation kann um Templates erweitert werden, die aufgrund von Informationen der Modell-Instanz während der Generierung dynamisch aufgerufen werden. Hierbei werden für einzelne Interaktoren, wie zum Beispiel der Interaktor der Rückgabe der *getPreview*-Operation des *DVBInfo-Services*, spezielle Widgets generiert. Bedingung hierbei ist, dass die Templates der Server-Komponente im Vorfeld der Generierung bekannt gemacht werden.

/AG03/ Automatisierung der Generierung

Die Automatisierung der Generierung wird durch die *Server-Komponente* realisiert, indem diese ein Verzeichnis überwacht und bei Vorhandensein von SBIC-Instanzen in diesem Verzeichnis für jede Instanz die M2C-Transformation und den anschließenden Build- und Paketierungsprozess startet. Abschließend wird die generierte, interaktive Komponente automatisch in dem im Vorfeld definierten Komponenten-Repository veröffentlicht, wie Abschnitt 5.1 verdeutlicht hat.

/AG04/ Generalität

Die Generierung von interaktiven Komponenten ist für beliebige SBIC-Instanzen möglich, die eine definierte Menge von Interaktoren und Annotationen verwenden.

/AG05/ Benutzbarkeit der generierten Komponenten

Das Generierungsverfahren sorgt dafür, dass eine reichhaltige Benutzerschnittstelle der interaktiven Komponenten generiert wird. Diese Benutzbarkeit wurde anhand zweier durchgeführter Usability-Studien nachgewiesen, siehe 5.3.

III. Anforderungen an die Distribution interaktiver Komponenten

/AD01/ Mechanismus zur Veröffentlichung und Verbreitung der Komponenten

Die generierten Komponenten werden automatisch, nachdem sie generiert wurden, in einem im Vorfeld definierten zentralen Repository veröffentlicht. Die Zielanwendung, der dieses Repository bekannt gemacht werden muss, kann die veröffentlichten interaktiven Komponenten aus diesem Repository laden und installieren.

/AD02/ Versionierung veröffentlichter Komponenten

Jede veröffentlichte interaktive Komponente des Komponenten-Repositories besitzt eine eindeutige Revisionsnummer. Diese Revisionsnummer kann wiederum genau einer Version der veröffentlichten Komponente zugewiesen werden. Durch die Verwendung eines SVN-Repositories, wobei WebDAV als Übertragungsprotokoll eingesetzt wird, ist es der Zielanwendung möglich, die Revisionsnummern der veröffentlichten interaktiven Komponente in Erfahrung zu bringen. Die *Repository-Komponente* der Zielanwendung übernimmt hierbei die lokale Versionsverwaltung.

/AD03/ Vertrauenswürdigkeit

Die Vertrauenswürdigkeit gegenüber den Klienten des Komponenten-Repositories, wird über einen Authentifizierungsmechanismus, bestehend aus Nutzername und Passwort, erreicht. Der Mechanismus zur Authentifizierung ist Bestandteil der SVN-Umgebung. Sowohl die Server-Komponente beim Veröffentlichen der interaktiven Komponenten, als auch die Zielanwendung beim Laden der Komponenten müssen sich zuvor beim Komponenten-Repository authentifizieren.

Die Vertrauenswürdigkeit gegenüber dem Repository-Server wird durch die Verwendung von HTTPS zur Kommunikation mit dem Server erreicht, wobei nur Verbindungen zu vertrauenswürdigen Quellen akzeptiert werden.

IV. Anforderungen an die Zielanwendung

/AZ01/ Integration der Komponenten

Die Zielanwendung gewährleistet eine transparente Integration generierter interaktiver Komponenten. *Transparent* bedeutet hierbei, dass sich die generierten Komponenten nahtlos in die Zielanwendung einfügen und die Integration völlig automatisch abläuft. Die

transparente Integration wurde anhand der durchgeführten Usability-Studien bestätigt, siehe 5.3.

/AZ02/ Registrierung der Komponenten

Die Registrierung der interaktiven Komponenten wird mittels der *Component-Registry* der *Core-Komponente* der Zielanwendung realisiert. Der implementierte Registrierungsmechanismus sorgt dafür, dass eine interaktive Komponente nur dann integriert wird, wenn geforderte Eingabeparameter und geforderte Plattformfunktionalitäten in der Zielanwendung verfügbar sind. Visualisiert wird eine interaktive Komponente erst, wenn der geforderte Zielkontext, dem des aktuell eingestellten Zielkontextes der Zielanwendung entspricht.

/AZ03/ Bereitstellung von zusätzlichen Plattformfunktionalitäten

Die Zielanwendung ermittelt während der Registrierung einer interaktiven Komponente, ob diese für den Start zusätzliche Plattformfunktionalitäten benötigt. Ist dies der Fall, werden diese bei Nichtvorhandensein nachinstalliert und der interaktiven Komponente beim Start übermittelt. Bei Vorhandensein wird das Handle direkt übermittelt. Die Verwaltung der verfügbaren Plattformfunktionalitäten übernimmt hierbei der *PlatformFunctionality-Manger* der *Dependency-Komponente* der Zielanwendung, welcher zugleich das Handle mit den geforderten Funktionalitäten einer interaktiven Komponente erzeugt. Die Informationen zur Nachinstallation erhält die Zielanwendung aus den Registrierungsdaten der jeweiligen interaktiven Komponente.

/AZ04/ Ansteuerung der Komponenten

Zum Ansteuern, Starten und Stoppen einer interaktiven Komponente wurde das Protokoll umgesetzt, welches Bestandteil der Arbeit von [Fel11] ist. Dieses Protokoll beschreibt die Kommunikation zwischen Zielanwendung und einer interaktiven Komponente. Teil dieses Protokolls sind zum einen die Registrierungsdaten einer interaktiven Komponente und die Schnittstelle zum Starten und Stoppen der Komponente, welche mit dem *InteractiveComponentFrontend* realisiert wurde.

/AZ05/ Kontextadaptierbarkeit

Kontextadaptierbarkeit einer Zielanwendung wird erreicht, indem der konkreten Zielanwendung zusätzliche *Kontext-Provider* hinzugefügt bzw. eigene *Data-Provider* definiert werden können, welche ein Data-Entity in der Zielanwendung global verfügbar machen. Eine Zielanwendung lässt sich so an einen spezielles Anwendungsszenario anpassen. Zum Beispiel wurde die Authentifizierung der Zielanwendung des Heimautomatisierungsszenarios mit einem *Data-Provider* realisiert. In der Zielanwendung zur Unterstützung eines Landarztes wurden spezielle *Kontext-Provider* definiert, die den geforderten Kontexten des Szenarios entsprechen.

V. Übergeordnete Anforderungen

Die übergeordneten Anforderungen, welche aus der Eingliederung dieser Arbeit in den Gesamtansatz (siehe 1.2) resultieren, wurden durch die Integration des Systems zur Generierung von interaktiven Komponenten in die Prozesskette des Gesamtansatzes in Form eines konkreten Plattformadapters für Android erfüllt, wie im Abschnitt 5.4 verdeutlicht wurde.

6. Zusammenfassung und Ausblick

Das Ziel der vorliegenden Arbeit war es, ein System zur vollautomatischen Generierung und Distribution von interaktiven dienstbasierten Komponenten in Form von OSGi-Bundles zur Integration in eine Zielanwendung, welche als Android-Anwendung realisiert werden sollte, zu erstellen. Die Motivation bestand darin, dass Anwendungen welche in Form von Dienstannotationen beschrieben werden, wobei insbesondere eine Beschreibung von Navigations- und Datenflüssen möglich ist, für einen vollautomatischen Generierungsprozess verwendet werden können, um eine Instanz eines Meta-Modells zur Beschreibung interaktiver dienstbasierter Komponenten zu erzeugen. Diese Instanzen bilden den Ausgangspunkt für das in dieser Arbeit entwickelte Verfahren zur Generierung von interaktiven Komponenten. Diese Arbeit stellt hierbei einen möglichen letzten Teil des Gesamtkonzeptes, welches in Kapitel 1.2 vorgestellt wurde, in Form eines Plattformadapters für Android dar. Die diskutierte Anwendungsbeschreibung in Form von Annotationen wird hierbei durch das in [Mar10] entstandene Designtime-Werkzeug visuell unterstützt. Die anschließende Generierung einer Modell-Instanz, auf Basis der Anwendungsbeschreibung, zur Beschreibung einer interaktiven Komponente, wird im Rahmen des Gesamtansatzes durch das Generierungsverfahren, welches in [Fel11] entstanden ist, gewährleistet.

Zunächst wurde in Kapitel 2 ein geeignetes Anwendungsszenario beschrieben, welches als Referenz für die anschließende Beschreibung der Anforderungen genutzt wurde. In Kapitel 3 wurden zunächst die notwendigen Grundlagen zur Konzeption eines Systems, zur automatischen Generierung interaktiver Komponenten in Form von OSGi-Bundles, erläutert, welche in eine Zielanwendung in Form einer Android-Anwendung integriert werden. Daraufhin wurden verwandte Ansätze auf dem Gebiet der automatischen Generierung interaktiver dienstbasierter Anwendungen bzw. Ansätze auf dem Gebiet der Integration von UI-Komponenten in einen gemeinsamen Anwendungskontext betrachtet, deren Charakteristika ermittelt und abschließend mit denen dieser Arbeit verglichen. Hierbei wurde ermittelt, dass kein Ansatz zur vollautomatischen Generierung von interaktiven Komponenten zur Integration in einen gemeinsamen Anwendungskontext existiert. Basierend auf den Grundlagen und den formulierten Anforderungen stellt Kapitel 4 das Konzept und die Umsetzung des Systems zur Generierung interaktiver Komponenten, das Konzept und die Umsetzung einer interaktiven Komponente an sich und das Konzept und die Umsetzung einer Zielanwendung zur Integration interaktiver Komponenten vor. Die Umsetzung des aufgestellten Konzepts wurde in Kapitel 5 validiert. Dabei wurde das System anhand des beschriebenen Anwendungsszenarios, eines weiteren Szenarios, einer Usability-Studie in zwei Iterationen und hinsichtlich der Integration in den Gesamtansatz (siehe 1.2) bewer-

tet. Diese Bewertungen bildeten die Grundlage für eine abschließende Überprüfung der aufgestellten Anforderungen.

Die Ergebnisse dieser Arbeit zeigen, dass Instanzen eines Meta-Modells zur Beschreibung interaktiver Anwendungen zur Generierung interaktiver Komponenten genutzt werden können, welche dynamisch und transparent in eine Zielanwendung integriert werden. Die Zielanwendung setzt sich so aus einer Menge von interaktiven Komponenten zu einer komplexen Anwendung zusammen, wobei der Eindruck einer einheitlichen Anwendung entsteht, was in Kapitel 5.3 nachgewiesen wurde.

Die Resultate der in der zweiten Iteration der Usability-Studie könnten in einem weiten Vorgehen dazu verwendet werden, das Generierungsverfahren bzw. die entstandene Zielanwendung hinsichtlich Strukturierung der Navigation zu verbessern. Man könnte zur Schonung der begrenzten Ressourcen eines mobilen Endgerätes, einen zentralen OSGi-Container für alle Zielanwendungen eines Gerätes verwenden, welcher auf OSGi-Ebene die installierten, interaktiven Komponenten aller Zielanwendungen verwaltet..

Neben den Verbesserungen der Architektur der Anwendung, könnte das Konzept der Zielanwendung bzw. der integrierten Komponenten erweitert werden. Zum gegenwärtigen Zeitpunkt ist es bereits möglich, dass eine interaktive Komponente einen Ausgabeparameter, einer anderen interaktiven Komponente, als Eingabeparameter nutzt. Hier könnte man ansetzen, um das Konzept der Interkomponentenoperabilität zu verbessern. Betrachtet man eine interaktive Komponente als Mittel zum Bewältigen einer konkreten Aufgabe, könnte eine Abfolge solcher Aufgaben mittels einer Art einer Workflow-Beschreibungssprache beschrieben werden, welche dann von der Zielanwendung interpretiert werden müsste. Eine Instanz einer solchen Sprache, würde dann neben den interaktiven Komponenten mit im Repository veröffentlicht werden.

A. Anhang

A.1. Usability-Studie

Usability Study

Georg Berndt, TU-Dresden

This usability study is separated into two tasks. The first task based on the patient service scenario, which has been used in the evaluation of the *Interactive Component Editor (ICE)*. In the second task another service scenario will be used, which is based on a home network environment.

Task 1

The application, which integrates the generated components resulting of modeling the patient service scenario, is called *Mobile Medical System*. You can find this application on the home screen of the device, which is used during the evaluation. In a first step you should explore the application. After a certain time, you should be able to answer the following questions and perform the following subtasks. If you could answer the given questions and perform the given subtasks, you can start with the first evaluation task. If you have some troubles completing a subtask, please write them down in few sentences and nevertheless start with the first evaluation task.

Q1.1 Which bloodgroup has Mr. Schulz?

Notes: _____

Q1.2 What is the PZN-Number of the medicament *Buscopan plus*? The category for this medicament is *"Schmerzen"*, which is the german description for painkillers.

Notes: _____

ST1.1 Update the weight of Mr. Schulz to 83.6.

Notes: _____

ST1.2 Add a patient to the *Mobile Medical System*. (Name: John Doe; Bloodgroup: AB; Height: 182; Weight: 83)

Notes: _____

Evaluation 1

expectation		**0**	**1**	**2**	**3**	**4**	**5**	**6**		**NA**
Have your expectations been met regarding to the application, which you have modeled by using the *ICE*?	not a bit								absolutely	
Was the modeling by ICE helpful for the navigation in this application?	not a bit								absolutely	
How good has the modeled instance been realized as application?	very bad								very good	

terminology and information		**0**	**1**	**2**	**3**	**4**	**5**	**6**		**NA**
Using of terms throughout the application?	inconsistent								consistent	
Terminology is intuitive?	never								always	
Prompts for input?	confusing								very clear	
Application informs about progress?	never								always	
Error messages?	unhelpful								helpful	

learning		**0**	**1**	**2**	**3**	**4**	**5**	**6**		**NA**
Learning to use the application?	difficult								easy	
Exploring features by trial and error?	difficult								easy	
Performing tasks is straightforward?	never								always	

application capabilities		**0**	**1**	**2**	**3**	**4**	**5**	**6**		**NA**
Application speed?	too slow								fast enough	
Application reliability?	unreliable								reliable	
Correcting your mistakes?	difficult								easy	

Task 2

The application, which is used in the second task, is called *My Home Manager*. This application can manage several devices or services of a home network. Currently an *alarm system*, a *network router*, a *lighting system* and a *DVB device* can be accessed. As in the first task, you should firstly discover the application. The required initial user credentials consist of *"user"* as user name, and of *"pass"* as password. After that you should be able to answer the following questions and perform the following subtasks. You can start with the evaluation, if the subtasks have been completed. Are there some problems during performing these tasks, note them in few sentences, and start also with the evaluation.

Q2.1 How many rooms can be managed by the lighting service?

Notes: _____

Q2.2 What is the Channel-ID of channel "ARTE"?

Notes: _____

Q2.3 Which port is currently forwarded by the network router?

Notes: _____

ST2.1 Configure the repository user credentials. The configuration screen is accessible by the application menu which can be opened by pressing the *menu*-key. Use the following user credentials: User: **teststudent**; Password: **teststudent@rn**

Notes: _____

ST2.2 Start a component update by using the option *Check for Update* of the application menu.

Notes: _____

Evaluation 2

integration		**0**	**1**	**2**	**3**	**4**	**5**	**6**		**NA**
How easy was the configuration of the repository access?	difficult								easy	
How easy was the updating of the components?	difficult								easy	
How good has the integration of the different components been realized?	very bad								very good	

application usability		**0**	**1**	**2**	**3**	**4**	**5**	**6**		**NA**
Organization of the information?	confusing								very clear	
Sequence of screens?	confusing								very clear	

Free Questions

What do you consider the biggest problem of this application at present?

_____

_____

What should be improved in the future?

_____

_____

18. How can such an improvement be done?

_____

_____

A.2. Ant-Tasks

```
<project name="runAll" default="deploy" basedir="..">

  <target name="runWorkflow">
    <echo message="start generation ..."/>
    <ant antfile="ant/runWorkflow.xml" dir="." target="generate">
      <property name="workflow.exec.dir" value="${workflow.exec.dir}"/>
    </ant>
  </target>

  <target name="build" depends="runWorkflow">
    <echo message="start build ..."/>
    <ant antfile="ant/build.xml" dir="." target="subbuild"/>
  </target>

  <target name="deploy" depends="build">
    <echo message="start deployment ..."/>
    <ant antfile="ant/deploy.xml" dir="." target="svn-commit">
      <property name="repository.dir" value="${repository.dir}"/>
      <property name="repository.iComponentFolder" value="${repository.iComponentFolder}"/>
      <property name="repository.user" value="${repository.user}"/>
      <property name="repository.pass" value="${repository.pass}"/>
    </ant>
  </target>

</project>
```

Listing A.1: Übergeordneter Task

```
<project default="generate" basedir="..">

  <property name="lib.workflow" value="../libDependencies"/>
  <property name="sbic.jar" value="../rn.sbicmodel/sbic.jar"/>
  <property name="helper.jar" value="generator-helper.jar"/>

  <target name='generate'>
    <echo message="Workflow Execution Dir: ${workflow.exec.dir}"/>
    <java resultproperty="returnCode" classname="org.eclipse.emf.mwe.core.WorkflowRunner" dir="${
        workflow.exec.dir}">
      <classpath>
        <fileset dir="${lib.workflow}">
          <include name="**/*.jar"/>
        </fileset>
        <fileset dir="lib/hybridlabs-beautifier">
          <include name="**/*.jar"/>
        </fileset>
        <pathelement location="${sbic.jar}"/>
        <pathelement location="${helper.jar}"/>
      </classpath>
      <arg value="src/workflow/workflow.mwe"/>
    </java>
    <available file="ant-gen/bnd.properties" property="workflow.success"/>
    <fail unless="${workflow.success}" message="Workflow interrupted!"/>
  </target>

</project>
```

Listing A.2: Task zum Starten der M2C-Transformation

```
<project default="subbuild" basedir="..">

  <property file="ant/build.global.properties"/>
  <property file="ant-gen/bnd.properties"/>

  <target name="clean">
    <delete dir="${build.dir}"/>
    <mkdir dir="${build.dir}"/>
  </target>

  <target name="subbuild" depends="clean">
    <ant antfile="ant/build_bundle.xml" dir="." target="dex">
      <property name="bundle" value="${UIBundleName}"/>
    </ant>
    <ant antfile="ant/build_bundle.xml" dir="." target="dex">
      <property name="bundle" value="${CoordBundleName}"/>
    </ant>
    <ant antfile="ant/build_bundle.xml" dir="." target="dex">
      <property name="bundle" value="${ConnectBundleName}"/>
    </ant>
    <ant antfile="ant/build_bundle.xml" dir="." target="dex">
      <property name="bundle" value="${ServicesBundleName}"/>
    </ant>
  </target>

</project>
```

Listing A.3: Übergeordneter Task zur Paketierung generierter Komponenten

```
<project default="dex" basedir="..">

  <property file="ant/build.global.properties"/>
  <property file="ant/build.local.properties"/>

  <echo message="Build ${bundle}"/>

  <taskdef resource="aQute/bnd/ant/taskdef.properties"
      classpath="lib/bnd-0.0.356.jar"/>

  <taskdef name="ipojo"
      classname="org.apache.felix.ipojo.task.IPojoTask"
      classpath="lib/org.apache.felix.ipojo.ant-1.4.2.jar"/>

    <target name="package">
    <bnd
      classpath="lib/org.apache.felix.ipojo.annotations-1.4.0.jar"
        failok="false"
      eclipse="true"
      exceptions="true"
        files="ant-gen/${bundle}.bnd"
      output="${build.dir}"/>
    <ipojo
      input="${build.dir}/${bundle}.jar"
      metadata="ant-gen/metadata_${bundle}.xml"/>
  </target>

    <target name="dex" depends="package" >
      <delete file="classes.dex"/>
      <exec executable="${android.home}/tools/dx.bat">
        <arg line="--dex --output=${project.dir}/classes.dex ${project.dir}/${build.dir}/${bundle}.
            jar"/>
      </exec>
      <exec executable="${android.home}/tools/aapt.exe">
        <arg line="add ${project.dir}/${build.dir}/${bundle}.jar classes.dex"/>
      </exec>
    </target>

</project>
```

Listing A.4: Subtask zur Paketierung generierter Komponenten

```
<project default="svn-commit" basedir="..">

  <property file="ant/build.global.properties"/>
  <taskdef name="svn" classname="org.tigris.subversion.svnant.SvnTask">
    <classpath>
      <fileset dir="${lib.dir}">
        <include name="**/svn*.jar"/>
      </fileset>
    </classpath>
  </taskdef>

  <target name="copy">
    <copy todir="${repository.dir}/${repository.iComponentFolder}">
      <fileset dir="${build.dir}">
        <include name="*.jar"/>
      </fileset>
    </copy>
  </target>

  <target name="svn-add" depends="copy">
    <svn username="${repository.user}" password="${repository.pass}">
       <add dir="${repository.dir}" force="true">
        <fileset dir="${repository.dir}">
          <include name="*.jar"/>
        </fileset>
      </add>
    </svn>
  </target>

  <target name="svn-commit" depends="svn-add">
    <svn username="${repository.user}" password="${repository.pass}">
        <commit dir="${repository.dir}" message="component update"/>
        <info target="${repository.dir}"/>
    </svn>
    <echo>
        Last Revision = ${svn.info.lastRev} by ${svn.info.author} on ${svn.info.lastDate}
    </echo>
  </target>

</project>
```

Listing A.5: Task zum Veröffentlichen interaktiver Komponenten

A.3. Xpand-Caller

```
1  package dynamicinvoke;
2
3  public class XpandCaller {
4
5    private final XpandFacade facade;
6
7    @SuppressWarnings("unchecked")
8    public XpandCaller( final String mmFile, String srcGenPath, String[] mmPackages ){
9
10     //src-gen
11     IPath path = new Path(srcGenPath);
12     String containerName = path.toPortableString();
13
14     // configure outlets
15     OutputImpl output = new OutputImpl();
16     Outlet outlet = new Outlet(containerName);
17     outlet.setOverwrite(true);
18     output.addOutlet(outlet);
19
20     // resolver for protected regions
21     ProtectedRegionResolverImpl prs = new ProtectedRegionResolverImpl();
22     prs.setSrcPathes(containerName);
23
24     // create execution context
25     Map globalVarsMap = new HashMap();
26     XpandExecutionContextImpl execCtx = new XpandExecutionContextImpl(
27         output, prs, globalVarsMap, null, null);
28
29     for (int i=0; i<mmPackages.length; i++) {
30       EmfMetaModel mm = new EmfMetaModel();
31         if (i==0)
32           mm.setMetaModelFile(mmFile);
33         mm.setMetaModelPackage(mmPackages[i]);
34         execCtx.registerMetaModel(mm);
35     }
36       facade = XpandFacade.create(execCtx);
37   }
38
39   public void evaluate(String definitionName, Object targetObject,
40       Object... params) {
41     facade.evaluate(definitionName, targetObject, params);
42   }
43 }
```

Listing A.6: XpandCaller - Implementierung

Abkürzungsverzeichnis

ASM	Annotated Serive Model.
B2B	Business-to-Business.
B2C	Business-to-Consumer.
CAM	Composite Application Model.
CRUISe	Composition of Rich User Interface Services.
CTT	Concur Task Trees.
GUIDD	GUI-Deployment-Descriptor.
ICE	Interactive Component Editor.
JVM	Java Virtual Machine.
LOC	Lines of Code.
M2C	Modell-zu-Code.
M2M	Modell-zu-Modell.
MARIA	Model-based description of Interactive Applications.
MWE	Modeling Workflow Engine.
OWL-S	Web Ontology Language for Web Services.
RCP	Rich Client Platform.
SBIC	Service-based Interactive Component-Model.
SOA	Service-Oriented Architecture.
SVN	Apache Subversion.
UI	User Interface.
UIS	User Interface Service.
UUID	Universally Unique Identifier.
WSDL	Web Service Definition Language.
WSGUI	Web Services Graphical User Interface.

XSLT XSL-Transformation.

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Algorithmenverzeichnis

Literaturverzeichnis

Ber09 Georg Berndt. Ad-hoc Integration of Annotated Services to Instances of a Meta-model of Interactive Service-based Applications. Großer beleg, Dresden University of Technologie, 2009. 3.2.8

BHL+04 Mark Burstein, Jerry Hobbs, Ora Lassila, Drew Mcdermott, Sheila Mcilraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. OWL-S: Semantic Markup for Web Services. Website, November 2004. 3.2.4

Boy09 John Boyer. XForms 1.1. W3C recommendation, W3C, October 2009. http://www.w3.org/TR/2009/REC-xforms-20091020/. 3.2.1

BSR+07 Gregor Broll, Sven Siorpaes, Enrico Rukzio, Massimo Paolucci, John Hamard, Matthias Wagner, and Albrecht Schmidt. Supporting Mobile Service Usage through Physical Mobile Interaction. In *PERCOM '07: Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications*, pages 262–271, Washington, DC, USA, 2007. IEEE Computer Society. 3.2.4

CCMW01 Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Service Definition Language (WSDL). Technical Report NOTE-wsdl-20010315, World Wide Web Consortium, March 2001. 1.2

Cle08 Clement. iPOJO on Android. Blog, 2008. Available online at `http://ipojo-dark-side.blogspot.com/2008/10/ipojo-on-android.html`; visited on March 3rd 2010. 3.1.2

CRU10 Composition of Rich User Interface Services. `http://mmt.inf.tu-dresden.de/cruise`, 2010. 3.2.7

Dus07 L. Dusseault. HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV). RFC 4918 (Proposed Standard), June 2007. Updated by RFC 5689. 4.2.2

Fel11 Marius Feldmann. *Ein Verfahren zur Generierung interaktiver Komponenten aus annotierten funktionalen Dienstschnittstellenbeschreibungen.* Dissertation, Dresden Technical University, 2011. 1.2, 1.2, 2.2, 2.2, 3.1.3, 3.1.3, 3.1.3, 3.1.3, 3.2.8, 4.1.3.2, 4.2.1.1, 4.2.3.3, 4.2.3.3, 5, 5.1, 5.2, 5.3, 5.3, 5.4, 5.5, 6

FJN+09 Marius Feldmann, Jordan Janeiro, Tobias Nestler, Gerald Hübsch, Uwe Jugel, André Preussner, and Alexander Schill. An Integrated Approach for Creating Service-Based Interactive Applications. In *INTERACT '09: Proceedings of the 12th IFIP TC 13 International Conference on Human-Computer Interaction*, pages 896–899, Berlin, Heidelberg, 2009. Springer-Verlag.

FNM+09 M. Feldmann, T. Nestler, K. Muthmann, U. Jugel, G. Hübsch, and A. Schill. Overview of an end-user enabled model-driven development approach for interactive applications based on annotated services. In *WEWST '09: Proceedings of the 4th Workshop on Emerging Web Services Technology*, pages 19–28, New York, NY, USA, 2009. ACM.

GHJV95 Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995. 4.2.1.2, 4.2.3.1

Gru09 Wolfgang Gruber. Metawidget: King of the Hill. Blog, 2009. Available online at `http://it-republik.de/jaxenter/artikel/Metawidget-King-of-the-Hill-2681.html`; visited on August 26th 2010.

HY07 Jiang He and I-Ling Yen. Adaptive User Interface Generation for Web Services. In *ICEBE '07: Proceedings of the IEEE International Conference on e-Business Engineering*, pages 536–539, Washington, DC, USA, 2007. IEEE Computer Society.

JSRa Jsr-000168 Portlet Specification. `http://jcp.org/aboutJava/communityprocess/final/jsr168/`. 3.2.6

JSRb Jsr-000177 Security and Trust Services API for J2ME. `http://jcp.org/aboutJava/communityprocess/final/jsr177/index.html`.

JSRc Jsr-000286 Portlet Specification 2.0. `http://jcp.org/aboutJava/communityprocess/final/jsr286/`. 3.2.6

Kaw04 Jalal Kawash. Declarative user interfaces for handheld devices. In *WISICT '04: Proceedings of the winter international synposium on Information and communication technologies*, pages 1–6. Trinity College Dublin, 2004. 3.2.3

KEL09 Richard Kennard, Ernest Edmonds, and John Leaney. Separation anxiety: stresses of developing a modern day separable user interface. In *Proceedings of the 2nd conference on Human System Interactions*, HSI'09, pages 225–232, Piscataway, NJ, USA, 2009. IEEE Press.

KL10 Richard Kennard and John Leaney. Towards a general purpose architecture for UI generation. *Journal of Systems and Software*, In Press, Corrected Proof:–, 2010. 3.2.5

Kri10 Peter Kriens. About OSGi @ONLINE, June 2010.

Lie07 Daniel Liebhart. *SOA goes real*. Hanser Fachbuchverlag, 2007. 1.1

Lie08 Liebing. Rest-basierte Web Services. Master's thesis, Dresden Technical University, 2008. 1.1

Mar09 Felix Martens. Ad-hoc Integration of Annotated Services for Google Android. Großer beleg, Dresden University of Technologie, 2009.

Mar10 Felix Martens. Konzeption und Umsetzung eines Werkzeugs zur Definition von Navigationsflüssen mittels Dienstannotationen. Diplomarbeit, Dresden University of Technologie, 2010. 1.2, 1.2, 3.1.3, 3.2.8, 5, 5.1, 5.2, 5.3, 5.4, 5.4, 6

Mas07 Dieter Masak. *SOA?: Serviceorientierung in Business und Software (Xpert.press) (German Edition)*. Springer, 2007. 1.1

OSG07 OSGi Alliance. OSGi Service Platform Release 4. [Online]. Available: http://www.osgi.org/Main/HomePage. [Accessed: Jun. 17, 2009], 2007. 3.1.1

PMM97 Fabio Paternò, Cristiano Mancini, and Silvia Meniconi. ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In *INTERACT '97: Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction*, pages 362–369, London, UK, UK, 1997. Chapman & Hall, Ltd. 3.2.2.2

Pro09 Metawidget Project. Metawidget - Project Page. Website, 2009. Available online at `http://metawidget.org`; visited on August 26th 2010. 3.2.5

PSS09 Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. Support for authoring service front-ends. In *EICS '09: Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, pages 85–90, New York, NY, USA, 2009. ACM. 3.2.2.2

PVRM09 Stefan Pietschmann, Martin Voigt, Andreas Rümpel, and Klaus Mei. Cruise: Composition of Rich User Interface Services. In *ICWE '9: Proceedings of the 9th International Conference on Web Engineering*, pages 473–476, Berlin, Heidelberg, 2009. Springer-Verlag. 3.2.7

RLMM09 Rick Rogers, John Lombardo, Zigurd Mednieks, and Blake Meike. *Android Application Development: Programming with the Google SDK*. O'Reilly, Beijing, 2009.

Rum09 Przemyslaw Rumik. Android + Metawidget = Interesting cooperation. Blog, 2009. Available online at `http://racjonalny.blogspot.com/2009/11/android-metawidget-interesting.html`; visited on August 23th 2010. 3.2.5

Ser09 The ServFace Annotation Model. Technical report, ServFace Project, September 2009. 1.2, 4.1.1.3, 5.1

Spi06 Josef Spillner. Project Dynvocation. Diploma, Dresden Technical University, 2006. 1.1, 1.2, 3.2.1

WHKL08 Gerd Wütherich, Nils Hartmann, Bernd Kolb, and Matthias Lübken. *Die OSGi Service Platform: Eine Einführung mit Eclipse Equinox*. dpunkt, Heidelberg, 2008. 3.1.1

YA06 Xiaobo Yang and Rob Allan. A Deep Look at Web Services for Remote Portlets (WSRP) and WSRP4J, 2006.

ZF09 Klaus Zeppenfeld and Patrick Finger. *SOA und WebServices (Informatik im Fokus) (German Edition)*. Springer, 2009. 1.1