Konfigurierung von eHome-Systemen

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der Rheinisch-Westfälischen Technischen Hochschule Aachen zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Ulrich Norbisrath

aus Aachen

Berichter:
Universitätsprofessor Dr.-Ing. Manfred Nagl, RWTH-Aachen
Universitätsprofessor Dr. rer. nat. Albert Zündorf, Universität Kassel

Tag der mündlichen Prüfung: 27.6.2007

DAS EIGENTLICHE BRAUCHT KEINE REGELN. LICHT IST LICHT, WIRKEN IST WIRKEN, HINGABE IST HINGABE. WER DAVON SPRICHT, VERLÄSST DAS EIGENTLICHE. WER VON PFLICHT REDET, HAT DAS TAO VERLASSEN. WER REGELN UND VERGLEICHE BRAUCHT, KENNT DAS EIGENTLICHE NICHT. WER WIRKT UM DER WERKE WILLEN, IST NICHT MEHR EIGENTLICH. WER LIEBE WILL, WIRD NICHT GELIEBT. WER RUHM WILL, WIRD NICHT BERÜHMT. WER MACHT WILL, IST NICHT MÄCHTIG. WER FRIEDEN WILL, BLEIBT IM UNFRIEDEN. WER TUGEND WILL, WIRD NICHT TUGENDHAFT. WER SICH IM SPIEGEL SCHAUT, SIEHT SICH NICHT. DAS EIGENTLICHE IST EINFACH DA. DAS EIGENTLICHE KANN GEFUNDEN WERDEN, ABER NICHT GESUCHT.

(ZITAT GUNTHER DUECK, OMNISOPHIE)

Zusammenfassung

Diese Arbeit stellt eine Möglichkeit vor, den Prozess der Einrichtung sogenannter Smart-Home- oder eHome-Systeme aus softwaretechnischer Sicht zu unterstützen. Dies beinhaltet insbesondere, die Erstellung und Zusammenstellung von Software für solche Systeme zu vereinfachen. Das Hauptaugenmerk dieser Arbeit liegt darauf, eine Verschiebung des Entwicklungsprozesses zu einem beteiligten Diensteanbieter hin zu ermöglichen und gleichzeitig den Entwicklungsaufwand zu minimieren.

Es gibt verschiedene Gebiete in einem potentiell automatisierten Heim, in denen mit eHome-Diensten eine Unterstützung im Alltag erreicht werden könnte, wie zum Beispiel Lebenskomfort, Sicherheit, Kommunikation oder Unterhaltung. Ein Sicherheitsdienst kann auch Dienste aus einem anderem Gebiet integrieren, insbesondere Dienste aus dem Kommunikationssektor zum Versand von Warnmeldungen oder Dienste aus dem Unterhaltungsbereich, um auf eine Gefahr aufmerksam zu machen. Gerade solche gebietsübergreifende Dienste sind die, denen das meiste Potential zugeschrieben wird, da sie sich am meisten einer intuitiven Vorstellung von einem intelligenten Dienst nähern.

Die Preise der Geräte, die in den gerade angesprochenen Diensten benutzt werden, sind größtenteils auf ein finanzierbares Maß gesunken, so dass es verwunderlich scheint, dass eHome-Systeme nicht schon längst in den meisten Wohnungen verfügbar sind. Einer der Hauptgründe, der eine Verbreitung von eHomes verhindert, ist der Preis solcher Systeme. Wie aber gerade beschrieben, ist es nicht der Preis der Hardware, der hier ins Gewicht fällt, sondern der Preis der Software, welche bisher noch für jedes neue eHome komplett neu entwickelt wird. Im Allgemeinen wird ein Hausbewohner aber keinen kompletten Software-Entwicklungsprozess finanzieren können.

Für den Softwaretechniker ist es nun besonders interessant, wie sich dieser Entwicklungsprozess für Dienste in neuen eHome-Systemen so vereinfachen oder anders strukturieren lässt, dass seine Kosten einem großen zu erwartenden Markt angemessen werden. Somit ist der Kern dieser Arbeit die Umwandlung und Neustrukturierung des eHome-Entwicklungsprozesses in einen teilweise automatisierten Konfigurierungsprozess (eHome-SCD-Prozess) und dessen werkzeugseitiger Unterstützung.

Mit Hilfe des in dieser Arbeit vorgestellten neu strukturierten Prozesses und einer diesen Prozess unterstützenden Werkzeugsuite, der eHomeConfigurator-Werkzeugsuite, wird die eigentliche Entwicklungsarbeit für den Endbenutzer auf die Spezifizierung seiner Umgebung, einiger zu benutzender Dienste und einiger beschreibender Parameter reduziert, ohne Code angeben zu müssen. Die Arbeit zeigt, dass durch den Einsatz sogenannter Funktionalitäten-basierter Komposition und automatischer Konfigurierungstechniken eine signifikante Erleichterung des Entwicklungsaufwandes erzielt werden kann.

Die Anwendung der eHomeConfigurator-Werkzeugsuite an zwei eigenen Demonstratoren und bei einem externen Kooperationspartner bestätigt, dass speziell für die Werkzeugsuite entwickelte Dienste in völlig unterschiedlichen Umgebungen ohne Anpassungen neu eingesetzt werden können und sich sowohl der Entwicklungsaufwand vor dem eHome-SCD-Prozess, also die Dienstentwicklung, als auch der Aufwand innerhalb des Prozesses durch den Einsatz automatisierbarer Konfigurierung und interaktiver Werkzeugunterstützung erheblich reduziert.

Abstract

An intelligent home, what could that be? More and more electronic appliances enter our private homes. Their presence seems to become ubiquitous. But we usually do not associate intelligence when we install new electronic appliances in our home environment. When will we start to call a setup of such appliances intelligent? Probably if this setup simplifies tasks at hand in our daily lifes or increases security or comfort of living. Setups of such intelligent home environments in different places all over this world have given us a clue of some useful setups and some useful services which could be provided by these. Such homes are usually called smart homes or eHomes.

This work introduces a software engineering approach to support the installation of such setups for regular home owners. Its key contribution is the dissolution of the software development process in favor of a partly automated configuration process.

There are various fields where the services provided in such eHome environments could support our daily life: One would be the enhancement of your comfort. This could be a central remote control, personal information management, or even an advanced wakeup service. Another need, which could be satisfied by such a service is security. In this field video surveillance or interactive alarm systems are possible. In the field of communication email, voice-over-ip, or instant messaging services are already widely used. There is a great potential for services in the health sector. Telemedicine or instant medical advise based on up-to-date sensor data are just some examples. Furthermore, infotainment services, like video on demand, teaching, or advanced multimedia experiences are of great interest. Via the monitoring facilities of energy consuming devices, automatic optimizations or exact billing is possible. Of course these areas are not very clearly separated from each other. A security service can also integrate services from other fields, especially using communication or even infotainment abilities. Services integrating different fields will be the most interesting services for potential customers of eHome systems.

As the prices of lots of appliances used in the previously discussed services have decreased to affordable amounts. The appliances are affordable for regular home-owners. Hence, from this point of view the hardware for the realization of smart home environments is already available at reasonable costs. The question arising is: Why are eHomes not more conventional? If you have a closer look at all these setups you will discover that these implementations are either research or hobby projects. One of the main barriers blocking a broader spreading of eHomes systems is the price of such systems. Even if appliances are affordable, the software driving an eHome is rather expensive as it is mostly developed or adapted for every single eHome. A complete software development process per case is not affordable for everyone.

As software engineers, we are particularly interested in simplifying the software development process, arising when implementing a service for a specific home environment. The vision is: If the software for eHomes could be reused, and its adaptation and configuration be automated, one of the price barriers on home automation mass-market would be broken. The key point of this work is the dissolution of the software development process for eHome systems.

It seems obvious that such systems have to be split in comprehensive components to achieve reusability. We distinguish between basic and extension services offered by components. Extension services are services composed of various basic services. These ser-

vices usually represent the functionality the inhabitants are interested in. For component-based development a close view on the middleware is important. Today's component-based middleware solutions offer dynamic composition, configuration and deployment facilities. We have used different middleware solutions for obtaining results for realizing one integrating service, but we are now focusing on realizing different integrating services in different home environments. Pure middleware solutions do not enable a customer to build up an eHome system for his personal environment on the base of an integrating service.

Our process is reduced to the level of mere specification of the environment and services, and interactive configuration of the given service components into the eHome system with no coding overhead. Our goal is the support of composition and configuration of integrating services for eHome systems. In this context, we introduce a special simplified specification, configuration, and deployment process. We will refer to this as the eHome-SCD-process.

Danksagung

Viele Menschen haben mich in der Zeit vor und während der Promotion begleitet. Ich kann sicher nicht alle hier hervorheben, werde aber doch versuchen die zu erwähnen, die mir in dieser Zeit viel bedeutet haben.

Mein erster Dank richtet sich an Sandra Saladin, die mich überhaupt erst auf die Idee der Promotion brachte. Weiterer Dank geht an Herrn Prof. Indermark, der mir den Mut und die Unterstützung gab, an die Universität zurückzukehren und der mir immer ein Vorbild für ein ausgewogenes Verhältnis zwischen Objektivität, Fairness, Korrektheit und Menschlichkeit bleiben wird.

Besonderen Dank schulde ich auch meinem Doktorvater Herrn Prof. Manfred Nagl, der mich aufgenommen und betreut hat. Unsere kontroversen Diskussionen haben sehr zum Gelingen dieser Arbeit beigetragen. Herrn Prof. Albert Zündorf danke ich für die Übernahme des Zweitgutachtens und dafür, dass er mir in wissenschaftlichen Krisen den Glauben und meine Motivation durchzuhalten wiedergegeben hat.

Vielen Dank richte ich auch an meine lieben Eltern Uschi und Walter Norbisrath, die mich immer unterstützt und auch im letzten Jahr wieder aufgenommen haben. Sie haben mir das Zweifeln und Denken als Grundstein ihrer Erziehung mit auf den Weg gegeben. Insbesondere danke ich beiden für zahlreiche Korrekturen und Walter für die Unterstützung beim Bau des Lego-eHome-Demonstrators.

Ein herzliches Aitäh geht an meinen Master-Studenten Priit Salumaa (Arbeit [Sal05]), der mir ein guter Freund geworden ist und mir interessante neue Wege in die Zukunft aufgezeigt hat. Natürlich danke ich ihm auch für viele hilfreiche Kommentare in der Korrekturphase.

Ich danke meinem Kollegen und ersten Diplomanden Christof Mosler (Arbeit [Skr04]) und meinen Kollegen Ibrahim Armac, Trinh Nguyen und Markus Heller, die ich sehr schätzen gelernt habe. Meinem ehemaligen Kollegen Michael Kirchhof danke ich für die faire und äußerst produktive Zusammenarbeit.

Dank richte ich auch an meinen Vorgesetzten in der Fachgruppe, Herrn Professor Kobbelt, der trotz meiner Doppelbeschäftigung mit Dissertationsschreiben und Fachgruppenbetreuung immer geduldig mit mir zusammengearbeitet hat. Ich danke Herrn Prof. Rossmanith, dem ich einige heitere Stunden, wertvolle Gespräche und viel Motivation verdanke, in eine akademische Zukunft zu investieren. Besonderen Dank spreche ich auch Herrn Prof. Spaniol aus, da er mir den Bau meines ersten Demonstrators ermöglichte und meine Arbeit damit sehr gefördert hat. Üks suur aitäh to Prof. Eero Vainikko for offering me a perspective which was far beyond what I ever could expect.

Unserer Fachgruppensekretärin Valentina Elsner danke ich als zuverlässiger und liebenswerter Kollegin und Vertrauten. Vielen Dank richte ich auch an unsere Sekretärinnen Angelika Fleck und Klaudia Winckler, die für jedes bürokratische Problem eine schnelle Lösung gefunden haben.

Meinem Diplomanden Christian Pöcher danke ich, dass er mich nach Estland begleitet und meinen Weg dort vorbereitet hat. Ebenso möchte ich Adam Malik meinen Dank für seine Pionierleistungen für den eHomeConfigurator mit seiner Diplomarbeit [Mal05] und seine Beteiligung an der Korrektur aussprechen.

Meinen weiteren Diplomanden Arash Akhoundi [Akh05], Philipp Böckers [Böc06],

Markus Klinke [Kli04], Ingo Kreienbrink [Kre04] und Tim Schwerdtner [Sch05b] möchte ich für die Unterstützung meines Projektes mit ihren Arbeiten danken.

Allen Teilnehmern des eHome-Praktikums, in dem der Grundstein des eHomeConfigurators gelegt wurde, danke ich ganz herzlich. Ohne euch hätte es vielleicht keinen eHomeConfigurator gegeben.

Peter Gabriel danke ich für quantitative Daten zu inHaus. Leif Geiger und Christian Schneider danke ich für ihre Geduld und ihren Fujaba- und CoObRA-Support. Weiterhin gebührt Dank für Hilfe bei der Korrektur und konstruktive Kritik Tibor Bulecza, Marion Reuter, Axel Behr und Andreas Neudecker. Klaas-Henning Müller unterstützte mich bei Druck und Abgabe. Vielen Dank für diese wichtige logistische Unterstützung.

Jah minu kallis Jenn: Thank you for the great support in the last weeks. Your encouragement and patience played a major role in finishing this piece of work. Ma armastan sind!

Let us discover the After-PhD-Uli!

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Einführung

Ein intelligentes Heim – was kann man sich unter dieser Formulierung vorstellen? Ein Haus, das für einen denkt? Sicher nicht. Aber vielleicht ein Haus, welches einem den Alltag erleichtert oder versüßt, könnte dieses Attribut verdienen. Immer mehr elektronische Geräte halten Einzug in unser tägliches Leben. Sie werden immer unsichtbarer, sind aber doch allgegenwärtig, oder, um das so gerne benutzte Fremdwort zu verwenden, sie werden *ubiquitär* (eng.: *ubiquitous*)! Ab wann können solche Geräteinstallationen intelligent genannt werden? Ein Indiz für eine intelligente Eigenschaft eines solchen Systems wäre, wenn es tatsächlich Abläufe im täglichen Leben vereinfachen würde oder Sicherheit oder Lebenskomfort steigerte [Wei91]. Verschiedene Installationen solcher intelligenter Heimumgebungen überall auf dieser Welt [inH05, Sch05a, Bei05, TS05, Phi02] geben uns eine Vorstellung davon, welche Zusammenschaltungen und welche Dienste in deren Zusammenhang nützlich und sinnvoll sein können. Heimumgebungen, die auf solche Weise intelligente Dienste anbieten, werden in der Literatur auch intelligentes Heim, SmartHome oder eHome genannt.

Diese Arbeit stellt eine Möglichkeit vor, den Prozess der Einrichtung vieler solcher Systeme softwaretechnisch zu unterstützen. Das Hauptaugenmerk liegt darauf, eine Verschiebung des Entwicklungsprozesses zu einem beteiligten Diensteanbieter hin zu ermöglichen und gleichzeitig den Entwicklungsaufwand zu minimieren. Die Notwendigkeit der Softwareentwicklung beim Kunden soll auf einen teilweise automatisierbaren Konfigurationsprozess reduziert werden. Dadurch wird die Einrichtung von eHome-Systemen nicht nur im Einzelfall sondern auch für einen breiten Markt möglich. Diese Arbeit zeigt, wie diese Herauslösung des Entwicklungsanteils und die teilweise Automatisierung des verbleibenden Konfigurierungsprozesses möglich werden.

Es gibt verschiedene Gebiete in einem potentiell automatisierten Heim, in denen mit eHome-Diensten eine Unterstützung im Alltag erreicht werden könnte: Eines wäre das Gebiet des Lebenskomforts. Hierzu würden eine zentrale Fernbedienung zur einheitlichen Steuerung aller elektronischen Geräte im Haus, ein Personal-Information-Management-System aber auch ein ausgeklügelter Weckdienst gehören. Ein anderes Gebiet, in dem ein eHome-System Sinn machen würde, wäre das der Sicherheit. Hier wären Videoüberwa-

chungssysteme oder interaktive Alarmanlagen möglich. Im Bereich der Kommunikation sind Dienste wie E-Mail, Voice- und Video-Over-IP oder Instant-Messaging bereits weit verbreitet. Ein großes Potential haben auch Dienste im Bereich des Gesundheitssektors. Telemedizin oder direkter medizinischer Rat basierend auf aktuellen Sensordaten sind nur zwei Beispiele. Dienste des Infotainment-Sektors sind bereits jetzt im Fokus großer Firmen wie Microsoft, Philips oder Sony [Ros05, Phi02, HAV01]. Hier sind Video-on-Demand, e-Learning, oder erweiterte Multimedia-Erfahrungen zum Beispiel im audiovisuellen 3D-Bereich von großem Interesse. Mittels der Abfrage und Messung von Verbrauchsdaten der Energieverbraucher wie Heizung oder Durchlauferhitzer, werden automatische Energieoptimierung oder die exakte Verbrauchserfassung und Abrechnung möglich. Natürlich sind die Übergänge zwischen diesen Gebieten fließend. Ein Sicherheitsdienst kann auch Dienste aus einem anderem Gebiet integrieren, insbesondere Dienste aus dem Kommunikationssektor zum Versand von Warnmeldungen oder Dienste aus dem Infotainmentbereich, um auf eine Gefahr aufmerksam zu machen. Gerade solche gebietsübergreifende Dienste sind die, denen das meiste Potential zugeschrieben wird, da sie sich am meisten der intuitiven Vorstellung von einem intelligenten Dienst nähern.

Die Preise der Geräte, die in den gerade angesprochenen Diensten benutzt werden, sind größtenteils auf ein Maß gesunken, dass sie in einer mittleren Einkommensschicht, bezogen auf westeuropäische Verhältnisse, finanzierbar sind. Das heißt, dass sich die meisten Menschen solche Geräte zur Einrichtung eines eHomes bereits leisten könnten. Die Frage, die sich stellt, ist: Warum sind eHome-Systeme nicht schon längst allgegenwärtig und in den meisten Wohnungen verfügbar? Bei der Betrachtung bereits bestehender eHome-Systeme lässt sich feststellen, dass es sich bei diesen in der Regel um Forschungs- [inH05, TS05, Phi02] oder Hobbyprojekte [Sch05a] handelt. Einer der Hauptgründe, der eine Verbreitung von eHomes verhindert, ist der Preis solcher Systeme. Wie aber gerade beschrieben ist es nicht der Preis der Hardware, der hier ins Gewicht fällt, sondern der Preis der Software. Denn diese wird bisher noch für jedes neue eHome komplett neu entwickelt. Im Allgemeinen wird ein Hausbesitzer oder Mieter aber keinen kompletten Software-Entwicklungsprozess finanzieren können.

Für den Softwaretechniker ist es nun besonders interessant, wie sich dieser Entwicklungsprozess, der auftritt, wenn ein neuer Dienst für eine neue Umgebung umgesetzt oder angepasst werden soll, vereinfachen oder anders strukturieren lässt, so dass seine Kosten einem großen zu erwartenden Markt angemessen werden. Die Vision dieser Arbeit ist: Wenn die Software für eHome-Systeme größtenteils wiederverwendet werden könnte und deren Adaption und Konfigurierung sich durch Automatisierung auf ein für nur wenig geschulte Benutzer erträgliches Maß reduzieren ließe, wäre dieser Hinderungsgrund aufgehoben. Somit ist der Kern dieser Arbeit die Umwandlung und Neustrukturierung des eHome-Entwicklungsprozesses in einen teilweise automatisierten Konfigurierungsprozess.

Es scheint offensichtlich, dass die Software solcher Systeme in Komponenten aufgeteilt werden muss, um wiederverwendbare Einheiten zu finden und neu rekombinieren zu können. Diese Komponenten realisieren dann zu ihrer Laufzeit verschiedene Dienste. Dabei wird in dieser Arbeit zwischen Basis- und Erweiterungsdiensten unterschieden, die durch Komponenten realisiert werden können. *Erweiterungsdienste* nutzen und erweitern das Dienstangebot von Basisdiensten oder anderen Erweiterungsdiensten. Unter den Erweiterungsdiensten befinden sich in der Regel die Dienste, die die Funktio-

nalität anbieten, die für Bewohner interessant ist, also als intelligent bezeichnet werden könnten. Um komponentenbasierte Entwicklung zu unterstützen, ist ein kurzer Blick auf Rahmenwerke, auch in diesem Kontext Middleware-Realisierungen genannt, hilfreich, die dieses Paradigma umsetzen. Die meisten solcher heute verfügbaren komponentenbasierter Middleware-Realisierungen bieten dynamische Komposition, Konfigurierungs- und Deployment-Fähigkeiten an. Für diese Arbeit wurden verschiedene Middleware-Realisierungen betrachtet, um die Softwarekomponenten für einen bestimmten Erweiterungsdienst zu realisieren. Der Fokus richtet sich mittlerweile allerdings darauf, verschiedene Erweiterungsdienste in verschiedenen Heim-Umgebungen zu realisieren. Die Fähigkeiten der betrachteten Middleware-Realisierungen reichen nicht aus, um es dem Endbenutzer zu ermöglichen, ein eHome-System in seiner persönlichen Umgebung einzurichten.

Mit Hilfe des in dieser Arbeit vorgestellten neu strukturierten Prozesses und einer diesen Prozess unterstützenden Werkzeugsuite, der eHomeConfigurator-Werkzeugsuite, wird die eigentliche Entwicklungsarbeit für den Endbenutzer auf die Spezifizierung seiner Umgebung, einiger zu benutzender Erweiterungsdienste und einiger beschreibender Parameter reduziert, ohne Code eingeben zu müssen. Das Ziel dieser Arbeit ist die Unterstützung der Komposition und Konfigurierung der Dienstkomponenten zu eHome-Systemen. Der spezielle Prozess, der innerhalb des Hauses beim Endbenutzer durchgeführt wird, um ein eHome-System aufzusetzen, besteht aus den Phasen Spezifizierung, Konfigurierung und Deployment. Er wird deshalb nach den Anfangsbuchstaben der Englischen Begriffe der Phasen (specifying, configuring und deployment – siehe für eine Begründung der partizipischen Formen die entsprechende Stelle der Nomenklatur in Abschnitt 1.3) eHome-SCD-Prozess genannt.

Die Arbeit zeigt, dass durch den Einsatz sogenannter Funktionalitäten-basierter Komposition und automatischer Konfigurierungstechniken eine signifikante Erleichterung des Entwicklungsaufwandes erzielt werden kann. Sie stellt jedoch auch klar, dass ein vollkommen automatisierter Konfigurierungsprozess nie erreicht werden wird. Dies liegt am menschlichen Faktor, der in die Spezifizierung einwirkt und so ein unvorhersehbares Element in den Prozess bringt.

Getestet und verifiziert wurden die Konzepte und Werkzeuge dieser Arbeit an zwei eigenen Demonstratoren und bei einem externen Kooperationspartner. Die Demonstratoren wurden aus handelsüblichen eHome-Komponenten, einigen selbstmodifizierten Gegenständen und handelsüblichen Lego-Bausteinen realisiert.

Die Anwendung der eHomeConfigurator-Werkzeugsuite in diesen drei Umgebungen bestätigt, dass speziell für die Werkzeugsuite entwickelte Dienste in völlig unterschiedlichen Umgebungen ohne Anpassungen neu eingesetzt werden können. Sowohl der Entwicklungsaufwand vor dem Beginn des eHome-SCD-Prozesses, also die Dienstentwicklung, als auch der Aufwand innerhalb des Prozesses wird durch den Einsatz automatisierbarer Konfigurierung und die interaktive Werkzeugunterstützung erheblich reduziert.

1.2 Leseleitfaden

Kapitel 1 ist die Einführung in diese Arbeit. Es gibt einen groben Überblick über den hier behandelten Themenkomplex und umreißt kurz die zu lösende Problematik und zeigt

die Lösung auf. Ein besonderes Augenmerk ist dabei auf Abschnitt 1.3 zu richten, in dem ein konsistenter Sprachgebrauch für die teilweise stark überladenen Begriffe des Themengebiets angeboten wird.

Kapitel 2 stellt das Szenario vor, in dessen Kontext sich diese Arbeit bewegt. Dabei wird zwischen einem Grob- und einem Feinszenario unterschieden. Das Grobszenario spiegelt das globale Denkmodell wieder, welches dem Schreiben dieser Arbeit zugrunde liegt. Hierunter fällt der Prozess, wie die hier besprochenen Systeme eingerichtet werden, aber auch potentielle Möglichkeiten für einsetzbare Dienste und Geräte. Im Feinszenario werden dann die Umgebungen, Dienste und Geräte vorgestellt, die tatsächlich eingesetzt wurden und zur Evaluierung beigetragen haben.

Kapitel 3 gibt den Stand der Technik wieder. Es werden Forschungs-eHome-Gebäude und der klassische Entwicklungsprozess in diesen vorgestellt.

Im folgenden Kapitel 4 wird der erste Schritt zur Verbesserung des Entwicklungsprozesses für eHome-Systeme durchgeführt. Ein vereinfachter Sicherheitsdienst wird komponentenbasiert in all diesen Rahmenwerken realisiert und wichtige Eigenschaften für die Konfigurierung werden bei der Analyse dieser unterschiedlichen Rahmenwerke für das nächste Kapitel abgeleitet. Die eigentliche Konfigurierung erfolgt aber hier noch manuell im Vorfeld innerhalb des Entwicklungsprozesses.

Kapitel 5 führt das Konzept der generativen Konfigurierung ein und erlaubt somit eine erhebliche Vereinfachung des Entwicklungsprozesses. Hier wird auch ein generisches eHome-Modell eingeführt, welches die Struktur der Konfigurierungsdaten beschreibt.

In Kapitel 6 wird die tatsächliche Umsetzung aller Werkzeuge der eHomeConfigurator-Werkzeug-Suite beschrieben, die das Implementationsergebnis dieser Arbeit darstellt und eine Möglichkeit der generativen komponentenbasierten Programmierung ermöglicht und deshalb den Titel "generative Konfigurierung" trägt.

Kapitel 7 beschreibt den Umgang mit der eHomeConfigurator-Werkzeugsuite und präsentiert einige Ergebnisse, was durch deren Nutzung erreicht werden kann. Es ist unter anderem für diejenigen interessant, die die Werkzeugsuite einsetzen wollen, um selber eHome-Systeme aufzusetzen oder neue Dienste für eHome-Systeme entwickeln wollen. Der Nutzen des Einsatzes der eHomeConfigurator-Werkzeugsuite und der generischen Spezifikatorerzeugung wird in Abschnitt 7.8 an einigen Ergebnissen dargestellt und bewertet.

Das Kapitel 8 gibt einen Überblick über andere Arbeiten, die sich ähnlichen oder verwandten Problemstellungen wie diese Arbeit widmen und unterzieht sie teilweise einer kritischen Begutachtung.

Kapitel 9 fasst die Ergebnisse dieser Arbeit noch einmal zusammen und gibt einen Ausblick auf weiterführende Arbeiten.

1.3 Nomenklatur

Neue Forschungsgebiete bestechen nicht selten damit, dass sie ein Fülle neuer Begriffe prägen und oft diese auch noch mit unterschiedlicher Semantik belegen. So ist es auch auf dem Gebiet der eHome-Systeme. Um dem Leser ein gewisses einheitliches Bild zu ermöglichen, sollen in diesem Abschnitt die wichtigsten Begriffe kurz erläutert werden. Da diese aber auch im Index aufgeführt sind, ist es möglich, diesen Abschnitt zu über-

springen und bei Unklarheiten mittels des Index' wieder hierhin zurückzukehren. Im Folgenden werden meist die englischen Übersetzungen mit angegeben, da im Konzept und Realisierungsteil (Kapitel 4-7) meist auch die englischen Begriffe verwendet werden.

Funktionalität: *Funktionalität* (eng.: *functionality*) leitet sich von dem Wort Funktion ab und soll auch an dieser Stelle dessen Bedeutung im normalen Sprachgebrauch übernehmen. Im Wörterbuch der deutschen Sprache von Gerhard Wahrig, Renate Wahrig-Burfeind [WWB97] heißt es:

> **Funk·ti'on** f.; -, -en **1** *Tätigkeit, Wirksamkeit*; die ~ des Herzens, der Schilddrüse 1.1 in ~ treten *zu arbeiten beginnen, tätig werden* **2** jmd. hat eine ~ *Amt, Aufgabe – innerhalb einer Gemeinschaft* – **3** etwas, ein Maschinenteil hat eine ~ *Zweck* **4** -Math.; Logik- *gesetzmäßige u. eindeutige Zuordnung der Elemente zweier verschiedener Mengen zueinander* 4.1 ~ eines Zeichens -Zeichentheorie- *Zuordnung einer Bedeutung zu einer in Lautzeichen, Buchstaben od. Symbolen dargestellten Form* 4.2 -Kyb.- *aus der Beziehung zwischen Eingabe u. Ausgabe eines dynamischen Systems zu erschließendes Verhalten des Systems*

Diese Arbeit verwendet den Sinn wie folgt: Die Funktion einer Einheit beschreibt den Nutzen oder Zweck dieser Einheit und was mit ihr getan werden kann. Funktion wird hier nicht im Sinne einer Funktion in Programmiersprachen oder in der Mathematik verwendet. Dort sind Funktionen normalerweise Prozeduren bzw. Abbildungen, die einen Wert zurückliefern oder ausrechnen.

Beispiele für Funktionalitäten in dem Fall dieser Arbeit sind:

- `detect switching`: zeigt an, ob ein Schalter gedrückt wurde
- `drive X10`: stellt eine Ansteuerungsmöglichkeit für X10-Geräte zur Verfügung
- `music follows person`: stellt den Dienst zur Verfügung, eine personenspezifische Musikauswahl an der Position in der Wohnung spielen zu lassen, an der sich eine Person gerade befindet und ihr durch die Wohnung zu folgen

Semantisches Etikett: Die Repräsentation einer Funktionalität in einem String, also als Zeichenkette, wird als *semantisches Etikett* (eng.: *semantic label*) bezeichnet. Zu den oben angegebenen Beispielen korrespondierend wären dies hier: `"detect.switching"`, `"drive.x10"`, `"music follows person"`.

Komponente: Die Ausdrücke *Komponente* und *Softwarekomponente* werden in dieser Arbeit synonym verwendet. Eine Komponente ist eine austauschbare Einheit einer Softwareanwendung. Sie kapselt eine definierte Funktionalität in Software ein. Eine Komponente hat eine definierte Import- und eine definierte Exportschnittstelle und sollte konform zu einer informellen Spezifikation, die die umgesetzte Funktionalität beschreibt, sein [Szy02]. Wenn eine Komponente in ihrem Komponentenrahmenwerk installiert und gestartet wurde, kann sie die spezifizierten

Funktionalitäten als Dienst anbieten. Zum Beispiel heißen Komponenten im OSGi-Rahmenwerk *bundles* (siehe für OSGi 4.3). Für die Werkzeug-Suite, die in dieser Arbeit vorgestellt wird, existieren bundles, die die entsprechenden Funktionalitäten implementieren. Ihre Binär-Repräsentation befindet sich in den Dateien: `ehx10-legotoggleswitch.jar` (eine andere Implementation derselben Funktionalität liegt mit der Datei `ehx10switchpanelselector.jar` vor), `ehc10.jar` und `ehmusicfollowsperson.jar`.

Dienst: Die Anzahl verschiedener Definitionen für Dienst (eng.: *service*) ist sogar noch höher als die der Definitionen für Komponenten. Vergleiche dafür folgende Arbeiten:

- [BHM+04, DJMZ05, ACKM03, Cer] Im Bereich der Web-Services ist ein Dienst eine Softwareanwendung, deren Funktionalität über das Netz angesprochen werden kann.
- [Szy02] Hier sind Dienste Softwarekomponenten, die von einem Anbieter ausgeführt werden. Der Anbieter wird nicht für die Softwarekomponenten bezahlt, jedoch für den Dienst, den diese Softwarekomponente erbringt.
- [JW03] In diesem Kontext ist der Dienst eine Menge von Methoden, die über Software von einem Gerät angeboten werden. Hier ist also der Dienst an die Funktionalitäten eines Geräts geknüpft.

Deshalb erhebt die hier genutzte Definition auch nicht den Anspruch, die am meisten akzeptierte zu sein, aber sie kann in dieser Arbeit konsistent genutzt werden.

Ein *Dienst* sei hiermit die abstrakte Sicht einer Komponente bezüglich ihrer zur Laufzeit angebotenen Funktionalitäten. Man kann also die Summe der angebotenen Funktionalitäten einer Softwarekomponente zur Laufzeit als Dienst dieser Komponente verstehen. Ein und derselbe Dienst kann von unterschiedlichen Komponenten zur Verfügung gestellt werden. Der hier vorgestellte Dienstbegriff unterscheidet sich vom Begriff der Funktionalität in [Kir05]. Die Dienste für das angegebene Beispiel heißen in dieser Arbeit: `Lego Switch`, `X10 Ppower Driver` und `Music Follows Person`. Es handelt sich hierbei um die Namen, die in der Dienstbeschreibung (siehe hierfür auch die Implementation in Kapitel 6 auf Seite 145) vergeben wurden. Der Dienst `Lego Switch` benötigt die Funktionalitäten `drive.legodemo` und bietet die Funktionalität `detect.switching` an. `X10 Ppower Driver` bietet die Funktionalität `drive.x10`. `Music Follows Person` erfordert die Funktionalitäten `detect.alarm activation`, `detect.switching`, `detect.person` und `drive.soundroute` und bietet die Funktionalität `music follows person`. Diese Abhängigkeitseigenschaften werden für eines der Hauptkonzepte, der Funktionalitäten-basierten Komposition, verwendet. Für eine genauere Beschreibung dieser und weiterer Beispiele siehe Abschnitt 5.5 auf Seite 142.

Top-level Dienst: Ein top-level Dienst (eng.: *top-level service*) ist ein Dienst, der Funktionalität anbietet, die direkt für den Kunden interessant ist. `Music Follows`

`Person` ist zum Beispiel ein top-level Dienst. Es handelt sich also um die Dienste, die vom Kunden als eHome-Dienste auf einem potentiellen eHome-Dienste-Bestellportal oder über ein Werkzeug, welches vom Anbieter zur Verfügung gestellt wurde, ausgewählt werden können.

Erweiterungsdienst: Ein *Erweiterungsdienst* (eng.: *extension service*) benutzt und kombiniert die Funktionalitäten anderer Dienste, die man in diesem Kontext *Unterdienste* (eng.: sub-services) nennt. Der Erweiterungsdienst bietet auf Grundlage der kombinierten Dienste eine oder mehrere neue Funktionalitäten an und stellt somit eine Erweiterung zu den kombinierten Diensten dar. `Music Follows Person` ist auch ein Erweiterungsdienst, welcher die Funktionalitäten `detect.alarm activation`, `detect.switching`, `detect.person` und `drive.soundroute` erweitert.

Basisdienst/Treiberdienst/Treiber: Dienste, die den direkten Zugriff auf Geräte ermöglichen, werden *Treiberdienst* oder einfach *Treiber* genannt. Dienste, die keine Unterdienste haben, werden *Basisdienste* genannt. Meist sind Treiber Basisdienste, so dass diese Bezeichnung oft synonym verwendet werden kann.

eHome: Als ein *automatisiertes Heim* oder ein *Smart-Home* werden normalerweise Wohnungen mit vernetzten elektronischen Geräten, die über ein Gateway (einen zentralen Rechner) auch von außerhalb des Hauses gesteuert werden, definiert [PSM00, GBV99, JLY04]. Im Gegensatz dazu sollen sich eHomes darüber definieren, dass sie einen oder mehrere der oben genannten top-level Dienste anbieten. eHomes nutzen den Wert, der durch die Kombination verschiedener durch elektronische Geräte angebotener Einzelfunktionalitäten zu einer oder mehreren Gesamtfunktionalitäten entsteht. Der oft zitierte intelligente Kühlschrank [Lev03, PP02] ist in der Tat ein Element eines Smart-Homes. Aber als Einzelanwendung reicht er in diesem Kontext nicht, um eine Wohnung als eHome auszuzeichnen. Erst wenn weitere Dienste, wie zum Beispiel die Verwaltung der lokal gelagerten Güter mit integriert würden, könnte von einem eHome gesprochen werden. Dies könnten externe Dienste für die Güterbestellung oder interne Dienste für die Kontrolle des Energieverbrauchs sein.

eHome-System: Ein *eHome-System* (eng.: *eHome system*) berücksichtigt zusätzlich zur eigentlich automatisierten Wohnung auch alle externen Dienstanbieter und ihre Netzwerke und die Kommunikation zwischen Kunde und Dienstanbieter. Diese Systeme werden näher in [Kir05] betrachtet. Die vorliegende Arbeit befasst sich mehr mit eHomes als mit eHome-Systemen.

eHome-Dienst: Ein *eHome-Dienst* (eng.: *eHome service*) ist ein Synonym für einen hier beschriebenen top-level Dienst.

Spezifizierung und Spezifikation: *Spezifizierung* ist das Substantiv zu der Aktion etwas zu spezifizieren. Etwas zu spezifizieren bedeutet hier, etwas zu beschreiben, zu definieren oder festzulegen. Dies steht im Gegensatz zu *Spezifikation*, welche normalerweise ein Dokument bezeichnet, das Anforderungen beinhaltet. Die in dieser

Arbeit vorgestellte Werkzeugsuite unterstützt die Spezifizierung einer Funktionalitätenhierarchie, am Markt vorhandener Geräte und Im- und Exporten von Dienst-Funktionalitäten durch den Provider. Außerdem unterstützt sie die Spezifizierung der gewünschten top-level Dienste und der physikalischen Wohnumgebung, die vom Kunden durchgeführt werden kann. Eine Spezifikation findet sich im klassischen Entwicklungsprozess für eHome-Systeme als Dokument mit einer Beschreibung der Anforderungen.

Konfigurieren, Konfigurierung, Konfigurationsbeschreibung und Konfiguration: *Konfigurierung* ist der Vorgang, etwas zu konfigurieren. Etwas zu *konfigurieren* bedeutet in diesem Zusammenhang, Relationen und Parameter festzulegen und beschreibende Instanzen für Geräte und Dienste zu erstellen. Die *Konfigurationsbeschreibung* oder synonym nur *Konfiguration* ist dabei das Dokument, welches all diese konfigurierten Informationen speichert. Weiterhin werden in der Konfiguration auch alle spezifizierten Informationen gespeichert. Die Konfiguration ist kein lineares Dokument; es handelt sich hierbei vielmehr um einen Graph. Verschiedene Teile dieses Graphen werden für die verschiedenen Rollen von Bedienern in der Werkzeugsuite verschieden visualisiert. Auch beinhaltet dieses Dokument in unterschiedlichen Konfigurierungsphasen unterschiedlich viel Informationen. Somit variiert die Granularität der Konfiguration über den Konfigurierungsprozess (Abschnitt 5.1 auf Seite 123).

Deployen, Deployment und Deployment-Konfiguration: Deployen beschreibt hier das Installieren (Laden und Binden) und Starten dessen, was in der Konfiguration spezifiziert und konfiguriert wurde. Dies ist die allgemein gängige Definition für deployen. Das Deployment und das Deployen haben in dieser Arbeit dieselbe Bedeutung. Deployment-Konfiguration beschreibt genau den Teil (den Teilgraphen) der Konfiguration, der für das Deployment gebraucht wird. Das heißt, dass das Deployment folgendes beinhaltet: die Installation, die Aktivierung, De-Aktivierung, das Release, die Deinstallation, das Update und die Adaption. Somit enthält die Deployment-Konfiguration alle dafür relevanten Daten.

Kontext: Ein *Kontext* (eng.: *context*) besteht aus einer Menge Fakten, die aus einer bestimmten Sicht heraus benachbart, nahe oder nahe beieinander sind. Der Kontext eines Objektes aus der hier benutzten Konfiguration, also ein Knoten des Konfigurationsgraphen, ist eine Menge benachbarter Knoten. Es ist also eine Menge Knoten, die über Assoziationen bis zu einer bestimmten Tiefe miteinander verbunden sind und den Knoten des zu betrachtenden Kontexts beinhalten. Da der Konfigurationsgraph ein zusammenhängender Graph ist, ist jeder zusammenhängende Teilgraph, der ein bestimmtest Objekt als Knoten enthält, ein Kontext dieses Objekts.

Smart: Der begriff *smart* wird sehr häufig in ubiquitären Systemen eingesetzt: Smart Homes, Smart Devices, Smart Appliances, Smart Objects. Smart mit intelligent zu übersetzen, trifft die Bedeutung des Wortes nicht.[1] Eine Mischung aus "elegant"

[1] Trotzdem wird diese Übersetzung bereits verwendet und auch wieder ins Englische zurückübersetzt. Siehe hierfür auch die kritische Auseinandersetzung mit ambient intelligence in Abschnitt 8.1.1 auf Seite 245.

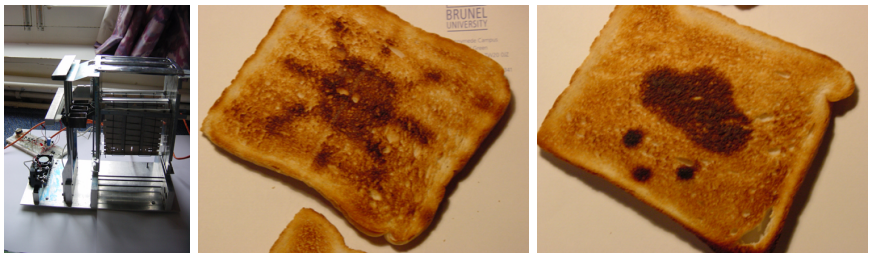

Abbildung 1.1: Wetter-Toaster und seine Ergebnisse (Wetter auf Toast) (Quelle [And])

und "pfiffig" trifft sicherlich eher zu. Die Verwendung von smart im Bereich ubiquitärer Systeme bezeichnet aber immer nur die Möglichkeit ein Gerät über das Netz zu adressieren und zu steuern. Dies entspricht sicherlich nicht dem gewöhnlichen Verständnis des Begriffes "smart" im Englischen oder der Begriffe "elegant" und "pfiffig" im Deutschen. So ist es nicht verwunderlich, dass die Benutzung dieses Begriffes mittlerweile in wissenschaftlichen Kreisen eine gewisse Reserviertheit hervorruft. Oft benutzte Beispiele für smarte Geräte sind der Smart-Fridge, der Smart-Pencil oder Smart-Clothing. Im Deutschen werden diese Geräte dann auch noch mit intelligenter Kühlschrank, intelligenter Stift oder intelligenter Kleidung übersetzt, was die Schieflage der möglichen Interpretationen noch verstärkt. All diese Geräte sind eigenständige Geräte. Sie können zwar mit ihrer Umgebung kommunizieren, aber ihre angebotenen Dienste sind in ihnen selbst implementiert und unabhängig von anderen Geräten. Der Kühlschrank kann Buch über die in ihm bewahrten Lebensmittel führen und gegebenenfalls neue Lebensmittel bestellen. Der Stift kann aufnehmen, was mit ihm geschrieben wurde. Die Kleidung kann Gespräche aufzeichnen oder speichern an welchen Orten sie gewesen ist. All diese Geräte besitzen eine gewisse Attraktivität und sind schöne technische Spielzeuge mit einem gewissen Geek-Faktor[2]. Aber sind sie wirklich smart oder gar intelligent? Werden sie den Alltag verändern oder sind sie nur spezielle Luxusgegenstände? Schon interessanter sind Geräte wie der smarte Toaster (Robin Southgate, Brunel-University, siehe Abbildung 1.1) oder der smarte Regenschirm, da sie eine Software benötigen, die Informationen aus unterschiedlichen Bereichen kombinieren muss. Der Toaster kann die Wettervorhersage als Bild am Morgen auf den Toast drucken und der Regenschirm kann seinen Besitzer daran erinnern, ihn mitzunehmen, wenn das Wetter schlecht angesagt ist, bzw. ihn nicht stehen zu lassen, wenn der Besitzer gerade bei jemanden zu Besuch ist und aufbricht. Diese Geräte benötigen andere Dienste, um einen Erweiterungsdienst anbieten zu können. Sie können also smartere Dienste durch die Zusammenarbeit mit anderen Geräten bzw. anderen Diensten anbieten. Eine Hoffnung dieser Arbeit ist es, dass durch die Ermöglichung die-

[2]Geek-Faktor ist ein im Computerjargon gebrauchtes Wort und bezeichnet eine technische Neuerung, die besonders klein, kompliziert oder auf besondere Weise technisch verspielt ist. Beispiele heutzutage sind zum Beispiel ein PDA eingebaut in eine Armbanduhr, ein vollwertiger PC so groß wie eine Zigarettenschachtel. Auch iPods haben einen solchen Geek-Faktor wegen der schönen Kombination aus Technik und Design.

ser Zusammenarbeit und Integration verschiedener Dienste auf einer breiten Basis, überhaupt erst das smarte Potential smarter Applikation ermöglicht wird.

Intelligent: Wie schon angedeutet, wird anstelle von smart insbesondere im Deutschen häufig auch das Wort *intelligent* eingesetzt. Dies legt oft den Vergleich mit künstlicher Intelligenz nahe, der aber völlig in die falsche Richtung führt. Es gibt zwar einige Applikationen im Bereich von eHome-Systemen, die auf neuronalen Netzwerken beruhen [Moz98], doch geht es normalerweise bei der Benutzung des Intelligenzbegriffes hier um sinnvolle oder intuitive Unterstützung des Benutzers im Alltagsleben.

Umgebungsintelligenz/Ambient Intelligence: Ambient Intelligence, deutsch auch Umgebungsintelligenz genannt, ist eines der neuen Stichwörter, in denen sich die Bereiche Ubiquitous Computing und Pervasive Computing widerspiegeln. Es wird insbesondere im Zusammenhang mit dem europäischen Forschungsprogramm "Information Society Technologies" benutzt.

Auch Begriffe wie "Next Generation Media", "Wearable Computing" und "Organic Computing" fallen in diesen Bereich. Genau wie beim eHome-System soll es Ziel der Forschungsanstrengungen sein, Sensoren, Funkmodule und Computerprozessoren massiv zu vernetzen, um so den Alltag zu verbessern. Für eine kritische Betrachtung dieses Begriffes und Forschung in diesem Bereich siehe auch Abschnitt 8.1.1 auf Seite 245.

1.4 Problemfestlegung

eHomes sind nicht mehr bloß eine Vision. Es gibt prototypische Realisierungen, die verschiedene experimentelle eHome-Dienste implementieren und erfolgreich einsetzen [inH05, TS05, Phi02] (siehe für eine genauere Beschreibung auch Abschnitt 2.1.2 auf Seite 19). Es gibt auch einige private Projekte, die ein eHome-System eingerichtet haben [Sch05a, Lau04]. Aber die benutzten Techniken sind nicht ausgereift genug, um eine breite Klientel anzusprechen. Auch wenn es bereits viele elektronisch steuerbare Geräte und einige beispielhafte Anwendungen für diese gibt, ist es immer noch sehr schwierig ein komplexes und integriertes System aus diesen aufzubauen. Gründe dafür sind unter anderem die mangelnde Kompatibilität der eingesetzten Geräte sowie der verfügbaren Software-Komponenten. Bis heute gibt es keinen akzeptierten Standard im Bereich der eHome-Gerätekommunikation. Die meisten Hardware-Anbieter erschaffen immer wieder neue eigene Protokolle, die in der Regel auch nicht offen liegen. Sogar die physikalische Konnektivität ist oft nicht gegeben. Auch die Kommunikation zwischen Anwendung und Geräten, die vorher aufeinander abgestimmt wurden, führen in der Praxis oft zu unerwarteten Ergebnissen[3]. Im Prinzip sind die notwendigen Techniken, Geräte und Anwendungen für funktionierende eHome-Systeme am Markt vorhanden, aber deren Kombination und Konfigurierung für den Alltagsgebrauch ist bisher noch zu komplex

[3]Nicht selten verhält sich neu angeschlossene Peripherie (insbesondere USB-Geräte) anders als erwartet. Webcams nehmen keine Bilder in der höchsten Auflösung auf. Die Farbgebung beim Scanner ist nicht wie erwartet oder der Inhalt einer DVD kann nur mit ganz bestimmten Grafiktreibern flüssig wiedergegeben werden.

und zu kostenintensiv. Um funktionierende funktionierende eHome-Systeme für einen breiten Markt zu ermöglichen, bedarf es bisher noch eines komplexen Entwicklungsprozesses, in dem Komponenten konfiguriert oder gar neu entwickelt werden und in die entsprechende physische und virtuell lokale Ausführungsumgebung eingebettet werden müssen. Während dieser Konfigurierung müssen eine Menge Informationen und Parameter berücksichtigt werden: Abhängigkeiten, Versionen, Kompatibilität, Verfügbarkeit von Softwarekomponenten, aber auch Orts- und Adressinformationen. Die Fülle dieser Informationen lässt sich nur schwer verwalten. Deshalb liegen die Fähigkeiten real verfügbarer eHome-Dienste (wie Alarm-Systeme oder Beleuchtungssteuerungen) weit hinter den Möglichkeiten in den Beispielimplementierungen wie den oben genannten zurück. Die verfügbaren Dienste sind in der Regel keine Erweiterungsdienste und lassen sich auch nur selten zu neuen Diensten erweitern. Meist sind die Dienste komplett in einem Gerät gekapselt. Beispiele hierfür sind Hifi-Systeme, in die ein Weckdienst integriert ist, also mit einfacher Weckerfunktionalität, die es ermöglicht Musik statt eines Alarms beim Aufwecken wiederzugeben. Auch der intelligente Kühlschrank [Lev03, PP02] ist ein solches Beispiel. Lösungen (wie Alarm-Systeme), die aus mehreren Einheiten bestehen, sind normalerweise komplett von einem Hersteller und nicht kombinier- oder aufrüstbar mit oder durch Produkte anderer Hersteller. Sie sind auf proprietäre Standards des Herstellers angewiesen und bereits im Vorfeld miteinander harmonisiert. Änderungen oder Erweiterungen solcher vorkonfigurierter Lösungen sind praktisch unmöglich. eHome-Dienste sollten verschiedene Produkte unterschiedlicher Hersteller bedienen können. Durch die Integration der angebotenen Funktionalitäten werden neue und komplexe Funktionalitäten realisierbar und können dem Kunden so einen höheren Wert anbieten als Basisdienste.

Eine weitere Forderung ist die nach der Fähigkeit, eine durch einen Dienst angebotene Funktionalität mehrfach benutzen zu können. Zum Beispiel kann eine Kamera in einem Sicherheitsdienst sowohl zur Bewegungserkennung und Fotoüberwachung benutzt werden als auch in einer anderen Situation als Überwachungskamera für ein Baby, um dieses von außerhalb zu beobachten, eingesetzt werden. Es muss also möglich sein, installierte Dienste um Funktionalitäten zu erweitern, indem man diese in einfacher Weise in neue Diensten integriert. Diese Erweiterung sollte von einfachen Mechanismen zur Erkennung angeschlossener Geräte – wie im plug-and-play-Paradigma [UPn00] beschrieben – unterstützt werden. Solche Anforderungen spielen eine besondere Rolle in der Verbreitung von eHome-Diensten in Millionen unterschiedlicher Haushalte. Um mehr Flexibilität und zahlreiche Erweiterungen in eHome-Systemen zu ermöglichen, muss der Entwicklungsprozess zur Realisierung von eHome-Diensten für beliebige eHomes vereinfacht werden. Der typische Ansatz, der in der Softwaretechnik dafür verfolgt wird, ist die Werkzeugunterstützung dieses Prozesses. Solche Werkzeuge müssen den Dienstinstallateur dabei unterstützen, die Grundeinrichtung der entsprechenden Software nach der Anbringung der Geräte durchzuführen. Weiterhin sollten die Werkzeuge die Einrichtung der logischen Repräsentation der Umgebung unterstützen, so dass diese vom eHome-System verstanden und ausgewertet werden kann. Im optimalen Fall ist der Installateur nicht nötig und der Kunde bzw. die Bewohner sind selbst in der Lage, die Grundeinrichtung durchzuführen.

Also ist eines der Hauptziele dieser Arbeit die Vereinfachung des Entwicklungsprozesses und der Konfigurationserstellung. Diese Aufgabe beinhaltet die Kombination der benötigten Geräte und Anwendungen. Solchen Werkzeugen muss eine breite Wissensbasis zugrunde liegen, die das Inferenzieren (im Sinne von Ableiten von Informationen)

von Funktionalitäten und Integrationsregeln erlaubt, um herauszufinden, wie mit Hilfe welcher Funktionalitäten andere Funktionalitäten realisiert werden können, wie Abhängigkeiten aufgelöst werden können und welche Geräte benötigt werden. Natürlich muss diese Information in einer Weise abgespeichert werden, dass alle benötigten Parameter und abgeleiteten Informationen dort wieder gespeichert werden können. Das hat zur Folge, dass diese Wissensbasis sehr schnell wachsen wird, insbesondere je mehr Geräte und Funktionalitäten unterstützt werden. Diese stetig wachsende Menge von Informationen und voraussichtlich auch das Auftauchen von neu strukturierten Informationen erfordern einen Blick auf die Anpassbarkeit der Veränderbarkeit der Wissensbasis und auch ihrer Struktur. Solche Änderungen bedingen normalerweise schwerwiegende und fehleranfällige Kodierungsarbeiten, die es zu verhindern gilt.

1.5 Zielsetzung

Bisher wurde im Bereich der eHome-Systeme für jede neu einzurichtende eHome-Umgebung ein kompletter Entwicklungsprozess durchgeführt. Dies führte zu sehr hohen Entwicklungskosten und einer sehr niedrigen Verbreitung von eHome-Systemen.

Abstraktionen, die eine Wiederverwendung von Komponenten in einem anderen Umfeld ermöglichten, gab es bisher nur in ganz speziellen Domänen und nicht allgemein für eHome-Systeme. Auch die Kopplung der Konfigurationsbeschreibung des gesamten eHome-Software-Systems an die physikalischen Eigenschaften des eHomes war bisher nicht vorhanden.

Im ersten Schritt muss der Entwicklungsprozess an eine Entwicklung für einen großen, vielfältigen Wohnungsmarkt umgestaltet werden, was in der Dissertation von Herrn Michael Kirchhof [Kir05] teilweise bereits geschah. Relevant für die vorliegende Arbeit ist das Herauslösen des Entwicklungsanteils durch ein spezielles Kommunikationsmodell und eine besondere Art der Programmierung der Softwarekomponenten der Dienste.

Zweitens müssen die Vielfalt der Parameter, Softwareabhängigkeiten und physikalischen Ortsinformationen berücksichtigt werden, da dies immer noch für jedes neue eHome-System konfiguriert werden muss. Somit ist ein erstes wesentliches Ziel dieser Arbeit die Schaffung eines Modells, welches die Konfigurationsbeschreibung innerhalb des gesamten Entwicklungsprozesses eines eHomes und auch später zur Laufzeit beschreiben kann. Auch die Inferierung von physisch lokalen Kontexten in einer Instanz dieses Modells ist ein wichtiger Beitrag zur Realisierung von eHome-Diensten. Da die physikalischen Informationen erweiterbar sein müssen, und auch einige mögliche Gerätezusammenhänge im Vorfeld nicht erahnt werden können, muss dieses Modell ähnlich einer Wissensbasis von der Struktur her sehr flexibel und leicht erweiterbar sein. Strukturelle Änderungen des Modells sollen keinen großen Aufwand in der Programmierung der unterstützenden Werkzeuge hervorrufen.

Drittens muss es möglich sein, Dienste zu neuen Diensten mit mehr Funktionalität zu kombinieren. Es ist also ein Mechanismus erforderlich, der die Kombination von Funktionalitäten und deren Wiederverwendung unterstützt.

Viertens bedarf es einer Werkzeugunterstützung der Phasen Spezifikation, Konfigurierung und Deployment, um das Modell entsprechend dieser Phasen zu bearbeiten. Das

Ergebnis dieser Werkzeugunterstützung ist die Werkzeugsuite eHomeConfigurator. Die Spezifikatoren dafür werden generisch aus dem Modell gewonnen und garantieren somit auch eine leichte Wartung bei Modelländerungen, die insbesondere bei der Anpassung auf zukünftige Umgebungserweiterungen zu mehreren gleichzeitig betrachteten eHomes nötig sein werden. Die Konfigurierungsunterstützung nutzt Erkenntnisse aus dem Konfigurationsmanagement und wendet sie in der Domäne der eHome-Systeme neu an. Sie werden allerdings um einen Ansatz zur semantisch-basierten Komposition (oder hier Funktionalitäten-basierten Komposition genannten) ergänzt und ermöglichen somit eine teilweise automatisierte Konfigurierung, was völlig neu im Bereich der eHome-Systeme ist. Für das Deployment muss ein Werkzeug entwickelt werden, das eine Instanz des beschriebenen Modells bei installierten Geräten auf einem Komponentenrahmenwerk automatisch in Betrieb nehmen kann.

1.6 Lösung

Um diese Zielsetzung zu erfüllen, wird zunächst das Arbeitsszenario festgelegt (Kapitel 2). Somit wird die Sichtweise der Arbeitsgruppe, der diese Arbeit zugeordnet ist, herausgearbeitet. Bei der groben Sicht ist es wichtig, festzustellen, dass es sich bei dem Einrichten von eHome-Systemen um eine Kunde-zu-Anbieter-Beziehung handelt. Der Kunde ist dabei ein (oder mehrere) Bewohner einer Wohnung, der den Wunsch hat, eHome-Dienste in seiner (ihrer) Wohnung in Anspruch zu nehmen. Der Anbieter (eng.: Provider) ist ein Hardwarehersteller oder ein Dienstleistungsunternehmen, welche eHome-System-Einrichtungen vermarkten. Es wird hier ein Dienst- bzw. Nutzen-getriebenes Verhalten des Kunden angenommen. Es geht also nicht darum, einzelne technisch hoch-integrierte oder formschöne Geräte einzurichten. Vielmehr ist die Kombination verschiedener Gerätefunktionalitäten zu neuen Funktionalitäten angestrebt, die die Einzelfunktionalitäten ergänzen bzw. übertreffen und somit für den Kunden einen wünschenswerten Mehrwert darstellen. Der Provider bietet dem Kunden eine Auswahl von Diensten an, aus denen dieser gewünschte Dienste wählen kann. Diese können nach erfolgter Einrichtung in der Wohnung des Kunden die gewünschten Funktionalitäten anbieten und ausführen. Ein unmittelbares Problem, welches sich aus dieser groben Sicht ergibt, ist das Anbieten von Diensten unabhängig von allen möglichen physikalischen Umgebungen, die bei unterschiedlichen Kunden zu erwarten sind.

In einer Verfeinerung des Szenarios wird zuerst auf den der eHome-System-Entwicklung zugrunde liegenden extrem kostenintensiven Softwareentwicklungsprozess aufmerksam gemacht. Weiterhin werden der Aufbau eines generellen eHomes und verschiedene mögliche Dienste eines solchen eHome-Systems besprochen. In einer weiteren Verfeinerung werden sechs spezielle Dienste (siehe Abschnitt 7.4.1) für die in dieser Arbeit durchzuführenden Untersuchungen als Arbeits- bzw. Feinszenario ausgewählt. Diese berühren die einfache Beleuchtungssteuerung bis hin zu komplexen Sicherheits- und Multimediadiensten. Um verschiedene Umgebungen testen zu können, wird die Entwicklung an zwei Demonstratoren und bei einem Kooperationspartner in einem echten Haus getestet. Die Analyse der Vielfältigkeit der möglichen Dienstkombinationen und unterschiedlichen Umgebungen in diesem Kapitel suggeriert, dass es für eine einheitliche Verwaltung ein Modell geben muss, welches die Umgebungsinformationen, Komponentenabhängig-

keiten und angebotenen Funktionalitäten in einen strukturierten Bezug setzen muss und so in einer Instanz deren Speicherung und Verwaltung erlaubt.

In der anschließenden Untersuchung des Stands der Technik (siehe Kapitel 3) wird betrachtet, was im Sinne von Erweiterungsdiensten bereits am Markt vorhanden ist und wie weit Forschungseinrichtungen für ubiquitäre und eHome-Systeme im Bereich der Entwicklungsprozessunterstützung bereits sind. Die Untersuchung zeigt, dass es zwar durchaus hochintegrierte Geräte mit einem großen Funktionalitätsangebot gibt, diese sich aber normalerweise nicht zu neuen Diensten kombinieren lassen. Auch steht in den betrachteten Forschungseinrichtungen eine solche Dienstkombination nicht im Vordergrund. Vereinzelt werden solche Kombinationen und Erweiterungen speziell für die betrachtete Umgebung entwickelt. Der Gedanke der Unterstützung der Entwicklung durch Wiederverwendung wird nur im inHaus (siehe Abschnitt 3.2.2) durch den Einsatz von komponentenbasierter Entwicklung angesprochen. Doch werden auch dort die Dienstkombination und die Übertragung auf andere Umgebungen nicht betrachtet.

Deshalb befasst sich Kapitel 4 auch mit der Analyse der Dienstentwicklung in unterschiedlichen Rahmenwerken zur komponentenbasierten Programmierung, versucht also den Ansatz, der im inHaus verfolgt wird, weiterzudenken und weitere Punkte zur Entwicklungsprozessunterstützung und insbesondere auch weitere Eingaben und Bedingungen für die Erstellung des gerade angesprochenen Modells zu finden. Dabei wird ein vereinfachter Sicherheitsdienst (siehe Abschnitt 2.2.1) in allen drei Rahmenwerken implementiert. Insbesondere wird untersucht, wie Konfigurierung und Deployment in diesen Rahmenwerken funktionieren. Da die Rahmenwerke alle fast keine, bzw. Openwings nur minimale, Unterstützung für automatisierte oder werkzeugunterstützte Konfigurierung mitbringen, werden hier die manuelle Konfigurierung und ihre mögliche Automatisierung untersucht. Weiterhin werden die verschiedenen Implementationen in den unterschiedlichen Rahmenwerken dafür genutzt, einen Eindruck über den Aufwand der komponentenbasierten Entwicklung mit manueller Konfigurierung zu gewinnen.

Der nächste Schritt, dargelegt in Kapitel 5, ist die Konzeptionalisierung eines unterstützten und somit vereinfachten Entwicklungsprozesses für eHome-Systeme. Dafür wird in den sich immer wiederholenden klassischen Entwicklungsprozess ein sogenannter Spezifizierungs-, Konfigurierungs- und Deployment-Prozess (SCD-Prozess) eingeführt, der erheblich leichter zu handhaben ist und somit den sich wiederholenden Teil in einen durch Werkzeuge unterstützbaren Teil überführt. Der nun zeitlich vor diesem Prozess liegende Entwicklungsanteil für die wiederverwendbaren Dienste ist dann nicht mehr repetitiv und wird erheblich seltener ausgeführt. Er muss nur bei Anpassungen an neue Hardware oder der Entwicklung von neuen Diensten und Updates ausgeführt werden. Weiterhin wird das unterstützende Modell für diesen Prozess in Kapitel 5 konzipiert. Die hier beschriebene Konfigurierungsunterstützung in einem SCD-Prozess wird generative Konfigurierung genannt.

Im folgenden Kapitel 6 werden die gerade besprochenen Konzepte in einer Werkzeugsuite, dem eHomeConfigurator, implementiert. Die Implementation der einzelnen Werkzeuge, insbesondere des Spezifizierungs-, des Konfigurierungs- und des Deploymentwerkzeugs, werden hier beschrieben. Für die Benutzung des Modells in diesen Werkzeugen wird ein modellgetriebener Entwicklungsansatz (Model Driven Development [KWB03, Obj04, MBB02, Zün05, DGZ04]) verwendet, so dass ein Großteil der Implementierung direkt aus dem Modell erzeugt werden kann.

In Kapitel 7 werden sowohl die Benutzung der eHomeConfigurator-Werkzeugsuite als auch die Programmierung neuer Dienste für diese an Beispielen vorgestellt. Das Kapitel endet mit einem Vergleich der hier benutzten generativen Konfigurierung von eHome-Systemen mit dem Stand der Technik und bewertet den Effekt der eHomeConfigurator-Werkzeugsuite auf die Entwicklung von eHome-Systemen. Die Bewertung kommt zu dem Schluss, dass durch den Einsatz der in dieser Arbeit besprochenen Konzepte mit dieser Werkzeugsuite eine signifikante Vereinfachung der eHome-System-Softwareentwicklung insbesondere in Hinblick auf den Einsatz in unterschiedlichsten Umgebungen erreicht werden kann.

1.7 Wissenschaftlicher Beitrag

Wie in diesem Kapitel bereits dargestellt wurde, besteht bei der Entwicklung von eHome-Systemen das Problem eines immer wiederkehrenden sehr kostenaufwändigen Software-Entwicklungsprozesses. Insbesondere gestaltet sich die Wiederverwendung von Software in anderen Umgebungen zur Bereitstellung derselben Funktionalitäten als extrem schwierig.

In dieser Arbeit werden Methoden der Softwaretechnik angewandt, um dieses Problem zu lösen. Insbesondere sind dies strukturierte Modellierung, komponentenbasierte Programmierung und viele Aspekte des Konfigurationsmanagements. Es wird ein Modell konstruiert, welches in der Lage ist, die Softwarekonfiguration für beliebige eHome-Systeme im gesamten Lebenszyklus der Nutzung dieser Systeme zu beschreiben. Dieses Modell lässt sich auch auf andere Systeme, wie sie zum Beispiel in der Automobilbranche auftreten, übertragen.

Dieses Modell und die auf ihm operierenden Werkzeuge benutzen weiterhin semantische Beschreibungstechniken, um die Komponentenkomposition zu vereinfachen. Sie schaffen somit neue Zugänge zur generativen komponentenbasierten Programmierung, hier generative Konfigurierung genannt.

Auf Basis des Modells werden Werkzeuge konzipiert und prototypisch umgesetzt. Es wird nachgewiesen, dass die Kombination des Modells und der darauf operierenden Werkzeuge tatsächlich den Entwicklungsprozess für eHome-Systeme erheblich vereinfacht.

Bei der Implementation der Werkzeuge wird für das Spezifikationswerkzeug ein Verfahren zur modellgetriebenen Programmentwicklung genutzt. Die erstellte Werkzeugsuite wird erfolgreich an den Demonstratoren eingesetzt, und die Vereinfachung, die durch diesen modellgetriebenen Ansatz erzielt wird, wird nachgewiesen. Deshalb leistet die Arbeit auch einen Beitrag zur Forschung über modellgetriebene Softwareentwicklung.

Auch die Tauglichkeit der Visualisierung der Konfiguration eines eHome-Systems anhand verschiedener Graphsichten wird durch kurze Bedienzeiten der befragten Anwender bestätigt.

Die Idee, eine generische Kombination von Diensten zu neuen Diensten in mehr als zwei Schichten zu erlauben, und die Realisierung einer funktionierenden Konfigurierungsunterstützung des Entwicklungsprozesses ist so im Bereich der eHome-Systeme in verwandten Arbeiten nicht zu finden. Vielleicht lassen sich dadurch zukünftige eHome-System-Entwicklungen vereinfachen.

Kapitel 2

Szenario

Dieses Kapitel gibt eine Vorstellung der zugrunde liegenden Idee dieser Arbeit. Über das Grobszenario wird eine Einordnung möglich, in welchen Bereichen die in dieser Arbeit vorgestellten Techniken relevant sind. Auch wird der angenommene Prozess der Einrichtung eines neuen eHomes vorgestellt, der dieser Arbeit zugrunde liegt.

Im zweiten Teil dieses Kapitels werden im Feinszenario die Teile vorgestellt die tatsächlich untersucht wurden. Es werden des Weiteren die speziellen Dienste vorgestellt, die realisiert wurden und auch die Umgebungen, in denen diese getestet wurden.

2.1 Grobszenario

Das folgende Szenario wird in dieser Arbeit als typisch für das Einrichten neuer eHomes angenommen:

Am Anfang steht der Wunsch eines Hausbewohners bzw. Kunden, einen eHome-Dienst in seinen Räumen zu installieren. Präziser heißt dies, dass der Kunde angibt, welche Funktionalitäten ein bei ihm einzurichtendes eHome-System anbieten soll. Hier wird nicht angenommen wird, dass der Kunde besondere Geräte auswählt und dann darüber nachdenkt, was er mit diesen tun soll, sondern er wählt eine oder mehrere Funktionalitäten, die er gerne als Dienst in seiner Wohnung erfüllt sehen möchte. Weiter wird angenommen, dass der Kunde hier entsprechend eines Wunsches eine bewusste und vernunftgesteuerte Auswahl trifft.

Er wählt also entsprechend gewünschten Nutzens, den er sich von der Einrichtung des eHome-Systems verspricht. Ein solches Kaufverhalten deckt sich bei der Einführung neuer Produkte in den Markt mit der einschlägigen Literatur: Vergleiche hierfür auch Seite 252 bis 257 in [KA91]. Die bisherige Vorgehensweise bei der Einführung von eHome-Systemen in den Markt setzt mehr darauf, optisch ansprechende Einzelgeräte zu verkaufen und darauf zu hoffen, dass diese irgendwie mit anderen Einzelgeräten kommunizieren können und sich daraus automatisch ein Mehrwert ergibt. In einer grundlegenden Annahme, wird in dieser Arbeit davon ausgegangen, dass diese Vermutung falsch ist und der Kunde lieber, wie auch in der klassischen Marketing-Literatur gezeigt, einen für ihn erkennbaren Nutzen wählen wird, als ein Produkt des Produktes wegen zu erwerben. Natür-

lich ist die Vernetzungsfähigkeit der angebotenen Geräte eine Grundvoraussetzung dafür, aus ihnen mit der Hilfe von Software neue Funktionalität zu kombinieren. Dennoch muss es das Ziel sein, diese kombinierte Funktionalität, die durch einen Erweiterungsdienst angeboten wird, dem Kunden anzubieten, ihm benötigte Geräte und Software automatisch vorzuschlagen und ihm entsprechend seiner Wahl einen Großteil der Konfigurierung und Installation abzunehmen.

Um die Wahl dieser Funktionalität durchzuführen, wird der Kunde einen Provider entweder direkt oder indirekt mittels eines Softwarewerkzeugs kontaktieren, welches ihn bei der Auswahl der top-Level Dienste bzw. der von diesen angebotenen Funktionalitäten unterstützt. Der Provider bietet hierbei Hardware, Software, und Konfigurierungssupport an. In dem hier gedachten Fall, wird der Kunde mittels eines Konfigurierungswerkzeugs, welches er vom Provider zur Verfügung gestellt bekommt, die Dienste auswählen, die die gewünschten Funktionalitäten zur Verfügung stellen. Die Dienste sind auf einem abstrakten Level beschrieben und klassifiziert, so dass eine intuitive Auswahl möglich wird. Das Werkzeug ermittelt Geräte, die zur Umsetzung der gewählten Dienste benötigt werden, und schlägt diese vor. Dabei werden vorhandene Geräte und vorhandene Infrastruktur berücksichtigt. Nach der Anbringung der benötigten Geräte unterstützt das Werkzeug den Konfigurierungsprozess und kann das Deployment automatisch durchführen und das eHome-System damit in Betrieb nehmen.

Wie bereits in Kapitel 1.3 dargestellt, unterscheidet die Arbeit zwischen Basis- und Erweiterungsdiensten. Erweiterungsdienste benutzen Funktionalität von Unter- oder Basisdiensten, um neue Funktionalität anzubieten, während Basisdienste die direkte Steuerung von physikalischen oder auch nur virtuellen Geräten (wie E-Mail-Kommunikation oder SMS-Versand) anbieten. Dabei richtet sich der Fokus der Arbeit besonders auf die Erweiterungsdienste, da es diese sind, an denen der Kunde normalerweise interessiert ist.

Es sollte zusätzlich bedacht werden, dass die ausgewählten Dienste auch in unterschiedlichen Umgebungen installiert werden können müssen. Insbesondere die Vielfalt verschiedener Raumkonstellationen oder Gerätekombinationen muss berücksichtigt werden.

2.1.1 Entwicklungsprozesse

Bei den bisherigen eHome-Systemen wurde immer für jedes neu einzurichtende eHome-System ein kompletter Entwicklungsprozess durchgeführt. Das heißt für jede neue Umgebung war ein hoher Implementierungsaufwand zu leisten, um eHome-Dienste, also insbesondere Dienste, die verschiedene Funktionalitäten zu neuen erweitern, in dieser Umgebung zu realisieren. Es existieren zwar Ansätze, durch Einsatz von Software-Komponenten den Aufwand zu reduzieren, doch erfordert auch das Zusammenfügen der Komponenten immer noch einen zu hohen und zu kostenintensiven Aufwand. Dieser entsteht durch die Programmierung des Glue-Codes bzw. durch das Ausprogrammieren der Konnektoren, über die die Komponenten miteinander verbunden sind.

Ziel muss es also sein, den Entwicklungsaufwand aus dem Enwicklungsprozess für jede neue Umgebung zu entfernen und im Vorfeld nur noch einmal durchzuführen. Aber auch das Zusammenfügen der dann im Vorfeld entwickelten Komponenten darf nicht zu komplex sein, sondern muss mit einem sehr geringen Aufwand, wenn nicht sogar ganz automatisch durchgeführt werden.

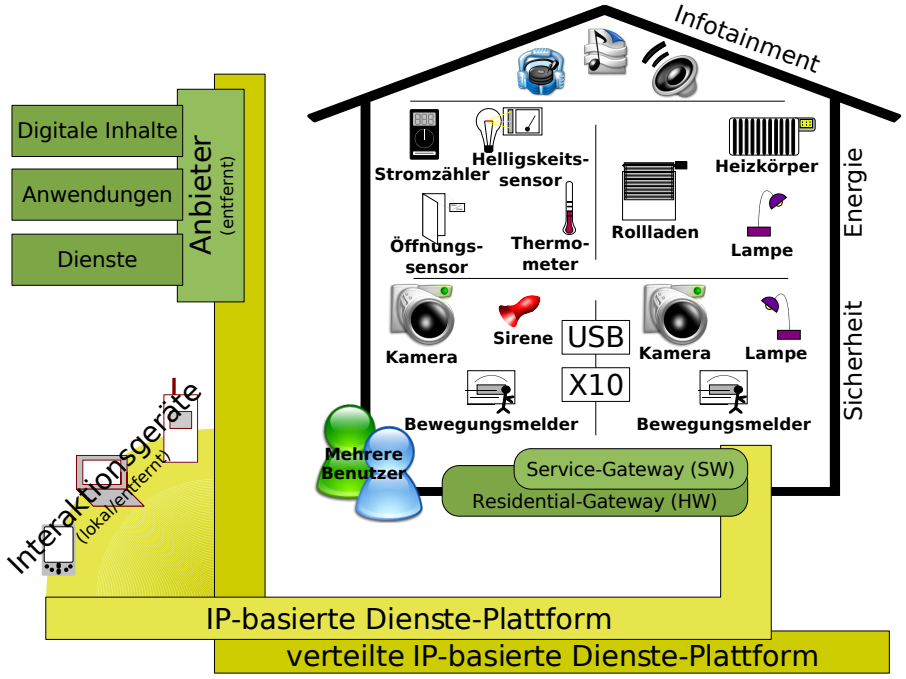

Abbildung 2.1: Übersicht über ein eHome-Systems

2.1.2 eHomes und eHome-Dienste

Dieser Abschnitt soll eine Idee geben, wie eHomes bzw. eHome-Systeme aussehen können und was mit ihnen geleistet werden kann. Abbildung 2.1 gibt eine Übersicht über ein eHome-System. Das eHome-System beinhaltet alle Hardware, Software und unterstützenden Systeme, welche die Dienste für einen oder mehrere (Multi-User) Bewohner anbieten. Es spiegelt die Sichtweise eines eHome-Systems für diese Arbeit wieder und kann somit als allgemeines Leitszenario angesehen werden.

Die Abbildung ist gegliedert in die Bereiche Anbieter (Dienstanbieter), Dienste-Plattform, Service-Gateway, Residential-Gateway, Dienste, Geräte und deren Anschlusstechniken.

Der Provider ist verantwortlich für die Lieferung von Leistungen wie die Installation von Diensten, die Wartung von Diensten oder die Versorgung mit Informationen wie zum Beispiel Nachrichten. Er ist die zentrale Anlaufstelle für Neu- und Bestandskunden und bedient sich dabei einer IP-basierten Service-Plattform, die es ihm ermöglicht, seine Leistungen aus der Ferne zu erbringen. Dadurch wird der Personalaufwand für den Betrieb eines eHomes gesenkt und gleichzeitig die Grundlage für neuartige Dienste, die sich externer Informationsquellen bedienen, geschaffen. Über diese Plattform soll ein Kunde einen Einstiegspunkt für die Planung seines eHomes erhalten. Nach erfolgter Einrich-

tung dient sie zur Versorgung des Kunden mit Informationen und zur Kommunikation des eHomes mit dem Provider.

Auf Kundenseite stellt ein Service-Gateway die zentrale Schnittstelle zwischen eHome und Internet und somit dem Provider dar. Das Service-Gateway soll es dem Provider ermöglichen, dem Kunden Dienste anzubieten, ohne ihn mit der damit verbundenen Komplexität zu konfrontieren. Das Service-Gateway ist eine Software und stellt die Zielplattform für die Konfigurationsinformationen, die in dieser Arbeit erzeugt werden, dar. Das Service-Gateway aggregiert die im eHome anfallenden Informationen und stellt sie den auf ihm installierten Diensten bereit. Diese Software wird auf einer Hardware-Plattform betrieben, dem Residential-Gateway.

Plattform (engl. *platform*) ist ein weit gefasster Begriff, der eine Hard- oder Software bezeichnet, die eine definierte Umgebung für weitere Komponenten bereitstellt. Zu den Hardwareplattformen zählen beispielsweise Standard-PCs auf x86 oder Apple-Basis, oder kleinere, in sich abgeschlossene Geräte wie Router, Set-Top-Boxen, PDAs und Mobiltelefone. Speziell für den eHome-Bereich sind die Hardwareplattformen der Geräte von Interesse. Dazu zählen der Europäische Installations-Bus (EIB) [Bec01, BPR00], das Local Operating Network (LON) [Ech02] und das Schalt-Protokoll X10 [X1004]. Zu den Softwareplattformen werden primär Betriebssysteme wie Linux, Windows, Mac OS, Palm OS oder Windows CE gerechnet. Auf Grundlage dieser laufen die meisten übergeordneten Softwareplattformen, wie die Java Virtual Machine, die eine Laufzeitumgebung für Java-Programme bereitstellt, oder Rahmenwerke für komponentenbasierte Software wie Enterprise Java Beans (EJB) [Sun03], .NET Enterprise Services [Mic03], Common Object Request Broker Architecture (CORBA) [Obj02]. Diese stellen spezialisiertere Umgebungen für den jeweiligen Einsatzzweck bereit. Komponentenrahmenwerke bieten eine maßgeschneiderte Infrastruktur für Softwarekomponenten. Durch das Rahmenwerk wird eine Menge von Schnittstellenbeschreibungen bereitgestellt, die von den Komponenten implementiert werden müssen. Diese können dann dynamisch integriert und ausgeführt werden. Das Rahmenwerk stellt Mechanismen bereit, die die gegenseitige Nutzung der Bausteine ermöglicht. Der Vorteil, der durch komponentenbasierte Software erreicht werden soll, ist ein hoher Grad an Wiederverwendbarkeit der Softwarekomponenten, da der Integrationsaufwand minimiert wird.

Laut Definition der Residential Gateway Group (RG Group) sind Residential-Gateways "zentralisierte intelligente Schnittstellen zwischen externen Netzwerken für den häuslichen Zugriff und hausinternen Netzwerken" [Hol97, AMW99]. Mit dem Begriff Residential-Gateway wird die zentrale Hardwareplattform im eHome, inklusive des Betriebssystems, bezeichnet. Es ist die Kontrollinstanz für alle Geräte, die mit diesem verbunden werden. Die Software für deren Ansteuerung befindet sich ebenfalls auf dem Residential-Gateway. Daher muss mindestens eines dieser Geräte in einem eHome vorhanden sein. Die Hardware für das Residential-Gateway lässt sich nach seiner Komplexität in Bezug auf bereitgestellte Dienste und benutzte Netzwerke kategorisieren. Gateways auf Basis von Standard-PCs stellen die komplexeste Form dar, da durch die große Verfügbarkeit von Schnittstellen auf ihnen praktisch alle Dienste und Anschlusstechniken vereint werden können. Weniger komplexe Geräte wie DSL-Router bedienen nur TCP/IP-Netzwerke und stellen Dienste mit geringer Komplexität zur Verfügung.

Das Service-Gateway bildet die Softwareplattform, die die Schnittstelle zwischen eHome und der Außenwelt bildet. Rahmenwerke, die komponentenbasierte Software un-

terstützen, sind für diese Arbeit von besonderem Interesse, da sie als Grundlage für den Aufbau des Gesamtsystems verwendet werden. Sämtliche Software, die die Dienste, die im eHome angeboten werden, implementiert, läuft auf dem Service-Gateway. Dazu gehört auch die Schnittstelle zum Provider und somit zur Außenwelt, über die Dienstleistungen erbracht werden. In dieser Arbeit wird der Begriff Gateway als Kurzform für die Einheit aus Service-Gateway und Residential-Gateway verwendet.

Im eHome an sich befinden sich schließlich die Geräte, die über verschiedenartige Anschlusstechniken mit dem Residential-Gateway verbunden sind und mit Hilfe derer das Service-Gateway die Dienste umsetzt.

Dienstkategorien

eHome-Dienste können grob in folgende Kategorien gegliedert werden:

- Einfache Steuerdienste,
- Infotainment,
- Komfort,
- Kommunikation,
- Sicherheit,
- Telemedizin,
- Verbrauchsüberwachung und -optimierung

Einfache Steuerdienste: Diese Dienste dienen dazu, einzelne Geräte anzusteuern und somit Basisfunktionalität für Erweiterungsdienste zur Verfügung zu stellen. Eine Ausprägung dieser Dienste sind Treiberdienste für alle elektronischen Geräte, die in einem eHome vorhanden sind.

Der Einsatz folgender Geräte in eHome-Diensten ist denkbar:

- **Internetgateways:** Internetgateways verbinden ein Hausnetzwerk oder auch nur ein Residential-Gateway mit dem Internet. Solche Geräte gibt es heutzutage schon recht kompakt mit eingebautem DSL-Anschluss oder einfachen Routing-Fähigkeiten. Oft helfen sie auch, ein strukturiertes Netz für mehrere IP-fähige Computer herzustellen, da sie Dienste wie IP-Masquerading, DHCP oder Firewalling zur Verfügung stellen.
- **Bildschirme:** Allgemein finden sich bereits sehr viele Geräte im Haus, die einen Bildschirm haben und somit als Ausgabegerät genutzt werden können. Dazu zählen Computermonitore, Fernseher, Webpads, PDAs, Mobiltelefone, Wecker, evtl. auch ein Kühlschrank, ein Display im Badezimmerspiegel [PP02], Heizungsthermostate, Waagen oder Uhren

- **Eingabegeräte:** Um mit elektronischen Systemen zu kommunizieren, bieten sich heutzutage PC-Tastaturen, Fernbedienungen, Mäuse, Touchscreens, Lichtschranken, fotooptische Geräte, Schalter, Taster, aber auch Tastaturen oder Mikrofone von Mobiltelefonen an. Im medizinischen Bereich gibt es Blutdruckmesser oder aber auch eine Matratze, die Puls- und Atemfrequenz messen kann (zu besichtigen im inHaus [inH05]).
- **Personenerkennende Geräte:** Eine Anwesenheits- und Personenerkennung ist heutzutage mittels Bewegungssensoren, Kameras oder mittels biometrischen Mitteln wie Fingerabdrucksensoren, Stimmerkennung, Iris-Scan oder Venenmustererkennung [SRSAGM00] möglich. Durch diese lässt sich natürlich auch eine *Personenlokalisierung* durchführen.
- **Geräte zur Beleuchtung und Beleuchtungssteuerung**: Lampen, Dimmer, Rollladen und Markisen
- **Hifi- und Multimediageräte**
  - **Audioausgabegeräte:** Lautsprecher und Alarmsysteme
  - **Audioeingabegeräte:** Mikrophone und Mobiltelefone
  - **Videoeingabegeräte:** Tageslicht- und Infrarot-Kameras
  - **Medienleser und -recorder:** CD/DVD-Spieler und -Brenner, Chip-Karten-Leser/Schreiber, Smartlabels (RFID-Leser/Schreiber) [RFI]
- **Küchengeräte:** Kühlschrank, Ofen, Brotbackmaschine, Toaster und Kaffeemaschine
- **Heizungssystem:** zentraler Brenner, Heizkörper, Klimaanlage (evtl. auch im Auto), Autostandheizung und Sensoren für Temperatur und Feuchtigkeit
- **Größere Haushaltsmaschinen:** Waschmaschinen und Kühltruhe
- **Geräte im Bad:** Badewanne mit Befüllungsanlage, Dusche mit elektronischer Temperaturregelung und verschiedenen Duschdüsen, Whirl-Pool, Durchlauferhitzer, Elektronische Waage, Spiegel mit integriertem Monitor und Sensoren für die Benutzung dieser Geräte
- **Feuermelder**
- **Pollenfilter**

Infotainment-Dienste: Diese Dienste verbessern das heimische Unterhaltungserlebnis. Ein wesentlicher Bestandteil dieses Bereiches ist das Home-Entertainment. Kinoleinwandähnliche Bildschirme oder Projektoren, besondere Sound-Systeme, Streaming-Media und Video-on-Demand sind nur einige wenige Techniken, die diese neuen Dienste ausmachen werden. Es sieht so aus, als wären dies die Dienste, die als nächstes den Markt und die Wohnungen erobern werden. Dies sind die Dienste, die große Firmen wie Microsoft, Sony, Philips oder auch Apple rund um den iPod gerade besonders stark erforschen und mit viel Marketing anpreisen [Ros05, Phi02]. Ein Beispiel ist die Möglichkeit, Musik oder Videos auf jedem beliebigen Bildschirm und jedem Lautsprecher im Haus wiedergeben zu können. Dabei können die Quelldaten von einem beliebigen Eingabegerät, welches im Haus

installiert ist, oder einem mobilen Gerät, welches von einer Person getragen wird, oder auch direkt aus dem Internet stammen.

Ein anderer Dienst ist die Möglichkeit, alle verschiedenen Bildschirme, Player und Recorder im Haus einheitlich mit einer Fernbedienung bedienen zu können, die überall im Haus funktioniert. Mit Hilfe einer Personendetektion (zum Beispiel via RFID-Chips oder Fingerabdrucksensoren) werden Dienste wie `music follows person` oder `video follows person` möglich. Diese Dienste ermöglichen einer Person, einen persönlichen Musik- oder Videostream weiterhören oder -sehen zu können, während sie sich durch das Haus bewegt. Die Musik und/oder der gewählte Videostream folgen der Person durch das Haus an den Ort, an dem sie sich gerade befindet. Natürlich erlaubt eine hausweite Medienabspielmöglichkeit audiovisuelle Benachrichtungen von Nachrichten, Ereignissen und Alarmen entsprechender Personen an ihrem aktuellen Standort.

Komfort: Dienste in dieser Kategorie verbessern den Lebenskomfort, unterstützen das Luxusgefühl und den Lifestyle oder vereinfachen den Alltag. Beispiele sind:

- Ein Portal, um alle Geräte im Haus zu beobachten und sowohl von innerhalb des Hauses über jedes Display als auch von außerhalb des Hauses über das Internet über ein Web-Portal zu steuern.
- Ein Rahmenwerk, welches Geräte termin- und zeitabhängig steuern kann (zum Beispiel Rollladen im Urlaub zu bestimmten Zeiten herunter- und wieder hochfahren). Dieses könnte einen Kalenderdienst beinhalten, der einen Benutzer bei der Planung seiner Termine unterstützt. Dieser Dienst unterstützt verschiedene andere Dienste, welche über diesen Dienst gesteuert werden können.
- Ein Weckdienst, der Stau und Wetterinformationen nutzt, um Hausbewohner pünktlich genug zu wecken, damit sie ihre Arbeitsstelle erreichen. Gleichzeitig werden Warmwassergeräte und Heizung mit Vorlauftemperaturen entsprechend gesteuert. Auch die Helligkeit der Beleuchtung im Haus kann dem aktuellen Wachheitsgrad angepasst werden.
- Ein Weckdienst via Hifi-System, abhängig vom Ort der schlafenden Person.
- Ein schwarzes Brett, um Nachrichten unter den Bewohnern auszutauschen. Das Brett bietet auch die Möglichkeit, Video oder Voice-Nachrichten auszutauschen, sowie die Möglichkeit Mails zu empfangen und zu präsentieren.
- Sprachsteuerung aller Dienste.
- Gestensteuerung aller Dienste.
- Licht-Profile für unterschiedliche Stimmungsssituationen. Beispiele wären hier Fernsehmodus, Arbeitsmodus, Nachtschaltung oder Aufwachmodus.
- Analog zu Musik-folgt-Person: Licht- oder Licht-Profil-folgt-Person.
- Automatische Bewässerung der Pflanzen abhängig von Temperatur, Licht und Luftfeuchtigkeit.

- Essensausgabesystem für Haustiere. Dies könnte eine zeitgesteuerte Anwendung sein, die sich manuell von außen über das Internet steuern lässt. Dieser Dienst ist sowohl für Fische in einem Aquarium als auch für Katzen oder Hunde denkbar.

Kommunikationsdienste: Zu Kommunikationsdiensten gehören E-Mail, SMS, Voice- und Video-over-IP, Instant-Messaging-Dienste. Diese sind bereits auf Heim-PCs weit verbreitet. Sie können als Unterdienste für andere Dienste benutzt werden, um Benachrichtigungsfunktionalitäten anzubieten oder aktuelle Nachrichten über das Wetter, den Verkehr, oder geschäftliche Informationen, die Aktivierungszeiten beeinflussen, zur Verfügung zu stellen.

Sicherheitsdienste: eHome-Dienste können auch das Grundbedürfnis nach Sicherheit abdecken. Alarm-Systeme, welche auf verschiedenste Sensor-Signale reagieren, sind schon heute möglich. Als Sensoren kommen dabei Bewegungssensoren, Kameras, Infrarotkameras, Gewichtsmatten, Bruchsensoren, Temperatursensoren, spezielle Schalter und Schlösser oder Personenerkennungssysteme in Frage. Reaktionen auf Signale solcher Sensoren sind lokale Alarme wie globale Geräuscherzeugung durch eine Sirene, oder aber auch lokale Geräuscherzeugung mittels der installierten Hifisysteme und das Blinken der Beleuchtung. Die lokale Geräuscherzeugung könnte sogar dem Eindringling durch das Haus folgen und so eine noch größere Abschreckungswirkung erzeugen. Auch kann ein Benachrichtigungsdienst Bilder von dem entdeckten Einbruchsversuch zum Wohnungsbesitzer senden oder direkt die Polizei via Internet oder einen mobilen Nachrichtendienst informieren. Die Sicherheitsdienste bieten natürlich auch sowohl die Fernaktivierung und -deaktivierung als die Fernüberwachung. Durch Hinzunahme von Rauch-, Feuer- und Wassersensoren kann auch die Sicherheit der anwesenden Bewohner gesteigert werden sowie das Gebäude selbst geschützt werden. Diese Dienste können lokal anwesende Bewohner mit einem speziellen Alarm alarmieren, wenn Feuer oder Rauch entdeckt wird. In Kombination mit Überwachungsdiensten können wichtige Informationen (Anzahl Leute im Haus, Grundriss des Hauses, Menge des ausgetretenen Wassers, beschädigte Räume, Bilder der Räume vor und während des Unfalls) im Notfall zur Feuerwehr oder zur Polizei übermittelt werden, um ein gezieltes Eingreifen zu ermöglichen.

Telemedizin: Dieser Kategorie werden Dienste zugeordnet, die die Gesundheit der Bewohner betreffen. Sie helfen bei der Überwachung der Körperfunktionen, bei der Kommunikation mit Ärzten und unterstützen ältere oder behinderte Menschen. Im Bereich der Ergotherapie werden ältere oder hilfebedürftige Personen durch elektronische Geräte unterstützt. Die erwähnten Dienste unterstützen die betroffenen Personen im Alltag angefangen bei Erinnerungsfunktionen, welche Schritte im Bad duchzuführen sind, über Lokalisierungshilfen und Vorlesehilfen für Blinde bis hin zur Überwachung der Gesundheit durch automatische Blutzuckeruntersuchung durch mobile Geräte oder der Messung der Zusammensetzung und der Temperatur des Urins beim Toilettenbesuch. Für weitere Informationen siehe [HLM05].

Verbrauchsüberwachung und -optimierung: Jedes elektronisch gesteuerte Gerät kann

bezüglich seines Stromverbrauchs überwacht werden. Heizungssysteme können bezüglich deren Gas-, Öl-, oder Stromverbrauchs überwacht werden, Solarkollektoren bezüglich der aufgefangenen Wärme. Wasseranschlussstellen bezüglich ihres abgegebenen Wassers. Wenn diese Überwachungsfunktionalität mit der Personenerkennung der Personen, die das entsprechende Gerät nutzten, kombiniert wird, kann dies zur Optimierung des Verbrauches verwendet werden. Alleine der pädagogische Effekt, der aus den gesammelten Daten gezogen werden kann, könnte bei der Verbrauchsoptimierung helfen. Auch neue, detailliertere Formen der Abrechnungen werden hierdurch möglich. Ebenso könnten so unterschiedlicher Energiepreise zu unterschiedlichen Tageszeiten ausgenutzt werden (um Lastspitzen auszugleichen). Dienste aus dem Gesundheitssektor könnten einen Diätplan zur Umsetzung eines persönlichen Fitnessplans oder zur Linderung von Allergien unterstützen.

All diese Dienste könnten in einem eHome angeboten werden. Viele davon erfordern die Kombination vieler unterschiedlicher Geräte in unterschiedlichen Räumen. Es ist schnell ersichtlich, dass die physikalische Beschaffenheit der Umgebung eine großen Einfluss auch auf die Dienstprogrammierung und -konfigurierung hat. Auch ist bei den unterschiedlichen Geräten, die angesteuert werden, aber teilweise gleiche Funktionalität anbieten, klar, dass es sinnvoll ist, diese in unterschiedlichen Kontexten einsetzen zu können. Um den Entwicklungsprozess für Erweiterungsdienste, die diese Gerätefunktionalität nutzen wollen, zu vereinfachen, wird auch hier eine sinnvolle Abstraktion nötig, welche im weiteren Verlauf der Arbeit erreicht werden soll.

2.1.3 eHome-Teilnehmer/Rollen

Ein Bewohner eines eHomes wird, abhängig von seinen installierten Diensten, mit verschiedenen Personen oder Institutionen interagieren:

Bewohner/Benutzer/Kunde: Eine besonders wichtiger Teilnehmer in einem eHome-System ist natürlich der *Bewohner* des eHomes selbst, der direkt mit diesem interagiert und von dessen Diensten profitiert. Er wird in dieser Arbeit auch *Benutzer* oder *Kunde* genannt.

Dienstanbieter: Der *Dienstanbieter* stellt die vorgestellten Dienste und benötigten Geräte für den Kunden zusammen und bietet sie diesem an. Er vermittelt Dienstleistungen von *Unterdienstanbietern*. Er wartet die Dienste und passt sie gegebenenfalls an neue Hardware oder auch neue Funktionalitäten an. Dies ist die Firma, bei der der Kunde eHome-Geräte bestellt oder die der Kunde mit der Einrichtung neuer eHome-Dienste beauftragt. Die Begriffe *eHome-Provider*, *eHome-Anbieter*, *Anbieter* und *Service-Provider* werden synonym zu Dienstanbieter verwendet.

Ernährungsberater: Der *Ernährungsberater* kann Dienste aus dem Bereich der Gesundheit und der Ernährung anbieten, welche mit Diensten der Gewichtsüberwachung oder einem Display auf dem Kühlschrank, welches direkte Ernährungsberatung anbietet, verbunden werden können.

Frisör, Kosmetiker, Modeberater: Diese Unterdienstanbieter bieten Dienste an, die sich auf das Aussehen des Kunden beziehen. Durch Einsatz einer Kamera, eines

Displays im Ankleideraum oder über ein in einen Spiegel integriertes Display im Badezimmer könnte hier eine interaktive Beratung stattfinden, aber auch archivierte Beratung oder Werbeangebote aufgerufen werden.

Datensicherheitsbeauftragter: Ein *Datensicherheitsbeauftragter* ist dafür verantwortlich, dass Kundendaten nur Berechtigten gelesen und verändert werden können.

Feuerwehr: Im Notfalls können wichtige Daten, wie die Anzahl der Verletzten oder Rauminformationen oder Bilder vom Unfall an die *Feuerwehr* über einen Dienst, den die Feuerwehr anbietet, übermittelt werden und somit das Eingreifen vereinfachen.

Polizei: Analog können auch Daten an die *Polizei* bei einem Einbruch oder Eindringen übertragen werden, um auf ein bevorstehendes Eingreifen besser vorbereitet zu sein.

Bedauerlicherweise werden auch Katastrophenszenarien oder Amokläufe immer mehr zur Realität. Die angesprochenen Dienste könnten hier Polizei und Feuerwehr mit Umgebungsinformationen und entsprechenden anderen Messungen unterstützen und so gegebenenfalls die Situation entschärfen.

Wettervorhersage: Diese Institution kann über einen Dienst aktuelle und zukünftige Wetterinformationen übermitteln, um zum Beispiel im Energiebereich Planungen zuzulassen oder bei Straßenglätte eine frühere Weckzeit einzustellen.

Verkehrsinformationsbüro: Über einen Dienst des *Verkehrsinformationsbüro* können Routen zur Arbeit besser geplant werden und bestehende oder zu erwartende Staus für die Berechnung der Weckzeit beim Weckdienst einbezogen werden.

Komponentenentwickler: Die Komponentenentwickler schreiben die Softwarekomponenten, die bestimmte Funktionalitäten oder respektive daraus zusammengesetzte Dienste realisieren. Sie schreiben sowohl die geräteorientierte Software, also die Gerätetreiber, als auch die höhere Applikationslogik für Erweiterungsdienste.

Zu einem eHome-System gehören neben dem eigentlichen Heim, seinen Geräten und den Benutzern auch externe Anbieter von Dienstleistungen. Beispiele solcher Anbieter sind oben genannt worden. Auch hier ergibt sich die Notwendigkeit, dass eine Systembeschreibung oder Konfiguration Daten für all diese Teilnehmer bereitstellen und speichern muss, um eine sinnvolle Interaktion und Diensterfüllung möglich zu machen. An dieser Stelle stellen sich natürlich auch Fragen bezüglich der Sicherheit solcher Daten, die aber nicht in dieser Arbeit behandelt werden.

2.2 Feinszenario

Da nicht alle in Abschnitt 2.1.2 (Seite 19) vorgestellten Geräte für diese Arbeit zur Verfügung standen und auch die zeitlichen Ressourcen für die Softwareentwicklung der die Geräte nutzenden Dienste begrenzt waren, werden hier nur die Dienste vorgestellt, die auch in den betrachteten Simulationsumgebungen realisiert wurden.

2.2.1 Untersuchte Beispieldienste

Es wurden folgende Beispieldienste implementiert und untersucht:

1. **Beleuchtungsdienst (Lighting-Service):** Dieser Dienst bietet die einfache Funktionalität der Beleuchtungssteuerung an. Das heißt, er ermöglicht das Ein-, Ausschalten und Dimmen des Lichts in einem Raum oder einem Bereich eines Raumes mittels Schaltern oder Tastern. Dieser Dienst stellt also keine wirkliche Neuerung gegenüber einer festen physikalischen Verdrahtung dar, wenn er allein genutzt wird. Jedoch bietet er mehr Flexibilität dahingehend, dass Beleuchtungsbereiche neu definiert werden können, ohne dass eine neue physikalische Verkabelung stattfinden muss und seine Unterdienste bieten die Möglichkeit, auch von andern Diensten gesteuert zu werden, um die Beleuchtung auch mit anderen Diensten zu kombinieren.

2. **Licht-Bewegungs-Dienst (Light-Motion-Service):** Der Licht-Bewegungs-Dienst erlaubt die Beleuchtungssteuerung mittels Bewegungssensoren bzw. mittels Geräten, die Bewegung erkennen können, welche auch normale oder Infrarotkameras sein können. Wird in einem Raum oder Bereich ein sich bewegendes Objekt einer konfigurierbaren Größe erkannt, so wird für eine konfigurierbare Zeit die Beleuchtung eingeschaltet bzw. auf ein konfigurierbares Niveau gedimmt. Auch dieser Dienst spielt seine Stärken gegenüber der klassischen physikalischen Methode durch Flexibilität und Kombinationsfähigkeit aus.

3. **Sicherheitsdienst (Security-Service):** In erster Instanz wurde ein *vereinfachter Sicherheitsdienst* betrachtet, siehe Abbildung 2.2. Dieser kombiniert Kameras, Bewegungsmelder, eine Sirene, einen Aktivierungsschalter und ein Residential Gateway mit Internetzugang. Der Aktivierungszustand, wird auf dem Residential Gateway bzw. auf einem normalen PC gespeichert. Ist dieser Zustand aktiv und wird über einen Bewegungsmelder eine Bewegung festgestellt, so wird über die Sirene ein Alarm ausgelöst und das aktuelle Bild der Kamera per E-Mail an eine bestimmte Adresse verschickt. Von welcher Kamera dieses Bild stammt, ist davon abhängig, welcher Bewegungsmelder ausgelöst wurde, d.h. es besteht eine Beziehung zwischen Bewegungsmeldern und Kameras. Diese Beziehung ist eine n:m-Beziehung. Zwar wird nur ein Bild in einer E-Mail gesendet, die Kamera, von der dieses stammt, wird aber aus mehreren ausgewählt. Dabei wird die erste Kamera ausgewählt, die aktiv ist. Ist dem Bewegungsmelder keine aktive Kamera zugeordnet, wird eine reine Textnachricht verschickt. Unabhängig davon wird die Sirene ausgelöst.

   Im allgemeinen Sicherheitsdienst, oder später auch nur als der Sicherheitsdienst bezeichnet, kommen einige Abstraktionen hinzu. Ist der Sicherheitsdienst aktiviert, bietet er die Überwachung des eHomes über eine Eindringlingserkennung an. Diese kann die gerade besprochene Bewegungserkennung sein aber auch durch Fußmattenkontakte oder Glasbruchsensoren ausgelöst werden. Wird auf diese Weise ein Eindringling erkannt, wird ein lokaler Alarm ausgelöst und die Polizei und/oder der Bewohner verständigt. Der lokale Alarm kann dabei eine Sirene sein, ein Geräusch, was im entsprechenden Bereich oder global abgespielt wird und/oder die

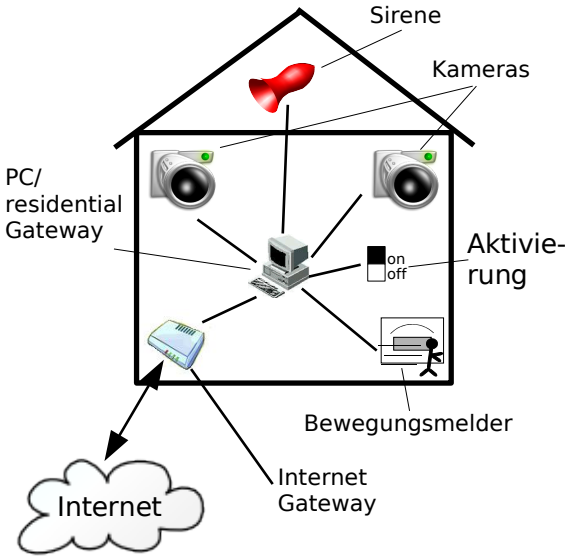

Abbildung 2.2: Einfacher Sicherheitsdienst

Beleuchtung, die ein visuelles Signal wie zum Beispiel Blinken gibt. Der Benachrichtigungsempfänger kann konfiguriert werden und die Nachricht per Internet als E-Mail oder über Mobilfunk als SMS oder MMS versendet werden. Da in diesem Dienst Kameras zum Einsatz kommen können, können ein oder mehrere Bilder vom betroffenen Bereich übermittelt werden. Bei diesem Dienst sieht man bereits Möglichkeiten, die physikalisch verdrahtete oder am Markt vorhandene Alarmsysteme nicht bieten können, da hier die Durchdringung unterschiedlicher Dienstdomänen (Unterhaltung, Komfort und Sicherheit) klar wird, die normalerweise nicht einfach kombinierbar sind.[1]

4. **Musik-folgt-Person-Dienst (Music-Follows-Person-Service):** Durch die Kombination von Multimediafunktionalitäten der Steuerungsdienste von Hifi- und anderer Homeentertainment-Geräten auf der einen Seite und Personenerkennung auf der anderen Seite, kann ein Musik-folgt-Person-Dienst realisiert werden. Dieser lässt einen einer Person zugeordneten Musikkanal durch dessen Heim folgen. Wenn eine Person ihre Musik zum Beispiel in der Küche einschaltet und dann ins Wohnzimmer geht, so schaltet sich die Musik in der Küche aus und spielt im Wohnzimmer an der entsprechenden Stelle weiter. Natürlich muss hier konfiguriert werden, wessen Musik Priorität besitzt, wenn eine zweite Person den Raum betritt. In dieser Arbeit wurde die Strategie gewählt, dass der zuletzt den Raum betretende die gespielte

[1] Eine mögliche Erweiterung dieses Dienstes wäre auch, durch Herunterlassen der Rollläden und Verschließen der Zimmertüren den Eindringling einzuschließen, bis die Polizei eintrifft. Dies wurde allerdings nicht in dieser Arbeit umgesetzt.

Musik angibt. Es kann zum Beispiel aber auch die Musik der Eltern höhere Priorität haben als die der Kinder. Es sei hier angemerkt, dass mangels einer wirklich leicht zu realisierenden Personenerkennung über installierte Kameras, die Personenerkennung natürlich auch via RFID-Chips, ein persönliches Mobiltelefon oder einfach auch nur über bestimmten Personen zugeordneten Schaltern realisiert werden kann.

5. **Alles-Ein-Dienst (All-On-Service):** Dieser Dienst ermöglicht das Anschalten sämtlicher Beleuchtung in der ganzen Wohnung über einen Schalter. Dieser Dienst kann zum Beispiel als eine Art Panikfunktion in der Nacht eingesetzt werden, um zum Beispiel bei einem Geräusch zu kontrollieren, ob etwas vorgefallen ist.

6. **Alles-Aus-Dienst (All-Off-Service):** Analog gibt es auch als entgegengesetzten Dienst den Alles-Aus-Dienst, über den sich zum Beispiel beim Verlassen des Gebäudes alles Licht und gegebenenfalls auch alle Geräte mit einem Knopfdruck, oder beim Verschließen der Haustür von außen ausschalten lassen.

2.2.2 Simulationsumgebungen

Da es ja ein Ziel dieser Arbeit ist, den Entwicklungsprozess für eHome-Systeme so zu verbessern, dass er für verschiedene Umgebungen funktioniert, wurden die gerade besprochenen Dienste und deren Konfigurierung im Rahmen der entwickelten eHomeConfigurator-Werkzeugsuite in drei unterschiedlichen Umgebungen evaluiert. Zwei dieser Umgebungen wurden für diese Arbeit als Demonstratoren gebaut. Die dritte ist ein reales Haus eines Kooperationspartners.

Die Diversität der Umgebungen und benutzten Techniken bildet ein realistisches Szenario, um die Einrichtung von eHome-Systemen und die damit verbundene Entwicklungs- und Konfigurierungsarbeit auszuwerten.

Einfacher X10-Demonstrator

Mit ein paar vorhanden X10-Geräten wurde zu Anfang der Arbeit der vereinfachte Sicherheitsdienst evaluiert.

Als Geräte werden in diesem Dienst X10-Bewegungsmelder, ein X10-Schalter mit angeschlossener Sirene und Webcams eingesetzt. Die Sirene ist in diesem Fall ein Gerät, welches – wie eine Lampe – ein- oder ausgeschaltet wird. Zu Testzwecken wurde hier einen Lautsprecherbox mit aktivem Verstärker genutzt, an der ein dauerhaftes Tonsignal an lag. Das heißt, das diese Lautsprecherbox ein- und ausgeschaltet wird. Ist sie eingeschaltet, ertönt das Signal. Der Aufbau des Szenarios, in dem dieser Dienst verwendet wird, ist in Abbildung 2.2 dargestellt. Diese Abbildung zeigt schematisch die im Haus eines Kunden vorhandenen Geräte und deren physikalischen Verbindungen.

Die Steuerungssoftware für die Geräte läuft auf einer Plattform auf dem Residential Gateway. Als Betriebssystem der Plattform wird Debian Linux [Deb] verwendet. An den PC, der das Residential Gateway darstellt, sind die einzelnen Geräte angeschlossen. Die Bewegungsmelder und der Schalter der Sirene werden in der Realität, anders als Abbildung 2.2 vermuten lässt, nicht direkt mit dem Rechner verbunden. Stattdessen ist mit dem Rechner über die serielle Schnittstelle ein sogenanntes X10-Gateway verbunden, welches

Abbildung 2.3: Der X10-eHome-Demonstrator (Aufsicht)

zwischen X10-Signalen auf dem Stromnetz des Hauses und seriellen Signalen vermittelt. An dem Stromnetz des Hauses hängen dann ein X10-Funkempfänger und ein X10-Schalter. Die Bewegungsdetektoren senden nun über Funk Signale an den Funkempfänger, der diese über das Stromnetz dem X10-Gateway und somit der Steuerungssoftware auf dem Residential-Gateway weiterreicht. Auf Betriebssystemebene wurde Wish [Hil02] als Treiber für die X10-Geräte verwendet. Die Kameras werden an USB-Schnittstellen des PCs angeschlossen. Um sie anzusprechen, wird FFMPEG [FFM] verwendet.

X10-eHome-Demonstrator

Der erste für diese Arbeit geschaffene Demonstrator besteht aus Holz und ist mit am Markt verfügbaren Geräten ausgestattet. Er hat ein Volumen von ca. einem Kubikmeter. Es wurden Geräte verwendet, die bereits jetzt für geringe Kosten am Markt zu erstehen sind und bei geeigneter Programmierung ein eHome-System ermöglichen. Als Geräte sind X10-Lampenfassungen mit Glühbirnen, X10-Bewegungssensoren[Hau], Schaltertafeln, Lautsprecheranschlüssen, Türsensoren, Touchscreens und Webcams installiert. Die Aufsicht ist in Abbildung 2.3 zu sehen. Unterhalb des Bodens befinden sich drei Rechner, an denen die gerade angegebenen Geräte angeschlossen sind. Der Demonstrator wurde das erste Mal öffentlich in Tartu, Estland auf einem Workshop vorgestellt [Ulr05b]. Er stellt eine eingeschossige Wohnung mit acht Zimmern dar. Jedes Zimmer hat eine X10-Lampenfassung mit einer Glühbirne. In sechs Zimmern können Lautsprecher angeschlossen werden, sind Bewegungsmelder installiert und Tastenfelder mit je zehn Tasten angebracht. In drei Zimmern gibt es Touchscreens und Kameras. Alle Türen besitzen Türsensoren, die anzeigen, ob eine Tür offen oder geschlossen ist. Im Wohnzimmer gibt es ein Fenster, das sich öffnen und schließen lässt, welches auch von einem Sensor überwacht wird. Die Personenerkennung wird hier über die Tastenfelder emuliert. Jeder Person ist eine Taste zugeordnet (siehe Abbildung 2.4). Die anderen Tasten werden für das Schalten des Lichts im entsprechenden Raum, das Ein- und Ausschalten der Musik für eine Person

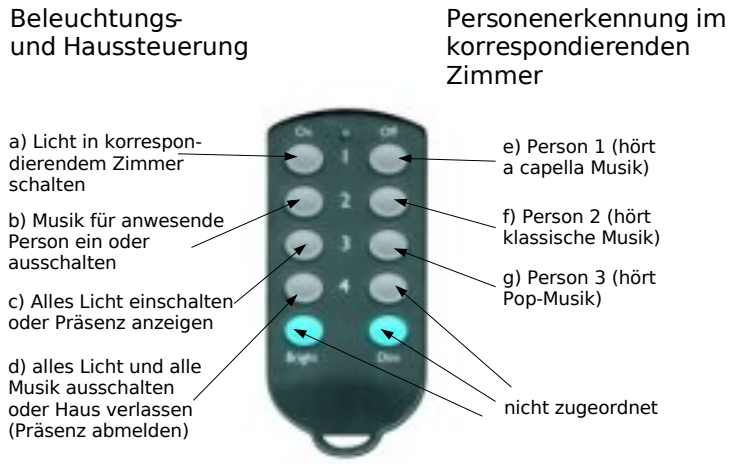

Abbildung 2.4: Zuordnungen der Tasten auf dem Tastenfeld

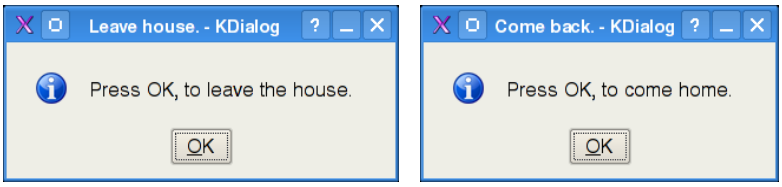

Abbildung 2.5: Wechselnde Auswahldialoge auf Touchscreen für Präsenzschaltung

und das Aktivieren der Dienste Alles-Ein und Alles-Aus benötigt. Die Benutzung eines RFID-Scanners zur Personenerkennung war in dem Kontext dieser Arbeit nicht möglich, da zu diesem Zeitpunkt hierfür keine probaten Mittel zur Verfügung standen. Eine Gesichtserkennung über die Kamera ist zur Zeit ohne einen enormen Programmier- und Trainingsaufwand nicht möglich.

Im Einzelnen werden hier die gerade vorgestellten Dienste wie folgt realisiert:

1. **Beleuchtungsdienst:** Beim Druck auf die linke obere Taste (a) auf dem Tastenfeld wird die Glühbirne des Zimmers, in dem sich das Tastenfeld befindet umgeschaltet. Ist das Licht aus, so wird das Licht eingeschaltet und umgekehrt.

2. **Sicherheitsdienst:** Durch Druck auf die dritte oder vierte linke Taste von oben des Tastenfeldes (c oder d) in der Diele kann die Präsenz ein- oder ausgeschaltet werden. Es kann also angegeben werden, ob jemand im Haus ist oder nicht und somit respektive der Sicherheitsdienst deaktiviert oder aktiviert werden. Diese Präsenz kann auch über ein Computerprogramm, welches die Dialoge, die in Abbildung 2.5 dargestellt sind, darstellt, verändert werden. Der Dialog wird auf dem Touchscreen links unten (siehe Abbildung 2.3) dargestellt und kann über dessen OK-Knopf geschaltet werden.

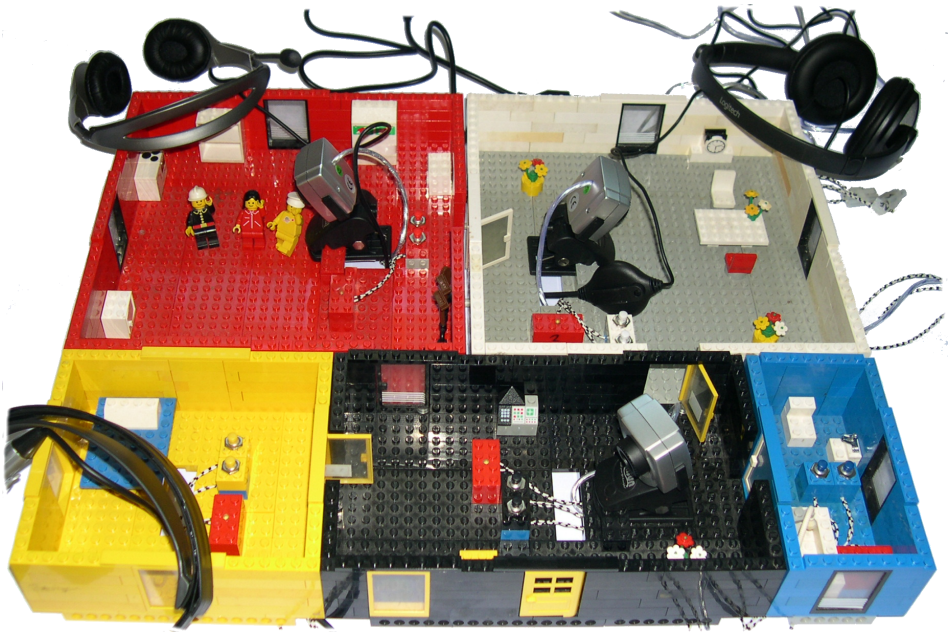

Abbildung 2.6: Der Lego-eHome-Demonstrator

3. **Musik-folgt-Person-Dienst:** Wie bereits angesprochen, muss in der X10-Demonstrator-Realisierung die Personenerkennung über die rechten Taster simuliert werden (vergleiche Abbildung 2.4, e-g). Mittels des zweiten linken Tasters kann die Musik für die gerade anwesende Person ein- bzw. durch nochmaliges Drücken wieder ausgeschaltet werden. Person 1 hört in diesem Fall a-capella-Musik, Person 2 hört klassische Musik und Person 3 Popmusik. Die Musik wird über die Klinkenstecker, die mit den Soundkarten der Rechner verbunden sind, im entsprechenden Raum ausgegeben.

4. **Alles-Ein-Dienst:** Durch Druck auf die dritte linken Taste des Tastenfeldes im Schlafzimmer (c) wird in allen Räumen das Licht eingeschaltet.

5. **Alles-Aus-Dienst:** Durch Druck auf die vierte linke Taste des Tastenfeldes im Schlafzimmer (d) wird in allen Räumen Licht und Musik ausgeschaltet.

Dieses Szenario wird auch in leichten Variationen getestet. Das heißt, die Dienste werden in weniger Räumen oder nicht alle gleichzeitig eingesetzt.

Lego-eHome-Demonstrator

Der zweite Demonstrator wurde aus Lego-Bausteinen angefertigt (siehe Abbildung 2.6). Seine Ausmaße sind erheblich geringer als die des X10-Demonstrator, so dass dieser auch im Reisegepäck verstaut werden kann. Er wurde das erste Mal in Tokio, Japan vorgestellt [Ulr05a]. Er ist mit selbst gebauten Legolampen, Legotastern, USB-Webcams und

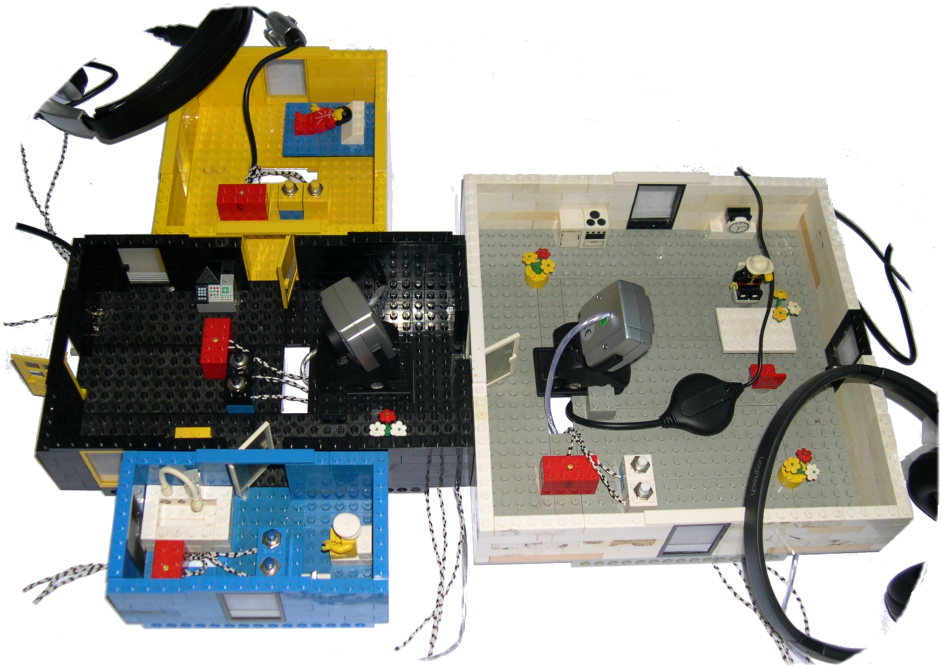

Abbildung 2.7: Der modifizierte Lego-eHome-Demonstrator

USB-Audio-Systemen ausgestattet und kann von einem externen Notebook komplett über USB angesteuert werden. Er besteht aus fünf unterschiedlich farbigen Räumen, die sich auf unterschiedliche Weise zusammen setzen lassen, um noch mehr Varianten abbilden zu können. Einen alternativen Aufbau zeigt Abbildung 2.7. Zum Demonstrator gehören drei USB-Kameras, die sich in unterschiedlichen Räumen installieren lassen. Im Grundaufbau befindet sich jeweils eine im Schlafzimmer, in der Küche und im Wohnzimmer. Weiterhin gehören drei USB-Audio-Headsets dazu, welche den entsprechenden Räumen, in denen auch jeweils die Kameras untergebracht sind, zugeordnet werden. Zehn Taster können über die Räume verteilt werden. Von diesen sind jeweils zwei gleich gefärbt, entsprechend den Farben der Räumen. Allerdings hat immer einer dieser zwei Taster noch eine weitere Farbe (vergleiche Abbildung 2.8). Im Grundaufbau, wie in Abbildung 2.6, sind die Taster jeweils in den gleichfarbigen Räumen angebracht, also je zwei pro Raum. In jedem Raum gibt es eine in einen Legostein eingebaute gelbe Leuchtdiode, die die Beleuchtung des entsprechenden Raumes symbolisiert (siehe Abbildung 2.9). Alle Taster und Leuchtdioden sind mittels einer parallelen Schnittstelle über USB an dem steuernden Laptop angeschlossen. Weiterhin gehören zu diesem Demonstrator drei unterschiedlich farbige Legofiguren, die die Bewohner des Hauses darstellen.

Die besprochenen Dienste werden auf diesem Demonstrator wie folgt umgesetzt:

1. **Beleuchtungsdienst:** Beim Druck auf einen gleichfarbigen Taster wird das Licht im korrespondierenden Raum geschaltet.

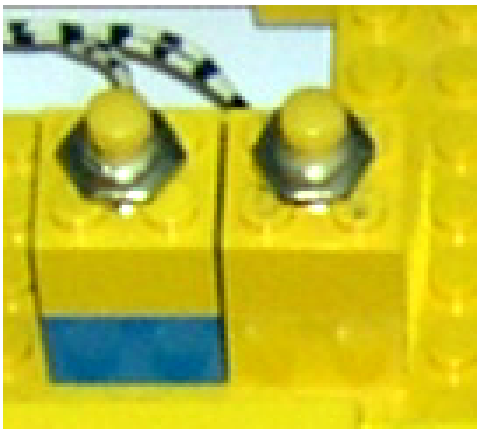

Abbildung 2.8: Zwei Lego-Taster in einem Raum, einer mit zusätzlichem Farbstein

Abbildung 2.9: Selbst gebaute Lego-Lampe: in Legostein eingebaute Leuchtdiode

2. **Sicherheitsdienst:** Über die Dialoge aus Abbildung 2.5, die auf dem angeschlossenen Laptop dargestellt werden, wird die Präsenz geschaltet, also der Dienst aktiviert bzw. deaktiviert. Die Kameras dienen hier im Aktivierungsfall gleichzeitig sowohl als Bewegungssensoren, als auch als Bildaufzeichnungsgeräte, wenn ein Eindringling detektiert wurde. Im Fall einer Detektion wird in dem Raum, wo die Bewegung festgestellt wurde, ein Alarmsound über das entsprechende Headset abgespielt und die Lampe wird zum Blinken gebracht. Das Bild, welches die Kamera gerade aufnimmt, wird wie beim X10-Demonstrator per E-Mail verschickt.

3. **Musik-folgt-Person-Dienst:** Die erforderliche Personenerkennung für diesen Dienst wird über die Kameras realisiert. Die Legofiguren werden an eine bestimmte Position vor die Kameras gestellt und können so von einer Bilderkennungssoftware erkannt werden. Dies ist sicherlich nicht mit einer professionellen Personenerkennung, wie sie heute möglich wäre, zu vergleichen. Es zeigt aber im Vergleich zum X10-Demonstrator, der für die Erkennung Taster benutzt, dass die Funktionalität Personenerkennung auf ganz unterschiedliche Weise umgesetzt werden kann, sich aber trotzdem in den Gesamtdienst integriert. Das An- und Ausschalten der Musik einer bestimmten Person (bzw. der entsprechenden Legofigur) in einem bestimmten Raum erfolgt über den mehrfarbigen Taster in dem entsprechenden Raum, in dem sich die Person befindet. Wird die Person nach Anschalten der Musik aus dem Raum entfernt, so verstummt die Musik dort. Wird sie dann in einen anderen Raum an die entsprechende Position gestellt, läuft die Musik dort weiter.

4. **Alles-Ein-Dienst:** Mit Druck auf den mehrfarbigen Taster im Badezimmer wird in allen Räumen das Licht eingeschaltet.

5. **Alles-Aus-Dienst:** Die Bedienung des mehrfarbigen Tasters in der Diele schaltet in allen Räumen Licht und Musik aus.

Um bei einer Konfigurierung alle Adressdaten der installierten Geräte zur Verfügung zu haben, wurde die Tabelle 2.1 erstellt. Die Lampen und die Knöpfe sind über einen Cleware-IO16-Controllers, einer Art Parallelen-Schnittstelle für USB, mit dem Steuerrechner verbunden. Dabei entsprechen die Adressparameter der Lampe (lamp) und der Knöpfe (unicolor and dualcolor) in der Tabelle den Adressleitungen des Cleware-IO16-Controllers. Die Kameranummer (camera) gibt an, welche von den gefundenen Kameras des Steuerrechners verwendet werden soll. Dabei werden die Nummern in der Reihenfolge der Initialisierung vergeben. Der Adressparameter in der Spalte sound gibt an, welche Soundkarte des steuernden Computer verwendet werden soll.

inHaus Duisburg

Die dritte Testumgebung ist ein reales Haus eines Kooperationspartners in Duisburg, das inHaus [inH05]. Dieses Haus besteht aus zwei Hälften: dem Wohnhaus und dem Werkstatthaus. Das Wohnhaus ist ein tatsächlich bewohnbares Haus, in dem sehr viele elektronisch steuerbaren Geräte installiert sind. Es ist somit eine ideale Testplattform für die Installation von eHome-Systemen. Zum Wohnhaus gehört das Wohnlabor bestehend aus Diele, Küche, Wohnzimmer, Schlafzimmer, Kinderzimmer und Bad. Weiterhin gehören

room	camera	sound	lamp	unicolor	dualcolor
hall	-	-	4	10	11
bedroom	1	4	5	6	7
bathroom	-	-	3	16	15
kitchen	3	3	1	8	9
livingroom	2	2	2	14	12

Tabelle 2.1: Adressparameter der im Lego-eHome-Demonstrator installierten Geräte

dazu ein Home-Office, ein Multimedia-Auto und ein smarter Garten. Das Werkstatthaus ist an ein Forschungslabor angelehnt und bietet Arbeitsräume und die Technik-Zentrale.

Im Wohnraum befinden sich steuerbare Küchengeräte, Sanitärinstallationen, Lüftungsanlage, Verschlusstechnik, Schaltelemente und viele Geräten der Unterhaltungselektronik bis hin zum Home-Theater. Innerhalb befinden sich ein European-Installation-Bus, der Honeywell-Funk-Bus und RFID-Sensoren. Für weitere Informationen zum inHaus siehe Abschnitt 3.2.2 auf Seite 42.

2.3 Anwendungsszenarien

Es gibt drei primäre Anwendungsfelder, wie die Einrichtung von eHome-Systemen unterstützt werden kann: Bauszenario, Ausrüstszenario und Steuerungsszenario

2.3.1 Bauszenario

Zunächst kann die Planung eines noch zu bauenden eHome-Systems unterstützt werden. In diesem Fall kann die Anzahl der verschiedenen Geräte bestimmt werden, die erworben werden müssen, um bestimmte Dienste zu erfüllen. Darüber kann mittels einer Umgebungsdarstellung deutlich gemacht werden, welche davon in welchem Raum installiert werden müssen. Im Folgenden wird dieser Fall als *Bauszenario* bezeichnet.

2.3.2 Ausrüstszenario

Ebenso ist es denkbar, dass ein Haus- oder Wohnungseigentümer seine Räumlichkeiten mit eHome-Diensten ausrüsten möchte. Er kann eventuell bereits Geräte besitzen, die er benutzen möchte. Er will also nur wissen, welche Geräte er zusätzlich für einen gewählten Dienst benötigt. Auf dieses Szenario wird unter dem Namen *Ausrüstszenario* Bezug genommen.

2.3.3 Steuerungsszenario

Schließlich gibt es den Fall, dass nur ein Gerät mit einer Steuerungssoftware und einer Vernetzung gewünscht werden. Das heißt, es wird insbesondere nach der richtigen Software zur Steuerung vorhandener elektronischer Geräte und deren Integration zu eHome-Diensten gesucht. In diesem *Steuerungsszenario* besteht die Aufgabe eines unterstützenden Werkzeugs darin, Geräte- oder Dienstattribute ändern zu können,

bereits installierte Dienste zu stoppen, deren Raumauswahl zu ändern oder neue Dienste hinzuzufügen.

Für jedes der drei Szenarien ergeben sich andere Ansprüche des Benutzers an ein unterstützendes Werkzeug. Im Bauszenario und im Ausrüstszenario steht die Liste mit den noch zu erwerbenden Geräten im Vordergrund des Interesses. Es ist wahrscheinlich, dass der Anwender sich anhand der gebotenen Funktionalitäten für eine Menge von Diensten entscheidet und diese dann ohne Einschränkung in vielen Räumen laufen lassen möchte. Im Gegensatz dazu ist es denkbar, dass der Benutzer im Steuerungsszenario sich zwar für einen neuen Dienst interessiert, aber kein Interesse am Erwerb von neuen Geräten für sein eHome hat, da er eine zusätzliche Investition scheut. In diesem Fall ist es für ihn interessant, in welchen Räumen er ohne zusätzliche Neuinstallationen diesen Dienst ausführen kann, notfalls mit eingeschränkter Funktionalität.

2.4 Zusammenfassung

In diesem Kapitel wurden der grobe Kontext und die der Arbeit zugrunde liegende Sichtweise vorgestellt. Mögliche Dienste wurden vorgestellt und die tatsächlich realisierten Dienste besprochen. Eine wichtige Erkenntnis ist, dass die physikalische Beschaffenheit der Umgebung einen großen Einfluss auf die Dienstprogrammierung und -konfigurierung hat. Auch ist bei den unterschiedlichen Geräten, die angesteuert werden, die aber teilweise gleiche Funktionalität anbieten, klar, dass es sinnvoll ist, diese in unterschiedlichen Kontexten einsetzen zu können. Um den Entwicklungsprozess für Erweiterungsdienste, die diese Gerätefunktionalität nutzen wollen, zu vereinfachen, wird auch hier eine sinnvolle Abstraktion nötig.

Es ist notwendig ein Modell für eine Konfiguration anzugeben, das Daten für alle beschriebenen Dienste und deren realisierenden Softwarekomponenten, ihre Abhängigkeiten, alle Geräte und die physikalischen Ortszusammenhänge speichern kann.

Kapitel 3

Stand der Technik

Dieses Kapitel gibt einen Überblick über die am Markt verfügbaren eHome-Dienste, solche die sich zur Zeit in der Entwicklungs- oder Testphase befinden und zeigt, wie deren Entwicklungsprozess gehandhabt wird.

3.1 eHome-Dienste

Einige der in Abschnitt 2.1.2 angesprochenen Dienste gibt es bereits am Markt zu kaufen.

3.1.1 Integrierter Hifiwecker

Einer der einfachsten dieser integrierten Dienste ist wohl der integrierte Hifiwecker. Diese Geräte gab es zum Zeitpunkt des Entstehens dieser Arbeit überall für sehr erschwingliche Beträge zu kaufen. Es handelt sich hierbei meist um ein kleines tragbares Radio oder einen tragbaren CD-Player mit integrierten Lautsprechern und eingebautem Wecker. Diesen kann man auf eine bestimmte Uhrzeit einstellen, zu welcher der CD-Spieler bzw. das Radio eingeschaltet werden soll. Hier werden ein Zeitsteuerungs-, bzw. Komfortdienst und ein Musikwiedergabedienst zu einem neuen Dienst integriert. Es handelt sich also um zwei Funktionalitäten.[1] An diesem Gerät wird deutlich, wie durch die Kombination zweier einfacher Einzeldienste ein pfiffiger (im Sinne von *smart*) Mehrwert geschaffen werden kann, der dem Benutzer das Leben etwas angenehmer gestaltet.

Solche Hifiwecker sind allerdings in sich abgeschlossene Systeme. Sie lassen sich nicht mit anderen Diensten in neue Dienste integrieren. Auch das Austauschen einer einzelnen Komponente bei Defekt ist in der Regel sehr schwierig.

3.1.2 Alarmsysteme

Unter Alarmsystemen werden in dieser Arbeit komplexe Alarmanlagen verstanden, die unterschiedliche Sensoren nutzen können und verschieden auf das Erkennen von gemes-

[1] Wie später zu sehen ist, kann der Lautsprecher, also die Musikwiedergabe, auch als eigener Dienst betrachtet werden, so dass CD-Player in die Dienste Medienlesedienst und Musikwiedergabedienst zerfallen würde.

senen Ereignissen der Sensoren reagieren können. Viele solcher Alarmsystem bieten heute schon die Integration von Bewegungssensoren, Glasbruchdetektoren und Fußmattensensoren des gleichen Herstellers an. Auch erlauben sie den Fernzugriff via Mobiltelefon oder gar über das Internet und bei installierten Kameras eine Online-Überwachung. Bei manchen Alarmanlagen lässt sich auch die Rollladensteuerung mit integrieren, um eine Anwesenheit im Urlaub zu simulieren.

Aber auch diese Systeme sind geschlossen und bieten keine Erweiterungsmöglichkeiten über den vordefinierten Standard hinaus. So ist zum Beispiel keine Einbindung der lokalen Beleuchtung oder der installierten Hifianlage zur Wiedergabe eines lokalen Alarms möglich. Auch kann bei diesen Systemen nur ganz bestimmte Hardware eingesetzt werden, nicht einfach jeder computerauslesbare Sensor. So ist die Anschaffung der Einzelkomponenten solcher Systeme mit einem hohen Kostenaufwand verbunden und die clevere Kombination mit Geräten aus anderen Bereichen nicht möglich.

3.1.3 Heizungssysteme

Auch Heizungssysteme bestehen aus unterschiedlichen Sensoren und Aktoren. Sensoren sind dabei Temperaturfühler, Thermostate und Druckventile. Aktoren sind Heizkessel, Mischer, Umwälzpumpe, Thermostate und wieder Druckventile[2]. An den Thermostaten lassen sich bestimmte Solltemperaturen einzelner Zimmer einstellen. Über Außenfühler wird versucht, die komplexen Regelkreisläufe an entsprechende Außentemperaturen anzupassen, um den Verbrauch zu minimieren.

Auch hier handelt es sich wieder um ein geschlossenes System, welches sich nicht mit externen Systemen koppeln lässt. Wünschenswert wäre hier eine Kopplungsmöglichkeit mit Rollladensystemen, mit einer Klimaanlage, aber auch mit Sensoren zur Speicherung von Messdaten, die Auskunft über die Anwesenheit der Bewohner geben oder eine genaue Auswertung des Energieverbrauchs zulassen würden.

Auch eine Information, ob sich gerade alle Bewohner zum Schlafen im Schlafzimmer aufhalten und so eine Nachtabsenkung durchgeführt werden könnte, wäre sinnvoll.

Dennoch ist es unmöglich, außer auf vorbestimmte Weise, den Regelkreislauf zu beeinflussen. Nachträgliche Optimierung und Integration in neue Systeme sind deshalb nicht bzw. nur durch Austausch der Systeme möglich. Auch Fehlerbehebungen und Fehlermeldungen sind wegen eingeschränkter Darstellungsmöglichkeiten des Hauptthermostats nur unzureichend und könnten durch die Integration in andere Systeme erheblich verbessert werden. Ist zum Beispiel der Außenfühler kaputt, bleibt oft nur eine binäre manuelle Regelung übrig, da die Heizung dann immer mit maximaler Leistung heizt. Es wäre angenehm, diesen Regler bis zur Reparatur, die normalerweise nicht selbst durchgeführt werden kann, durch Rekonfiguration der Steuersoftware eine Zeit lang überbrücken zu können.

[2]Die Thermostate werden hier doppelt aufgeführt, da man mit ihnen einerseits eine Temperatur einstellen kann, sie aber andererseits aktiv in den Heizregelkreis des entsprechenden Raumes eingreifen, um die eingestellte Temperatur zu erreichen.

3.2 eHome-Gebäude

Um Smart-Home-Techniken auszuprobieren und verschiedene Hersteller ins Gespräch zu bringen, gibt es einige Smart-Home-Prototypen. Es handelt sich dabei meist um reale Gebäude, die mit einer großen Auswahl unterschiedlicher Sensoren und Aktoren und auch integrierten elektronischen Geräten ausgerüstet sind. Der Großteil dieser Geräte lässt sich über ein Netzwerk oder ein Bussystem abfragen bzw. steuern. In diesem Kontext werden eHome-Dienste, aber auch einzelne integrierte Geräte getestet oder neu entwickelt.

3.2.1 Philips-Homelab

Dieses Labor entstand mit dem Ziel, Benutzerstudien im Bereich der Ambient Intelligence durchzuführen. Die Hausherren legen sehr viel Wert darauf, neue Techniken von Benutzern auszuprobieren zu lassen und durch deren Rückmeldung, den Nutzen der getesteten Dienste zu bewerten. Philips möchte Technik nicht nur deshalb einzusetzen, weil sie möglich ist, sondern ihre Bewältigung und angenehme und sinnvolle Benutzung unterstützen. Deshalb stellt Philips die empirischen Untersuchungen mit Anwendern und deren Interaktion mit Technik sehr in den Vordergrund.

Es geht im Philips-Homelab mehr darum, neue Arten von Diensten bzw. Anwendungen zu ersinnen, als bestehende Dienste sinnvoll zu kombinieren oder Geräte anzusteuern, wie in dem hier besprochenen Arbeitsgebiet oder auch im inHaus. Es soll hier nicht nur eine physikalische sondern auch eine soziale Integration smarter Techniken erreicht werden. In [Aar03] werden Visionen aus drei Bereichen besprochen:

1. Erfahrungs- und Inhalteaustausch, sowie die Erzeugung eines sozialen Zusammenhörigkeitsgefühls
2. Spannung und Entspannung
3. Ausgeglichenheit und Organisation des Alltags

Im ersten Bereich wurden Arbeiten durchgeführt, wie man leicht in einer sozialen Gemeinschaft mediale Inhalte wie Fotos, Filme oder Musik austauschen aber auch zusammen erleben kann.

Im zweiten Bereich wurde die Wirkung der Integration des Umgebungslicht in mediale Erlebnisse oder auch nur die Erzeugung gewisser Stimmungen durch Lichteinflüsse untersucht. So passte sich das Umgebungslicht in Temperatur, Helligkeit und Farbe der gehörten Musik bzw. den Szenen eines gerade laufenden Films an. Auch lassen sich Voreinstellungen des Lichtes abrufen, um das aktuelle Wohlbefinden derjenigen, die sich in dem so beleuchteten Raum aufhalten zu fördern. So können warme Farbtöne für romantische Situationen benutzt werde und kältere, frische Farbtöne in einer Arbeitsumgebung.

Weiterhin wurden in diesem Bereich neue Versionen integrierter Fernbedienungen ausprobiert, mit denen sich alle Geräte eines Hauses auf eine möglichst intuitive Weise steuern lassen sollen. Insbesondere wurden Möglichkeiten untersucht, wie sich der aktuelle Umgebungskontext auf die Benutzerfreundlichkeit auswirkt: zum Beispiel könnte die Fernbedienung erkennen, ob sie innerhalb einer Schublade liegt oder neben dem Fernsehen und sich jeweils unterschiedlich verhalten. So könnte das Legen in die Schublade, die

meisten Funktionen der Fernbedienung und der bedienten Geräte automatisch abschalten. Auch der Nutzen von Erinnerungsfunktionen auf der Fernbedienung wurde untersucht. So konnte die Fernbedienung automatisch an Sendungen erinnern, die zu einem bestimmten vorgegebenen Benutzerprofil passen.

Im dritten Bereich wurden neue Methoden der Speicherung von medialen Daten und der Navigation durch diese erprobt. Diese erlaubten den intuitiven Umgang mit Bilder- und Videosammlungen.

Die Ergebnisse in Benutzerstudien werden alle als durchweg positiv und die untersuchten neuen Dienste als sinnvoll bezeichnet. Trotzdem fehlt eine Gegenüberstellung des nötigen Entwicklungsaufwands gegenüber dem zu erwartenden Nutzen. So wurde auch hier der Aufwand des Softwareentwicklungsprozesses vernachlässigt.

3.2.2 inHaus

Diese Testumgebung ist ein reales Haus eines Kooperationspartners der eHome-Gruppe in Duisburg. Es wurde von inHaus [inH05], eine Abkürzung für integrierte Haussysteme, in enger Zusammenarbeit mit dem Fraunhofer Institut für mikroelektronische Schaltungen und Systeme [Fra] entworfen und gebaut. Unter den industriellen Kooperationspartnern befinden sich Thyssen Krupp, Panasonic und Vorwerk, welche Technik und Wissen für die Realisierung des Projektes zur Verfügung gestellt haben. Die in diesem eHome verwendeten Geräte befinden sich auf dem neusten Stand der Technik, sind jedoch zu großen Teilen noch nicht auf dem Markt verfügbar. Erklärtes Ziel des Projektes ist ein Vorantreiben der Entwicklung in Richtung eines Massenmarktes. Forschung erfolgt zu diesem Zweck auf verschiedenen Ebenen. Zum einen erfolgt eine Betrachtung der technischen Perspektive, in deren Zusammenhang die Vernetzung verschiedener Geräte untersucht wird und mögliche Mehrwertdienste erforscht werden. Es steht aber zum anderen auch die Entwicklung eines einheitlichen und intuitiven Bedienungskonzeptes im Vordergrund. Es werden aber auch betriebswirtschaftliche Faktoren betrachtet. Zu diesen gehören unter anderem Markt- und Akzeptanz-Forschung, aber auch die Suche nach neuen Vermarktungsmöglichkeiten und Marketingstrategien.

Das Haus besteht aus zwei Teilen: dem Wohnhaus und dem Werkstatthaus. Das Wohnhaus ist ein tatsächlich bewohnbares Haus, in dem sehr viele elektronisch steuerbaren Geräte installiert sind. Es ist somit eine ideale Testplattform für die Installation von eHome-Systemen. Zum Wohnhaus gehört das Wohnlabor, bestehend aus Diele, Küche, Wohnzimmer, Schlafzimmer, Kinderzimmer und Bad. Weiterhin gehören dazu ein Home-Office, ein Multimedia-Auto und ein smarter Garten. Die zweite Haushälfte, das Werkstatthaus ist an ein Forschungslabor angelehnt und bietet mehr Arbeitsräume als das Wohnhaus und die Technik-Zentrale. Die Sicht auf den Garten und das Doppelhaus ist in Abbildung 3.1 dargestellt.

Im Wohnhaus befinden sich steuerbare Küchengeräte, Sanitärinstallationen, Lüftungsanlage, Verschlusstechnik, Schaltelemente und viele Geräten der Unterhaltungselektronik bis hin zum Home-Theater. An technischen Installationen befinden sich dort ein European-Installation-Bus, ein Honeywell-Funk-Bus und RFID-Sensoren.

Im inHaus wird ein hoher Aufwand in die Entwicklung zur Steuerung und Abfrage der vorhandenen Bussysteme und der in diesen gehaltenen Geräte investiert. So lassen sich alle Aktoren im inHaus über für OSGi programmierte Dienste ansteuern. Jedoch lassen

Abbildung 3.1: inHaus Duisburg mit Garten [inH05]

sich nicht alle Sensoren auslesen, da manche Bussysteme wie der European-Installation-Bus dies nicht unterstützen.[3]

Ein weiterer Forschungsschwerpunkt liegt in der Entwicklung von Benutzerschnittstellen für diese Geräte im Hauskontext. Diese werden für mobile Geräte wie PDAs geschaffen, so das anpassbare Fernsteuerungen entstehen. Aber auch der Fernzugriff über Handy und aus dem Auto heraus auf alle installierten Geräte soll möglich werden. Auch eine Integration in die Media-Center-Oberfläche des Wohnzimmer-Medien-PCs wurde hier umgesetzt und bietet eine sehr bequeme Möglichkeit, mit allen installierten Geräten vom Wohnzimmersessel aus zu interagieren.

Auf die Integration mehrerer Geräte und Dienste zu neuen Mehrwertdiensten wurde weniger Wert im inHaus gelegt. Es gibt Möglichkeiten Geräte zeitlich zu steuern, Profile für Lichteinstellungen insbesondere Farbwahl festzulegen und Rollladen, Licht und Gartenbewässerung abhängig von Lichteinfall und Temperatur zu steuern, dennoch wird eine domänenübergreifende Integration, wie sie hier in der Arbeit ermöglicht wird, nicht umgesetzt.

Der Entwicklungsprozess wird im inHaus unterstützt, indem versucht wird, wiederverwendbare Komponenten im Sinne des Ansatzes in Kapitel 4 zu entwickeln. Das Forschungsteam im inHaus setzt als Rahmenwerk für seine Dienstentwicklung OSGi ein und

[3]Nach den Erfahrungen der Forschungsgruppe im inHaus ist es beim European-Installation-Bus nicht möglich das Drücken der eingesetzten Schaltern alleine abzufangen, was zum Beispiel bei dem vergleichsweise preiswerten X10-Pendant gar kein Problem ist. Bei einem Test des eHomeConfigurators im inHaus wurde festgestellt, dass mit EIB-Schaltern nur Licht geschaltet wird, aber kein Gerät oder Dienst, welche außerhalb des European-Installation-Bus lagen, bedient werden konnte. Dies war nur möglich, wenn der Schalter mit einem Gerät verknüpft wurde, dessen Status sich dann überwachen ließ. So mussten für die Tests im inHaus die Lego-Taster des Lego-eHome-Demonstrator zur Bedienung eingesetzt werden und die Räume des Lego-eHome-Demonstrators entsprechenden Räumen des inHaus zugeordnet werden.

entwickelt alle Dienste als Bundles dafür. Die Konfigurierung so wie die Konfigurierung mehrschichtiger, komplexer Dienste werden nicht unterstützt.

Die Forschungsgruppe des inHaus stellte freundlicherweise für diese Arbeit eine eigene Schätzung ihres Entwicklungsaufwandes zur Programmierung des Systems zur Verfügung. Die entwickelten Bundles werden in die Klassen Kern+System, Bus+Gerät ,und Applikation unterteilen.

Kern+System: Die Komponenten dieser Gruppe stellen die Basis des Rahmenwerks dar und übernehmen gegebenenfalls Verwaltungsaufgaben.

Dazu gehören: inHausBaseDriver, inHausBaseMapper, EventContainer, HttpTunneling, inHausMain, JavaMail, WindowSystem, eine CORBA-Anbindung, eine MySQL-Anbindung

Bus+Gerät: Diese Komponenten erlauben die Anbindung an verschiedene Bussysteme wie EIB, LON oder X10 und steuern einzelne Geräte und deren Parametrisierung

Implementiert sind hier: AmigoLampProxy, AxisPort, CBNed3858, Cuisinale, EHS-BundleLight, EIB-Bundle, EIB-Configurator, Fountain, HeatingSystem, inHausgarage, inHausProxy, IntercomSystem, Jalousie, Lamp, LightingComode, NevoLink, Oven_H4000, Reader, SecuritySensor, SmartOutletDevice, Sprinkler, ViessmannCalibration, ViessmannErrorAnalysis

Applikation: In dieser Gruppe werden die Dienste aufgeführt, die nicht direkt auf Geräten aufsetzen und evtl. Gerätetreiber mit andern Funktionalitäten erweitern.

Dies sind: inHausApplication, IntelliMed, MCEController, MCEServer, MCE-Steuerung, inHaus Media Center Edition

Für die Erstellung der Kern- und Systemkomponenten nennt die Forschungsgruppe des inHaus ein Personenjahr Entwicklungszeit. Für die Anbindung von EIB und EHS 100 Personentage und für die Anbindung anderer Geräte einen Entwicklungsaufwand von 5 bis 10 Entwicklungstagen pro Komponente, was etwa 150 weitere Personentage ergibt. Für die Applikationen sind die Zahlen sehr ungenau und belaufen sich auf insgesamt etwa 1,5 Personenjahre.

Insgesamt wurde also für die Einrichtung des inHauses, so wie es jetzt zu finden ist, mehr als 3 Personenjahre Entwicklungszeit aufgewendet.

Leider sind diese Daten bisher nicht veröffentlicht worden, so dass sie hier nicht referenziert werden können. Sie wurden auf meine Bitte hin von Peter Gabriel, einem Mitarbeiter der Forschungsgruppe im inHaus, geschätzt und für diese Arbeit zur Verfügung gestellt.

3.2.3 T-Com-Haus

Gemeinsam mit WeberHaus, Siemens und Neckermann hat die Deutsche Telekom ein smartes Haus in Berlin aufgebaut, das T-Com-Haus (siehe auch [TS05, Kre] und Außenansichten in Abbildung 3.2 und 3.3). Es wurde im April 2005 nach nur drei Monaten Bauzeit eröffnet. Es handelte sich hierbei nicht um eine Forschungseinrichtung sondern um ein Präsentationsobjekt, um normale Konsumenten für neuartige Dienste aus dem

Abbildung 3.2: T-Com-Haus aus [TC]

Bereich der Kommunikationstechnik oder andere eHome-Dienste zu sensibilisieren. Die Anlage wurde im Juni 2006 wieder geschlossen[4] und wird nun verkauft.

Im Haus bzw. über das Internet konnte man Dienste wie eine elektronische Pinnwand oder fernsteuerbare Heimgeräte und Überwachungskameras nutzen. Wichtigstes Ziel der Partner war es, eHome-Dienste, die für die meisten Menschen bisher eine Zukunftsvision darstellen, anfassbar zu machen und somit am Markt ein Interesse für diese zu erwecken. Es ging somit nicht darum, technikverliebte Menschen zu begeistern, sondern Möglichkeiten von eHome-Technik für den Alltagseinsatz, also für eine breite Basis von Anwendern, aufzuzeigen.

Wie im inHaus lassen sich auch hier alle Geräte des Hauses sowohl mittels besonderer Fernbedienungen, hier mit einem PDA, als auch über das Internet, steuern. Das ganze wird um ein sogenanntes *Family-Whiteboard* im Eingangsbereich erweitert. Dies ist eine Informationszentrale, an der sich jeder Bewohner mit RFID identifiziert. Jegliche Art von Nachrichten lassen sich hierüber austauschen und abrufen. So kann beim Eintreten eine kurze Video-, Audio- oder Textbotschaft an einen oder mehrere Hausbewohner gesendet werden, die sich im Haus aufhalten, aber auch über das Internet oder per SMS versendet werden, falls sie sich nicht im Haus befinden.

Die Firmen haben bei der Entwicklung sehr viel Wert auf die multimedialen Möglichkeiten im T-Com-Haus gelegt. So befinden sich fast in jedem Raum LCD-Bildschirme, die sowohl den Konsum von Medien, Kommunikation, als auch die Steuerung des Hauses erlauben. Als Fernbedienung zur Auswahl der Inhalte wird hier ein PDA eingesetzt. Die benötigte Software wurde von der T-Com und Siemens gemeinsam entwickelt.

Der Handcomputer übernimmt mit Hilfe der RFID-Technik zudem das so genannte "Mood-Management", das laut der T-Com-Werbung durch die Steuerung des Licht- und Klangambientes "immer für die perfekte Stimmung im Raum" sorgen soll. Über-

[4] Für weitere Informationen zur Schließung siehe [T-C]. Die Webseite unter [T-C05] existiert nicht mehr.

Abbildung 3.3: T-Com-Haus bei Nacht aus [TC]

haupt spielt die Farbgebung laut Berthold Feiertag, Produktmanager bei Neckermann, eine wichtige Rolle im T-Com-Haus: sie soll das "Wohnwohlfühlkonzept" fürs lifestyle-gerechte "Homing" unterstreichen. So herrschen im Wohnzimmer aktive Rot-Töne vor, der Fitnessraum soll mit Pink prickelnd wirken, die Schlafzimmer mit Blau und Gelb mal einschläfern, mal aufmuntern.

Die Vernetzung ist größtenteils über wireless LAN und nach außen hin mit DSL realisiert. DSL wird insbesondere für die Fernsteuerung von außen und für das Herunterladen von Medieninhalten genutzt.

In [T-C] wird der Zweck des T-Com-Hauses auch als Demonstration, wie "moderne Breitbandtechnologie neuartige Wohnerlebnisse und mehr Komfort im privaten Haushalt schafft", beschrieben.

Es geht also auch hier nicht um die Entwicklung domänenüberspannender Dienste geschweige denn einer Unterstützung des Entwicklungs- und eines Konfigurierungsprozesses für diesen, sondern um die Etablierung bestehender Systeme am Markt.

3.2.4 Futurelife Schweiz

Das Futurelife-Haus [Bei05, Zie04, Xne] (siehe Abbildung 3.4) ist ein Element eines Reihenhauses im Kanton Zug in der Schweiz. In dem Haus lebt seit Anfang 2001 die vierköpfige Familie Steiner und nutzt die Dienste, die in diesem Haus angeboten werden.

Im und am Haus befinden sich ein Türöffner mit Fingerabdrucksensor, Jalousien, die auf Zuruf geöffnet und geschlossen werden können, Kameras, ein vernetztes Audiosystem für jedes Zimmer, Beamer zur Filmwiedergabe und vernetzte Haushaltsgeräte. Weiterhin befinden sich dort Putzroboter, die Staub wischen und ein solargetriebener Rasenmäher für den Garten.

An das Haus angepasst ist auch ein Auto, ein BMW der 5er-Serie, der mit dem BMW-

Abbildung 3.4: Futurelife-Haus in der Schweiz im Kanton Zug [Bei05]

Online-Dienst ausgestattet ist und damit den Zugriff auf sämtliche Funktionen im Haus auch von unterwegs ermöglicht. Im Rahmen der ConnectedDrive-Initiative [BMW] engagiert BMW sich im Bereich der Haus-Fahrzeug-Vernetzung. Über einen sogenannten iDrive-Knopf in der Mittelkonsole des Fahrzeugs, WAP-Browser und GSM-Verbindung kann vom Fahrzeug aus auf die Geräte im Haus zugegriffen werden. Alle elektrischen Geräte, wie Kaffeemaschine, Backofen oder Waschmaschine lassen sich so fernsteuern. Auch ein Öffnen der Haustür für Bekannte, die vor dem Haus stehen, ist so möglich.

In diesem Haus gibt es zwei Kühlschränke: einen normalen und die sogenannte Sky-Box. Die Skybox ist von innerhalb und von außerhalb des Hauses zugängig und dient somit als ein Briefkasten für verderbliche Waren. Diese können im Internet bestellt werden. Nach Lieferung werden die Hausbesitzer per SMS oder E-Mail informiert, dass sie eingetroffen sind.

Zur Vernetzung der Geräte werden auch hier – wie im inHaus – EIB, Wifi und Ethernet benutzt. Die Ethernetverkabelung wird zum Ansteuern der zahlreichen Lichtschalter und Kameras im Haus, des Multi-Room-Audiosystems oder der mit dem Beamer verbundenen Videowiedergabeanlage benutzt.

Auch hier wurde die Entwicklung speziell an die spezifische Umgebung angepasst. Ein Prozess, um Dienste einfach zu kombinieren und in neuen Umgebungen einzusetzen, ist auch hier nicht beschrieben. So beziehen sich auch die meisten Dienste nur auf die Ansteuerung einzelner Geräte und nicht auf deren Vernetzung zu neuen Diensten. Im Futurelife-Haus finden sich keine weiteren Erweiterungsdienste als im inHaus.

Nach Ablauf der Förderung des Futurelife-Projekts Ende 2003 wurde das Haus von der Familie Steiner übernommen und wird jetzt von der Futurelife AG [Bei05] unter Leitung von Daniel Steiner weitergeführt. Ziel ist hier die Entwicklung einer Home-Server-Provider-Plattform – mit standardisierten Verknüpfungen zu Residential Gateways. Dies lässt mehr auf Treiberentwicklung als auf die Entwicklung von Erweiterungsdiensten schließen.

Abbildung 3.5: Smarthouse 213 [Sch05a]

3.2.5 smarthouse213

Matthias Schmidt, Maschinenbauingenieur und Unternehmer, ist Bewohner und Eigentümer des Smarthouse213 in Coburg [Sch05a]. Dieses ist ein privates Wohnhaus, welches mit zahlreichen vernetzten Aktoren und Sensoren ausgestattet ist.

Diese hat Herr Schmidt in seiner Freizeit zu einer durchdachten Gesamtlösung verbunden. Das Haus wurde im Jahr 2001/2002 durch das Architekturbüro Archi Viva entworfen und fertig gestellt. Neben seiner geradlinigen Architektur (siehe Abbildung 3.5) sollte es den Stand der Technik der Gebäudeautomation im Einfamilienhaus-Bereich repräsentieren.

Die Elektronik des Gebäudes ermöglicht die Steuerung der Beleuchtung, der Jalousien, Temperaturregelung in einzelnen Räumen und die Aktivierung bzw. Deaktivierung aller Steckdosen. Auch die Betriebsmodi der Gastherme, die Warmwasserbereitung und die Wohnraumlüftungsanlage sind zentral regelbar.

Außerdem wurden folgende Geräte eingebunden: Wetterstation, Überwachungskameras, Telefonanlage, Rauchmeldernetz, Briefkasten (um unnötige Gänge zu vermeiden), Klingelanlage, Zufahrts- und Garagentor, Trockner, Waschmaschine und das Fernsehgerät.

Der Großteil der elektrischen Geräte ist – wie beim inHaus – über den European-Installation-Bus (EIB) [Bec01, BPR00] angebunden. Im Erdgeschoss gruppieren sich Wohnbereich, Essen und Küche um die Informationszentrale. Diese besteht aus einem in einen Wandschrank eingebauten Display mit Maus und Tastatur. Der zugehörige Computerkorpus ist ein Gira-Homeserver [GIR], der sich eine Etage tiefer im Keller befindet. Neben der Gebäudesteuerung laufen auf diesem Computern noch Mailserver, E-Mail-Client, Webserver, Faxserver, digitaler Anrufbeantworter und ein Kalender für die zentrale Terminverwaltung der Familie. Über Internet sind alle Nachrichten und Termine mittels Handy oder Webbrowser abrufbar. Auch die Interaktion mit den installierten Geräten im Haus ist hierüber möglich.

Erweiterungen und Integrationen zwischen den Geräten wurden mit Hilfe einer proprietären Script-Language im Rahmen des Giro-Homeservers von Herrn Schmidt programmiert.

Hier geht es tatsächlich neben der bloßen Gerätesteuerung auch um die Integration von Diensten. Allerdings wird hier nicht im Sinne von Softwaretechnik der Blick auf Wiederverwendbarkeit und somit den Einsatz in anderen Umgebungen gerichtet. Es geschieht eine spezielle Anpassung an das Haus der Familie Schmidt. Herr Schmidt stellt allerdings seinen Quelltext im Internet frei zur Verfügung, so dass Anpassungen möglich sind. Der Prozess, wie hier Komponenten verschaltet und konfiguriert werden, ist nicht genauer definiert, so dass der Einsatz dieser Software in einer anderen Umgebung auf einen erneuten Entwicklungsprozess mit einem geringen Anteil an Wiederverwendung hinauslaufen wird.

3.2.6 Zusammenfassung

Viele der gerade vorgestellten Ansätze zeigen neue Möglichkeiten für den Einsatz von vernetzten und steuerbaren Geräten in der Wohnung und Wohnumgebung wie Garten oder Auto auf. Dabei kristallisieren sich drei Schwerpunkte heraus:

1. **Fernsteuerbarkeit:** Leicht zu implementieren und zu demonstrieren ist es, Geräte, die bereits über ein Netzwerk elektronisch steuerbar und auslesbar sind, auch über das Internet und somit auch über mobile Geräte, die Zugriff auf das Internet haben, fernzusteuern. Forschungen hierzu gibt es im inHaus, T-Com-Haus, Futurelife-Haus und smarthouse213. Allerdings geht es hier meist um die Steuerung von Geräten wie Küchengeräten und weniger um die Steuerung von Erweiterungsdiensten.

2. **Zentrale Bedienbarkeit, einheitlich Benutzerschnittstelle:** Ein weiterer Schwerpunkt ist die Möglichkeit, Geräte von einer zentralen Stelle aus zu bedienen und wenn möglich mit einem eingängigen Benutzerinterface anzubieten. Dies geschieht im inHaus über das Mediacenter, im T-Com-Haus über das Family-Whiteboard und einem besonders konfigurierten PDA und im smarthouse213 über die Informationszentrale

3. **Multimedia:** Die Verbesserung des Umgangs mit Multimedia scheint eines der bedeutendsten Forschungsgebiete überhaupt zu sein. In jeder der vorgestellten eHome-Umgebungen wird versucht, den Komfort beim Zugriff auf und das Abspielen von Video und Audio einfacher und erlebnisreicher zu gestalten. Dies scheint auch der Sektor zu sein, in welchem sich momentan am meisten Geld verdienen lässt, wie auch in dem Strategiepapier von Microsoft zu lesen ist, welches kurz in Abschnitt 8.1.7 auf Seite 248 besprochen ist.

Auffallend ist, dass nur sehr wenig Dienste entwickelt wurden, in denen Geräte aus unterschiedlichen Domänen, wie in Abschnitt 2.1.2 unter dem Punkt Dienstkategorien 2.1.2 auf Seite 21 definiert, zu neuen Diensten kombiniert oder gar Dienste selbst zu neuen Diensten kombiniert werden. Erahnen, was durch die Kombination möglich werden kann, lässt sich bei einem Blick auf die Energiedienste, die meistens gemessene Umgebungs- und Verbrauchsdaten benutzen, um Licht oder Temperatur regelnde Geräte zu steuern.

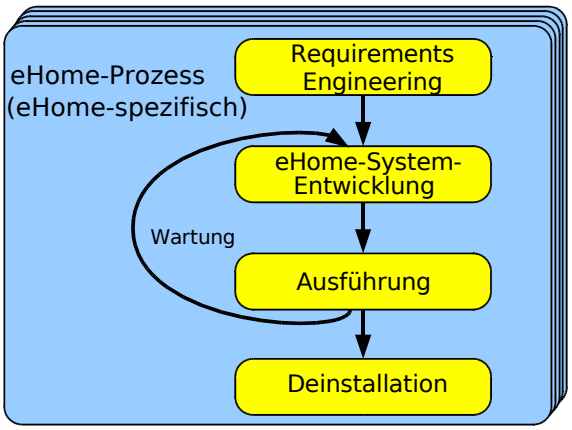

Abbildung 3.6: Der klassische eHome-Entwicklungsprozess

Ein Bezug zum Softwareentwicklungsprozess ist nur im inHaus mit der komponentenbasierten Softwareentwicklung für OSGi zu erkennen. Eine Unterstützung der Konfigurierung der einzelnen Dienste oder gar die Schaffung der Möglichkeit von kombinierten Diensten wird in keinem dieser Forschungsprojekte betrachtet.

3.3 Klassischer Entwicklungsprozess

Momentan werden eHome-Systeme individuell für jeden neuen Kunden bzw. jede neue eHome-Umgebung einzeln neu in einem klassischen Entwicklungsprozess entwickelt (siehe Abbildung 3.6). Dieser klassische Entwicklungsprozess impliziert für jedes eHome-System in einer neuen Umgebung die folgenden Schritte:

- **Anforderungsanalyse (Requirements Engineering):** Der Kunde muss im Vorfeld entscheiden, welche Dienste er in seiner Wohnung realisiert haben möchte und welche Funktionalitäten diese abdecken sollen.
- **eHome-System-Entwicklung (eHome system development):** Diese Dienste müssen implementiert werden. Dabei muss sowohl die Realisierung der Software als auch die zu benutzende Hardware, deren Vernetzung und Ansteuerung berücksichtigt werden. Es müssen also die Geräte, die zu benutzen sind, bestimmt werden, deren Ansteuerung und Vernetzung realisiert werden, sowie eine an die räumlichen Vorstellungen des Kunden angepasste Dienstlogik entwickelt werden. Dies erfordert eine enge Zusammenarbeit zwischen Hard- und Softwareanbietern. Die entstehende Lösung ist handkodiert und auf den Kunden bzw. dessen Umgebung zugeschnitten.
- **Ausführung und Wartung (execution and maintainance):** Die Software wird vom Anbieter installiert und deployed. Spätere Änderungs- oder Erweiterungswün-

sche erfordern weitere Kodierungs- und Entwicklungszeit, wie im vorhergehenden Schritt beschrieben.

- **Deinstallation:** Die Software-Komponenten werden de-installiert und der Kontakt zwischen Kunde und Provider endet.

Der Gedanke der Wiederverwendung wird hier nicht hervorgehoben. Natürlich können bei diesem klassischen Prozess Komponenten oder Module entstehen, die verwendet werden können, dies ist aber bei den hier beschriebenen Projekten eher Zufall. Eine Konfigurierungsunterstützung wird nicht geboten.

Deshalb wird im folgenden Kapitel (Kapitel 4) zunächst die komponentenbasierte Entwicklung untersucht und diese im darauf folgenden (Kapitel 5) mit einem Konfigurierungsansatz unterstützt.

Kapitel 4

Manuelle Konfigurierung

Ein erster Schritt zur Vereinfachung des klassischen Entwicklungsprozesses ist die Anwendung von komponentenbasierter Programmierung.[1] Dadurch wird es möglich, Teile der Entwicklung in anderen Kontexten wiederzuverwenden. Dennoch müssen Abhängigkeiten und Parameter der Komponenten für den einzusetzenden Kontext festgelegt werden. Diese müssen hier in Hinblick auf eine Konfigurierungsunterstützung festgehalten werden. Dieses Kapitel zeigt auf der einen Seite, wie komponentenbasiert für eHome-Systeme entwickelt werden kann, aber auch, in wie weit bestehende Komponentenrahmenwerke eine manuelle Konfigurierung, also speziell die Parametrisierung und die Abhängigkeitsbeschreibung, unterstützen. Insbesondere wird dabei überprüft, wie gut sich eine manuell erstellte Konfigurationsbeschreibung deployen lässt.

Anhand des vereinfachten Sicherheitsdienstes (siehe Kapitel 3 auf Seite 27) wird der komplette komponentenbasierte Entwicklungsprozess anhand dreier unterschiedlicher Rahmenwerke durchgeführt. Ziel ist es hierbei, aus den Konfigurierungs- und Deployment-Mechanismen der drei Rahmenwerke zu lernen, ein geeignetes Rahmenwerk auszuwählen und die nötigen Abstraktionen für eine Konfigurierungsunterstützung zu finden. Es werden die Rahmenwerke Rio [Den04, Ree03], Openwings [Gen] und OSGi [Ope] betrachtet.

Für die Implementierung des vereinfachten Sicherheits-Dienstes werden fünf Komponenten realisiert. Siehe dafür Abbildung 4.1. Die Komponenten `SirenProvider` und `CameraProvider`, dienen dazu die entsprechenden Geräte anzusteuern. Die Komponente `EMailProvider` ermöglicht es, E-Mails, die eine Bilddatei enthalten können, zu versenden. Die Komponente `MotionPublisher` dient dazu, Ereignisse, die von einem Bewegungsmelder beobachtet wurden, an andere Komponenten weiterzuleiten, die dann entsprechend reagieren können. Die Dienstlogik des vereinfachten Sicherheits-Dienstes wurde in der Komponente `SecurityProvider` implementiert.

Jede der fünf Komponenten enthält eine Klasse, von der zur Laufzeit Objekte erzeugt werden. Damit diese Klasse und deren Objekte benutzt werden können, implementieren sie jeweils eine Dienstschnittstelle. Diese ermöglicht die Verwendung des Objektes.

[1] Heutzutage wird komponentenbasierte Softwareentwicklung nur noch selten erwähnt und häufig unter dem Begriff der Service-orientierten Programmierung verwendet. Es steht dem Leser frei, sich diese an Stelle von komponentenbasierter Programmierung hier zu denken.

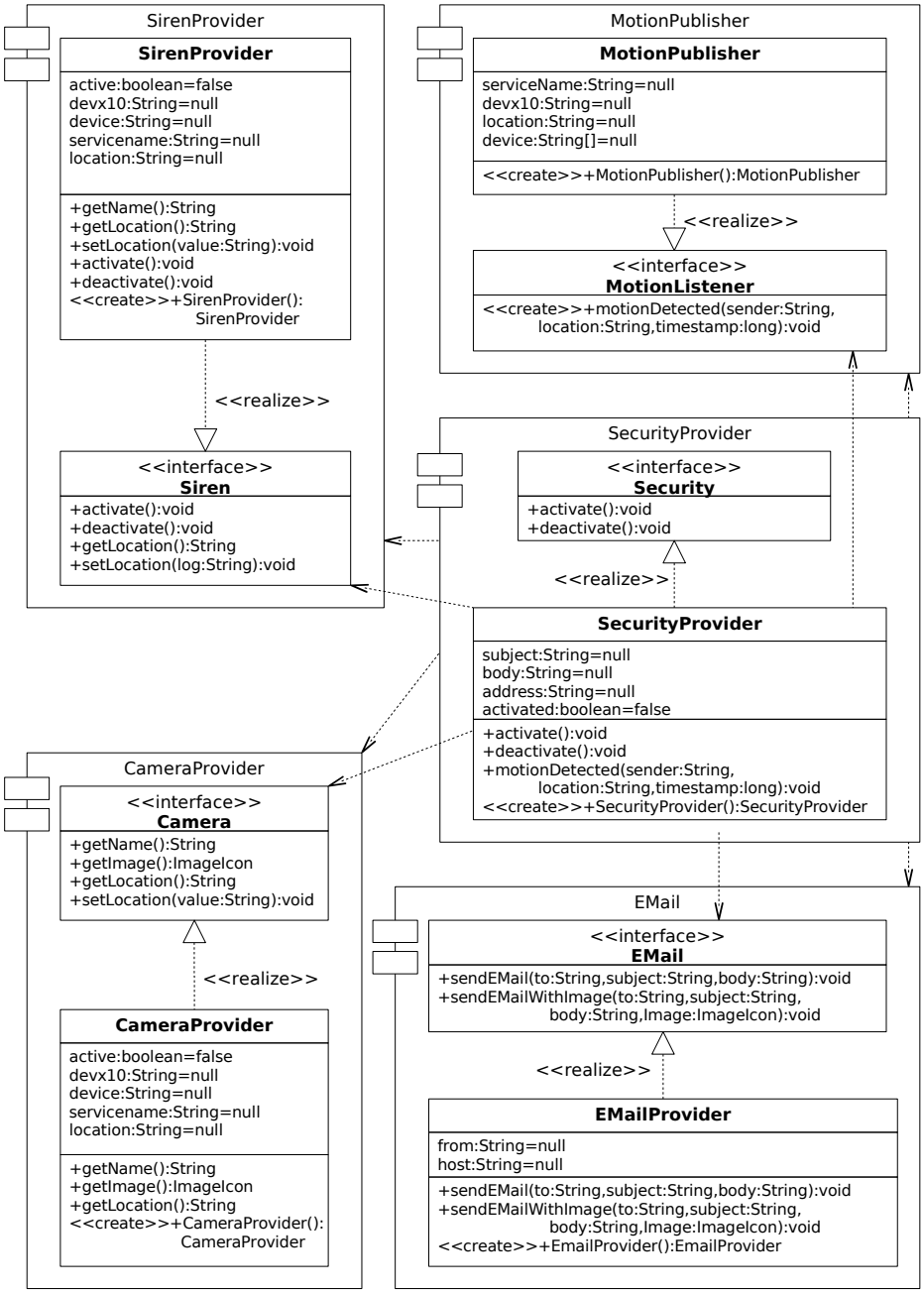

Abbildung 4.1: Komponentenübersicht: Vereinfachter Sicherheits-Dienst

Objekte, die die Schnittstelle `Camera` implementieren, werden als `Camera`-Dienstobjekte bezeichnet. Entsprechend werden die Bezeichnungen `Motion`-Dienstobjekt, `Security`-Dienstobjekt, `EMail`-Dienstobjekt und `Siren`-Dienstobjekt verwendet.

Bei der Instanzierung eines solchen Dienstobjekts müssen die Attribute der Klasse bzw. Parameter des Dienstobjektes, mit Werten belegt werden. Bei der Erzeugung eines Camera-Dienstobjekts (von der Klasse `CameraProvider`) sind dies die Attribute `servicename`, `location` und `url`. Um die Werte der Attribute zu bestimmen, die Objekte also zu parametrisieren bzw. zu konfigurieren, werden sogenannte Konfigurationsdaten benutzt. In Abbildung 4.2 sind die in dieser Arbeit benutzten Konfigurationsdaten angegeben. Dies sind also Daten die möglichst in einer Konfigurierungsunterstützung automatisch abgefragt werden sollten und in der *Konfiguration* vor dem Deployment gespeichert werden müssen.

Abhängig von der Implementierung der Klasse existieren für einige Attribute auch Standardwerte, die nicht angegeben werden müssen. Ist in der Konfiguration kein Wert für ein solches Attribut angegeben, wird es mit einem Standardwert belegt. Solche Attribute sind in Abbildung 4.2 *kursiv* gedruckt. Wird zum Beispiel in der Konfiguration des `Security`-Dienstobjekts für das Attribut `Address` kein Wert angegeben, werden keine E-Mails verschickt. Das Attribute `active` des `Siren`-Dienstobjekts wird nicht in der Konfiguration angegeben. Sein Wert wird bei demjenigen X10-Gerät abgefragt, das als Sirene benutzt wird.

Einige Attribute werden benutzt, um mit einem Dienstobjekt ein bestimmtes Gerät anzusprechen. Die `Camera`-Dienstobjekte benutzen die im Attribut `Url` angegebene Url, um Bilder einer bestimmten Kamera anzufordern. Attribute können also benutzt werden, eine Beziehung zwischen einem Gerät und den Dienstobjekten, die von ihm abstrahieren, herzustellen.

Alle Dienstobjekte besitzen das Attribut `Name`, welches zur Identifizierung von Dienstobjekten benutzt wird. Dienstobjekte werden durch dieses Attribut unterschieden. Dies ist notwendig, damit Beziehungen zwischen Kameras und Bewegungsmeldern definiert werden können. Die Bezeichnungen der anderen Dienstobjekte werden für die Implementierung des vereinfachten Sicherheits-Dienstes nicht benötigt. Bei diesem einfachen Dienst werden Attribute wie Ortsinformationen vernachlässigt. Zum Beispiel wird die physikalische Position, an der sich eine Kamera befindet, nicht untersucht. Diese werden im nächsten Kapitel bei der Erstellung des generischen eHome-Modells berücksichtigt. Damit wird es zum Beispiel dann möglich, eine Kamera in der Küche anzusprechen, wenn ein dort installierter Bewegungsmelder eine Bewegung registriert.

Im Folgenden werden diese Dienste in drei Rahmenwerken umgesetzt. Die Rahmenwerke werden zuerst jeweils ausgiebig beschrieben, insbesondere auf ihre Fähigkeiten einer möglichen Unterstützung der eHome-System-Entwicklung und Konfigurierung hin, und anschließend wird der vereinfachte Sicherheitsdienst in ihnen umgesetzt und implementiert.

Motion-Dienstobjekt	
Name	Motion1
Location	Wohnzimmer
Device	a1
DevX10	/dev/x10

(a) Motion1

Motion-Dienstobjekt	
Name	Motion2
Location	Küche
Device	a3
DevX10	/dev/x10

(b) Motion2

Siren-Dienstobjekt	
Name	Siren
Location	außen
Device	b6
DevX10	/dev/x10
active	

(c) Siren

EMail-Dienstobjekt	
Name	EMail
From	phd.myhouse@mail.ulno.net
Host	localhost
Tempdir	/tmp

(d) EMail

Camera-Dienstobjekt	
Name	Camera1
Location	Wohnzimmer
URL	http://myhouse.ulno.net/cam1.jpg

(e) Camera1

Camera-Dienstobjekt	
Name	Camera2
Location	Küche
URL	http://myhouse.ulno.net/cam2.jpg

(f) Camera2

Security-Dienstobjekt	
Name	Security
Address	phd.ehomeconfig@mail.ulno.net
Subject	Security-Service
Body	Alarm wurde aktiviert.
activated	true

(g) Security

Abbildung 4.2: Attribute der benutzten Dienstobjekte

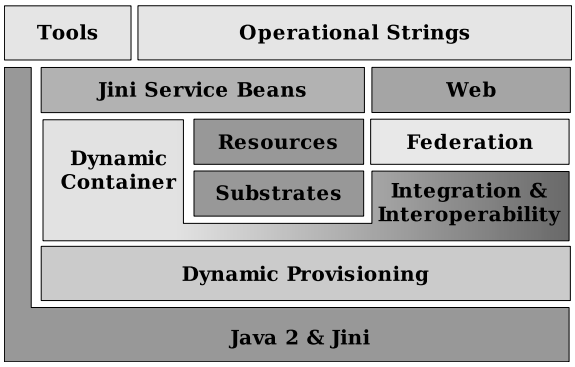

Abbildung 4.3: Rio Übersicht

4.1 Rio

4.1.1 Beschreibung

Der Hauptverantwortliche im *Rio*-Projekt [Den04, Ree03] ist Dennis Reedy von Sun Microsystems Incorporated. Weiterhin beteiligt sich die Firma Gigaspace Technologies [Gig00] an diesem Projekt. Rio ist eine Weiterentwicklung von Jini [Sun99a]. In Abbildung 4.3 sind die im Grundelemente des Rio-Rahmenwerks dargestellt.

Interessant sind hier die Konzepte der *Jini-Service-Beans (JSB)* [Rio03] und der *Operational-Strings*. Dabei stellen Jini-Service-Beans die Komponenten dar und Operational-Strings eine Zusammenfassung einiger von ihnen zu Modulen und deren Konfiguration.

Der Begriff Jini-Service-Beans deutet bereits die Ähnlichkeit dieser zu JAVA-Beans [Sun97] an. Rio stellt eine Menge von Klassen und Schnittstellen bereit, um dieses Konzept zu nutzen. Zusätzlich lässt Rio auch die Verwendung von Klassen aus dem Jini-Kontext zu.

Mittels eines Operational-Strings werden eine Menge von JSBs, deren Parametrisierung und deren Beziehungen untereinander beschrieben. In der Sprechweise von Rio ist ein Operational-String die Beschreibung eines Dienstes, der im Rio-Rahmenwerk instanziert werden soll. Die Operational-Strings sind also in der Sprechweise dieser Arbeit die Konfiguration für einen Dienst und seine Unterdienste. Sie eignen sich somit, um die Deploymentinformationen für eHome-Dienste bereitzustellen.

Instanzen von JSBs werden in Rio auf *Dynamic Containers* ausgeführt. Dynamic Container erlauben das Ausführen der JSBs in Abhängigkeit einiger Quality-of-Service-Parameter wie Speicher, Rechenleistung, Rechnerarchitekturen oder Betriebssysteme der beteiligten Geräte. Solche Geräte könnten zum Beispiel dann Server, Mobiltelefone, Handhelds und Küchengeräte sein.

Die Dynamic Containers werden in RIO von *Cybernodes* implementiert. Auf einem Gerät müssen ein Cybernode und ein http-Server ausgeführt werden, um seine Rechenleistung verfügbar zu machen. Dazu muss das Gerät über eine JAVA-Virtual-Machine (JVM) und JAVA-2-RMI-Bibliotheken verfügen. JAVA Microedition [Suna] oder ähnliche

JAVA-Varianten reichen nicht aus. Eine solche JVM ist aber selbst bei leistungsfähigen Thin-Clients oft nicht verfügbar [JN03].

Prinzipiell können in einem Netzwerk beliebig viele Cybernodes miteinander verbunden werden. Mit Hilfe der Quality of Service Parameter in den Operational-Strings lässt sich allerdings nur schwer steuern, auf welchem Cybernode Dienstobjekte ausgeführt werden. Um sicherzustellen, dass sie in einem bestimmten Cybernode ausgeführt werden, muss im Operational-String explizit die Bezeichnung des Cybernodes angegeben werden, auf dem dieses Dienstobjekt ausgeführt werden soll.

Eine Möglichkeit, Dienste von Geräten im Netz zur Verfügung zu stellen, wäre, auf diesem Gerät selbst einen Webserver, einen Cybernode und in diesem ein Dienstobjekt zu starten. Geräte, die in Zukunft genug Rechenleistung für so etwas anbieten könnten, wären vielleicht eine Waschmaschine oder eine Heizungsanlage. Natürlich wäre es dann sinnvoll in diesem Cybernode nur Dienstobjekte zu installieren, die den Dienst des Geräts selbst verfügbar machen. Es ist wahrscheinlich nicht sehr sinnvoll, die Steuerung für die Waschmaschine auf einer Kaffeemaschine oder auf einem Handy laufen zu lassen.

Dynamic Provisioning erlaubt es, JSBs dynamisch auf den verfügbaren Cybernodes im Netzwerk zu instanzieren. Welche Ressourcen eine JSB braucht, wird bei der Beschreibung einer JSB im Operational-String angegeben. Dadurch können Ressourcen, die im Netzwerk vorhanden sind, effizient genutzt werden. Dies wird von einem Dienst namens *Provision-Managern* im Rio-Rahmenwerk durchgeführt.

Die Bibliotheken, die den von einer JSB benötigten Code enthalten, müssen über einen Http-Server bereitgestellt werden. Der kleine Http-Server *Webster* ist in Rio enthalten.

Als weitere Hilfsmittel stellt Rio Events, Watches und Resource Pools bereit:

Events bieten eine einfache Möglichkeit, Nachrichten zu versenden, die von beliebig vielen Empfängern empfangen werden können. Zur Benutzung von Events bietet Rio ein eigenes Peer-to-Peer Event-Modell an, das in dieser Arbeit zur Kommunikation zwischen dem Security-Dienstobjekt und den Motion-Dienstobjekten benutzt wurde.

Watches können von Entwicklern verwendet werden, um die Performance und die Häufigkeit der Benutzung von Dienstobjekten einer JSBs zu messen. Sie werden in dieser Arbeit nicht verwendet. Das Werkzeug zur Auswertung dieser Watches ist *Watchsmith*.

Resource Pools können in Rio für Threads, Objekte und Datenbankverbindungen angelegt werden. Auch dieser Dienst wird in dieser Arbeit nicht verwendet.

Weitere Rahmenwerksdienste sind die *Rio Substrates*, welche Zugriff auf JavaSpaces, CORBA, Service Control Adapters und Federation (Lincoln) bieten:

JavaSpace wurde als eigenständiger Dienst für JAVA entwickelt und ist mittlerweile fester Bestandteil von Jini [Edw00]. Ein JavaSpace kann mehrere Spaces zur Verfügung stellen. Ein Space ist ein persistenter Speicher für Java-Objekte. JavaSpace stellt Methoden bereit, um Objekte in einem Space abzulegen, zu aktualisieren, oder aus einem Space zu löschen. Außerdem kann nach Objekten innerhalb eines Space

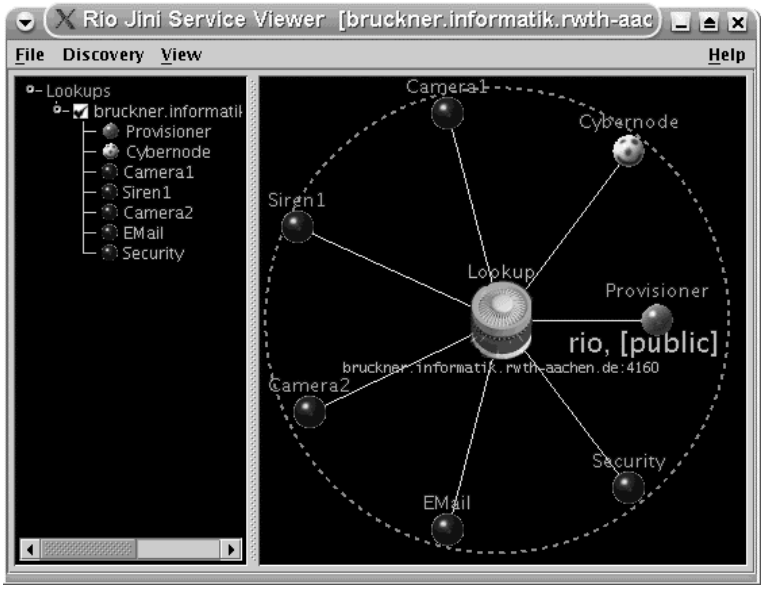

Abbildung 4.4: Rio-Viewer

gesucht werden. Auf einen Space können mehrere Dienstobjekte zugreifen. Diese können den Space benutzten, um miteinander zu kommunizieren. Ein Dienstobjekt kann Objekte innerhalb des Space ablegen, die andere Objekte nutzen können.

CORBA Support erlaubt den Zugriff auf CORBA-Objekte [Obj02].

Service Control Adapter (SCA) ermöglichen es, Dienste, die nicht innerhalb des Rio-Rahmenwerks laufen, zu kontrollieren und zu überwachen [Rio03]. So lassen sich zum Beispiel Hardware-nahe Dienste, die sich nicht in Java realisieren lassen, einbinden. Sie werden in dieser Arbeit nicht verwendet.

Lincoln ist der Jini-Discovery-Mechanismus, um Dienste aufzuspüren und um Look-up-Dienste zu finden. Er funktioniert nur innerhalb von Netzen, die Multicast-fähig sind. Dies ist meist nur innerhalb eines lokalen Netzes der Fall. Dienste die außerhalb dieser Grenzen liegen, können nicht gefunden werden.

Daneben stellt das Rahmenwerk noch zwei Tools bereit, die zur Überwachung des Rahmenwerks verwendet werden können. Der Rio-Viewer (Abbildung 4.4) zeigt alle im Netz erreichbaren Dienstobjekte an und der Operational-String Monitor (Abbildung 4.5) bietet die Möglichkeit, Operational-Strings zu laden und an einen Provision-Manager zu übergeben.

Bei der Beschreibung einer JSB wird angegeben, wie viele Dienstobjekte einer JSB erzeugt werden sollen. Bei der Benutzung einer JSB kann anders als in Openwings (Abschnitt 4.2) oder OSGi (Abschnitt 4.3) nicht vorhergesagt werden, welches Dienstobjekt ein benutzendes Dienstobjekt bekommt. Deshalb dient die Möglichkeit mehrere Dienstobjekte von einer JSB zu erzeugen nur dem Lastausgleich. In dieser Arbeit wurde von

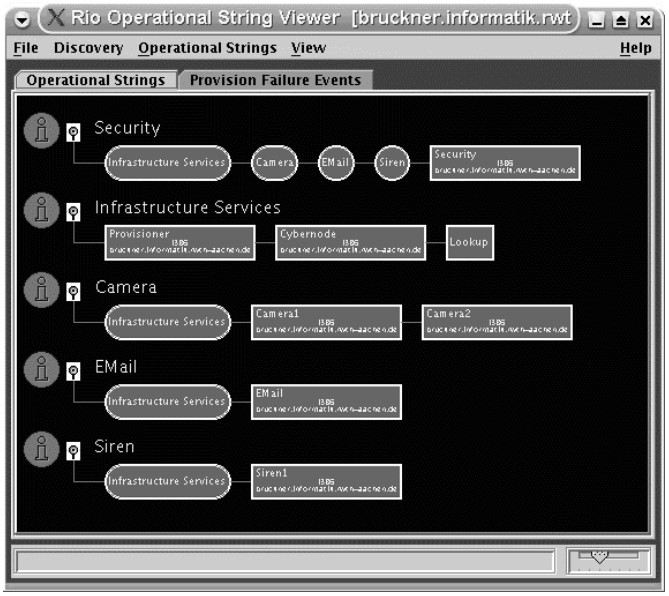

Abbildung 4.5: Rio Operational-String Manager

jeder JSB nur ein Dienstobjekt erzeugt. Im eHome-Bereich ist es meistens nicht sinnvoll mehrere Dienstobjekte von einer JSB zu erzeugen. Das Rahmenwerk selbst sorgt dafür, dass Dienstobjekte einer JSB erzeugt werden. Dazu benutzt es die Methoden der Schnittstelle

Eine JSB besteht aus:

1. Mehreren synchronen Dienstschnittstellen, die alle von der JSB erzeugten Dienstobjekte implementieren.

2. Einer Klasse, die alle Dienstschnittstellen der JSB implementiert, und vom Rahmenwerk benutzt werden kann, um Dienstobjekte zu erzeugen.

3. Bibliotheken, die benutzt werden, um die JSB zu implementieren. Außerdem wird eine Bibliothek benötigt, die eine Klasse enthält, um *Proxy-Objekte* zu erzeugen. Proxy-Objekt werden benutzt, um über ein Netzwerk mit einem Dienstobjekt zu kommunizieren.

4. Konfigurationsdaten, die zur Parametrisierung einzelner Dienstobjekte einer JSB benutzt werden. Dadurch erhält jedes Dienstobjekt dieselben Konfigurationsdaten.

JSBs werden im Operational-String beschrieben. Die Dateien, die zu einer JSB gehören, können von mehreren JSBs gleichzeitig benutzt werden. So kann also ein Jar-Archiv von mehreren JSBs benutzt werden. Jedes Dienstobjekt wird in einer Jini-Gemeinschaft angemeldet.

In Rio können synchrone Dienstschnittstellen definiert werden, die die synchrone Kommunikation zwischen JSBs ermöglichen. In Rio bieten alle Dienstobjekte einer JSB

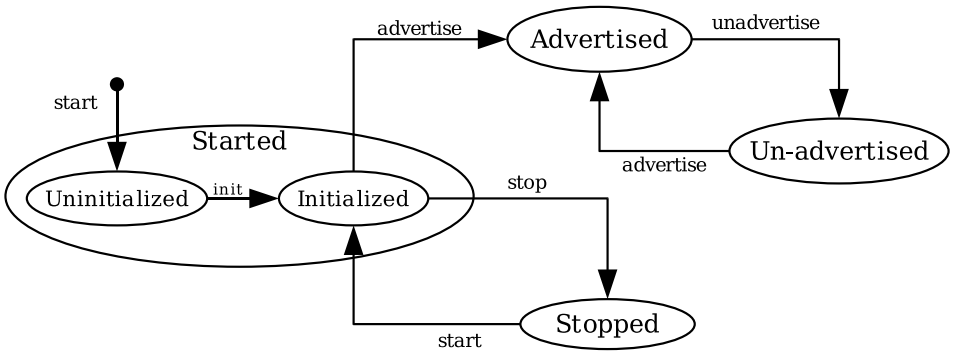

Abbildung 4.6: Lebenszyklus von Dienstobjekten einer JSB

dieselben synchronen Dienstschnittstellen an. In Openwings und OSGi können Dienstobjekte einer Komponente unterschiedliche Dienstschnittstellen anbieten. Damit eine Schnittstelle als synchrone Dienstschnittstelle einer JSB verwendet werden kann, muss sie von der von Rio definierten Schnittstelle `Service` abgeleitet werden. Die Methoden dieser Schnittstelle dienen der Verteilung und Überwachung von JSBs innerhalb des Rio-Rahmenwerks. Sie werden als Klasse `ServiceBeanAdapter` bzw. deren Oberklasse `ServiceProvider` implementiert. Besitzt eine JSB keine eigene synchrone Dienstschnittstelle, da sie ihren Dienst im Hintergrund verrichtet oder nur Events verschickt, muss sie trotzdem alle Methoden der Schnittstelle `Service` implementieren, um innerhalb des Rahmenwerks initialisiert und verwaltet zu werden.

In der Schnittstelle einer JSB kann der Entwickler beliebige Methoden spezifizieren, die den Dienst, den die JSB anbietet, nach außen nutzbar machen. Diese Methoden müssen so spezifiziert werden, dass sie eine Ausnahme vom Typ `RemoteException` auslösen können. Dies zwingt den Entwickler, über die möglichen Probleme bei der verteilten Programmierung nachzudenken. Ein Nachteil dabei ist, dass auch innerhalb der implementierenden Klasse diese Ausnahmen behandelt werden müssen, obwohl sie hier gar nicht auftreten können, da der Aufruf lokal geschieht.

Rio unterstützt keine asynchronen Dienstschnittstellen. Dafür benutzt Rio eine festgelegte Methode, um Events zu verschicken. Es bietet dafür eine Schnittstelle zum Erzeugen und eine zum Empfangen von Events (`RemoteServiceEventListener`), an.

Das Rahmenwerk erzeugt und verwaltet Dienstobjekte einer JSB, um die Erreichbarkeit des von einer JSB angebotenen Dienstes sicherzustellen. Dazu werden die Methoden der Schnittstelle `ServiceBean` benutzt. Diese Schnittstelle spezifiziert Methoden, mit denen der Zustand eines Dienstobjekts verändert werden kann. In Abbildung 4.6 ist der Lebenszyklus eines Dienstobjekts abgebildet. Die möglichen Zustände eines Dienstobjekts sind: `Uninitialized`, `Initialized`, `Started`, `Advertised`, `Stopped`, `Unadvertised` und `Aborted`.

In der Beschreibung einer JSB im Operational-String kann eine Menge von Parametern angegeben werden. Alle Dienstobjekte, die von einer JSB erzeugt werden, besitzen dieselben Parameter. Diese können vom Komponenten-Entwickler benutzt werden, um Dienstobjekte zu initialisieren.

Eine JSB erzeugt und initialisiert ein Event-Objekt einer Event-Klasse, um ein Event auszulösen (bzw. eine Nachricht zu versenden), das von anderen JSBs empfangen werden kann. Die einfachste Event-Klasse ist `RemoteServiceEvent`. Diese kann benutzt werden, wenn das Event keine weiteren Informationen enthält. Eigene Event-Klassen müssen von dieser Klasse abgeleitet werden. Jede Event-Klasse besitzt eine ID. Dadurch können Events gefiltert werden, und ein Event-Listener erhält nur die Events, die ihn interessieren. Das Filtern von Events mithilfe des Typs eines Events ist in Rio nicht möglich.

Die Implementierung des JSBs wird in Bibliotheken (jar-Archiven) bereitgestellt, die vom Rahmenwerk über einen Http-Server heruntergeladen werden. Die Bibliotheken können in zwei Gruppen unterteilt werden. Die erste besteht aus Bibliotheken, die Klassen enthalten, die das Dienstobjekt selbst benutzt. Die zweite besteht aus Bibliotheken, die Klassen enthalten, die benötigt werden, um ein Dienstobjekt zu benutzen. Die erste Gruppe wird als Ressource im Codebase-Bereich eines Operational-String spezifiziert, die zweite im Interface-Bereich. Native Bibliotheken werden dabei in ein jar-Archiv integriert. Daneben gibt es noch eine Bibliothek, die Klassen enthält, die für die Darstellung einer graphischen Benutzeroberfläche benutzt werden.

In Rio ist es problematisch ein Update eines jar-Archivs zu machen, da Rio nur die Möglichkeit besitzt, jar-Archive über einen Http-Server herunterzuladen. Die JVM lädt ein jar-Archiv nur einmal von einer Http-Quelle herunter. Danach wird es nur noch aus dem Cache geholt.

Ein Operational-String wird textuell als XML-Datei repräsentiert. Er besitzt einen eindeutigen Namen und wird auch nur einmal instanziert. Werden einem Operational-String weitere JSBs hinzugefügt, werden diese ebenfalls instanziert. Änderungen an der Beschreibung einer JSB werden ignoriert. Dies ist im Kontext von eHome-Systemen ein Nachteil von Rio, da dadurch eine Rekonfiguration der Parameter im laufenden Betrieb nicht möglich ist. Um die Konfigurationsdaten einer JSB anzupassen, muss zunächst der gesamte Operational-String deinstalliert und anschließend wieder installiert werden.

Zur Laufzeit wird ein Operational-String als Objekt der Klasse `OperationalString` repräsentiert. Jedes Element der textuellen Darstellung besitzt eine Entsprechung als Klasse oder Schnittstelle des Rio-Rahmenwerks. Zur Verwaltung eines Operational-Strings benutzt Rio die Klasse `OperationalStringManager`.

Operational-Strings bestehen aus mehreren ServiceBean-Elementen und weiteren eingebetteten Operational-Strings. Eine JSB wird in einem ServiceBean-Element beschrieben. In diesem sind alle Information enthalten, die zur Installation einer JSB notwendig sind. Daneben enthält ein ServiceBean-Element noch Informationen darüber, wie viele Dienstobjekte einer JSB erzeugt werden sollen.

Eingebettete Operational-Strings sind wieder vollwertige Operational-Strings, so dass Operational-Strings rekursiv aufgebaut werden können. Eingebettete Operational-Strings werden bei der Spezifikation von Beziehungen benutzt.

Ein Operational-String enthält unter anderem die im Folgenden aufgelisteten Elemente (vergleiche [Ree02, Rio03]):

Interfaces: Enthält eine Auflistung der synchronen Dienstschnittstellen, über die mit der JSB kommuniziert werden kann. Unter Ressourcen werden hier alle Bibliotheken angegeben, die zur Benutzung von Dienstobjekten der JSB notwendig sind.

Component: Wird benutzt, um die Klasse anzugeben, die zur Erzeugung von Dienstobjekten verwendet werden soll. Hier werden unter Ressourcen, die Bibliotheken angegeben, die von der JSB-Implementierung selbst benutzt werden.

Groups: Hier werden alle Gruppen angegeben, in denen die erzeugten Dienstobjekte genau wie andere Jini-Dienste auch angemeldet werden sollen. Dabei benutzt Rio dasselbe Gruppenkonzept wie Jini.

Associations: Hier werden die JSBs angegeben, zu denen eine Beziehung besteht. Soll eine Beziehung zu einer JSB aufgebaut werden, muss diese entweder im selben Operational-String oder in einem eingebetteten Operational-String definiert werden.

SLA: Service Level Agreements (SLAs) sind Teil des Quality of Service Konzepts. Dieses besteht einerseits darin, immer genügend Ressourcen für die von einer JSB instanzierten Dienstobjekte bereitzustellen, aber auch darin, immer genügend Dienstobjekte einer JSB anzubieten, sodass eine JSB, die eine andere JSB nutzen will, immer ein Dienstobjekt der JSB erhalten kann, welches noch nicht ausgelastet ist.

Parameters: Dies sind (Key,Value)-Paare, die dazu verwendet werden können, die Dienstobjekte einer JSB zu konfigurieren.

Maintain: Gibt an, wie viele Dienstobjekte von einer JSB erzeugt werden sollen. Dabei besitzen alle Dienstobjekte dieselben Parameter und Attribute. Zur Laufzeit können unterschiedliche Dienstobjekte sich durch die Werte von Attributen der Objekte, die sie benutzen, unterscheiden. Unterschiedliche Dienstobjekte können auch unterschiedliche Zustände besitzen.

Comment: Hier kann beschreibender Kommentar zu einer JSB abgelegt werden. Auf den Kommentar kann während der Laufzeit zugegriffen werden.

In Operational-Strings können *Beziehungen* (*Associations*) zwischen JSBs definiert werden. Es gibt zwei Arten von Beziehungen: schwache `uses`- und starke `requires`-Beziehungen. Die Beziehung A `uses` B bedeutet, dass JSB A die JSB B benutzen kann. Die Beziehung A `requires` B bedeutet, dass JSB A die JSB B benötigt, um funktionieren zu können. Beziehungen können nicht zwischen einzelnen Dienstobjekten geknüpft werden, sondern nur zwischen JSBs.

Bei der Installation einer JSB wird ein Association-Manager gestartet, der die Beziehungen der JSB verwaltet. Die Methoden eines Association-Managers sind in der Schnittstelle `AssociationManagement` spezifiziert und in `AssociationMgmt` implementiert.

Eine Beziehung kann die Zustände `Pending` und `Established` besitzen (Abbildung 4.7). Zunächst befindet sich eine Beziehung im Zustand `Pending`. Besteht eine Beziehung zwischen zwei JSBs A und B, dann versucht der Association-Manager jedes Dienstobjekt der JSB A mit einem Dienstobjekt der JSB B zu verbinden. Dabei wird die Auslastung der Dienstobjekte beachtet. Gelingt das Verbinden, wird die Beziehung in den Zustand `Established` versetzt.

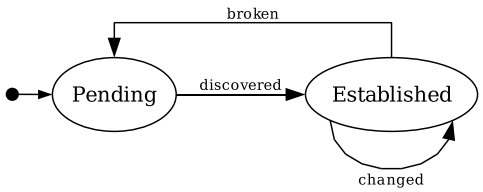

Abbildung 4.7: Rio: Mögliche Zustände einer Assoziation

Implementiert eine JSB die Schnittstelle `AssociationListener`, kann der Association-Manager die Dienstobjekte der JSB über Zustandsänderungen der Beziehungen informieren, sonst müssen Dienstobjekte der JSB selbst die Zustände der Beziehungen ihrer JSB abfragen. Eine Beziehung wird als Objekt der Klasse `Association` repräsentiert. Damit über dieses Objekt Zustandsänderungen abgefragt werden können, werden dort zusätzliche Zustände benutzt, die anzeigen, dass eine Zustandsänderung stattgefunden hat: `broken`, `discovered` und `changed`.

Die Methoden dieser Schnittstelle besitzen alle zwei Parameter: `association` und `service`. Über den Parameter `association`, wird die `Association`, deren Zustand sich verändert hat, und über den Parameter `service` wird ein Proxy-Objekt, mit dem auf ein Dienstobjekt der JSB zugegriffen werden kann, übergeben.

Ein Dienstobjekt einer JSB kann nicht nur über Zustandsänderungen einer Beziehung informiert werden und Proxy-Objekte abspeichern, um das Dienstobjekt, mit dem es verbunden wurde, zu benutzen, es kann auch direkt nach verfügbaren Dienstobjekten einer Beziehung suchen.

Wenn mehrere Dienstobjekte eine JSB verwenden würden, um mehrere Kameras desselben Typs zu repräsentieren, würden bei der Verwendung von Beziehungen folgende Probleme entstehen:

- Sobald eine Beziehung im Zustand `Discovered` ist, werden keine Ereignisse mehr ausgelöst. Wird eine neue Kamera hinzugefügt, dann erhält die `Security`-JSB nur dann eine Benachrichtigung, wenn dies die erste Kamera ist.

- Da Rio standardmäßig Round-Robin verwendet, um ein Dienstobjekt auszuwählen, müsste ein Komparator geschrieben werden, um das jeweils gewünschte Dienstobjekt zu erhalten, da sonst ein beliebiges gewählt werden kann.

Deshalb sollte die Möglichkeit mehrere Dienstobjekte von einer JSB zu erzeugen nur zum Lastausgleich benutzt werden. Dies ist im eHome-Bereich nur in wenigen Fällen nötig und sinnvoll.

In der Arbeit wurde deshalb für jede Kamera und jeden Bewegungsmelder eine eigene JSB definiert. Dann muss auch eine Beziehung von der Security-JSB zu jeder Kamera und zu jedem Bewegungsmelder definiert werden, die benutzt werden. Eine andere Möglichkeit bestünde darin, nach Dienstobjekten mit Jini-Methoden zu suchen.

Rio benutzt das von Jini angebotene Persistenzkonzept, um Dienstobjekte persistent zu speichern, und dadurch den Zustand eines Dienstobjekts zu sichern. Jini kann Dienstobjekte in einem speziellen Speicherbereich sichern und zu einem späteren Zeitpunkt

wiederherstellen. Wird Jini beendet und anschließend neu gestartet, werden alle abgespeicherten Dienstobjekte wiederhergestellt. Auch während Jini ausgeführt wird, können Dienstobjekte gespeichert und wiederhergestellt werden.

Fehlererkennung (eng: *Faultdetection*) in Rio ist ein Konzept, das es erlaubt, nicht mehr erreichbare Dienstobjekt zu erkennen. Das Rahmenwerk startet bei der Installation einer JSB einen Provisioner. Der Provisioner sorgt dafür, dass genügend Dienstobjekte einer JSB erreichbar sind, um die Last zu bewältigen. Der Provisioner startet neue Dienstobjekte, wenn ein Dienstobjekt nicht mehr erreichbar ist.

Mit dem Faultdetection-Konzept bietet Rio eine Möglichkeit die Funktionsfähigkeit von Dienstobjekten zu überwachen. Bei Dienstobjekten, die Geräte repräsentieren, ist ein anderer Aspekt viel wichtiger. Hier sollte überprüft werden, ob das Gerät funktionsbereit ist. Zum Beispiel könnte die Funktion einer Treppenbeleuchtung in regelmäßigen Abständen überprüft werden, indem die Beleuchtung kurz eingeschaltet wird und mithilfe eines Helligkeitssensors überprüft wird, ob die Beleuchtung funktioniert. Faultdetection ist dann ein weiterer Dienst, der im Haus installiert werden kann. Dienstobjekte, die nicht mehr erreichbar sind, einfach neu zu starten bringt im eHome-Bereich meistens nichts. Denn wenn ein Gerät nicht funktioniert, bringt ein Neustart seiner Repräsentation nichts. Welches Verhalten beim Ausfall eines Geräts sinnvoll ist, kann sehr unterschiedlich sein.

4.1.2 Implementierung

In Abbildung 4.8 sind die für den vereinfachten Sicherheits-Dienst in Rio implementierten Klassen dargestellt. In den Klassen, deren Name mit UI endet sind einfache graphische Benutzeroberflächen implementiert, die zum Austesten der Implementierung benutzt werden.

In dieser Arbeit wurden die Treiber als JSBs entwickelt. Sie steuern Geräte, die an einem normalen PC über USB bzw. X10 angeschlossen sind. Ein X10-Bewegungsmelder wurde durch eine JSB, die die Klassen und Schnittstellen aus dem Paket `ehome.scenario.rio.sensors.movement` verwendet, repräsentiert. Es wird allerdings auf die Abstraktion von allgemeinen X10-Aktoren verzichtet, da Rio keine Klassen zum Zugriff auf X10 bietet.

In dem hier realisierten vereinfachten Sicherheits-Dienst müssen zwei Camera-JSBs benutzt werden. Der zugehörige Operational-String ist in Listing 4.1 abgebildet. Die beiden JSBs sind in diesem beschreiben. Sie unterscheiden sich nur in unterschiedliche Parametrisierungen. Da beide `Camera`-JSBs denselben Code verwenden, entsteht kein zusätzlicher Implementierungsaufwand. Dasselbe gilt für beide `Motion`-JSBs.

Die in dieser Arbeit realisierten JSBs `Camera1` und `Camera2` werden durch die Klasse `CameraImpl` bzw. deren Unterklasse `Webcam` implementiert. In `CameraImpl` wurde die Implementierung der Methoden der Schnittstelle `ServiceBean` erweitert. Außerdem wurden in dieser Klasse Methoden implementiert, die reale Kameras meistens nicht anbieten können. Zum Beispiel die Methode `getLocation`: Eine Kamera weiß selbst nicht, in welchem Raum sie sich befindet. In `Webcam` befindet sich die Implementierung der Methode `getImage`. In Listing 4.8 ist die Implementierung der Methoden `initialize` und `destroy` in der Klasse `CameraImpl` dargestellt. In `CameraImpl` wird der Parameter `Location` bei der Initialisierung von Dienstobjekten ausgelesen

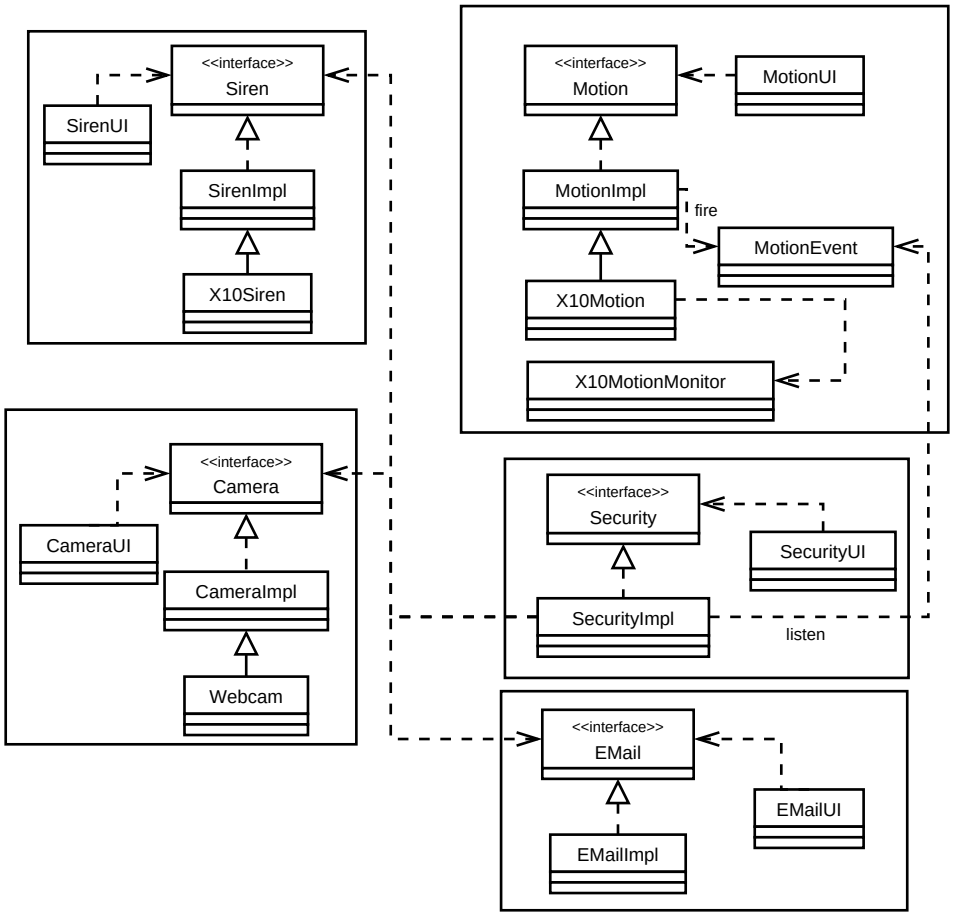

Abbildung 4.8: Klassenübersicht des vereinfachten Sicherheits-Dienstes in Rio

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE opstring SYSTEM "java://com/sun/rio/dtd/rio_opstring.dtd"
    [<!ENTITY Local.IP SYSTEM
         "java://java.net.InetAddress.getLocalHost().
         getHostAddress()" >
    <!ENTITY Local.Port "9000" >
    <!ENTITY CodeServerURL "http://&Local.IP;:&Local.Port;/" >]>
<opstring>
    <OperationalString Name="Camera">

        <ServiceBean Name="Camera1" MatchOnName="yes">
            <Codebase> &CodeServerURL; </Codebase>
            <Interfaces>
                <Interface>
                    ehome.scenario.rio.devices.camera.Camera
                </Interface>
                <Resources>
                    <JAR>camera-dl.jar</JAR> <JAR>rio-dl.jar</JAR>
                </Resources>
            </Interfaces>
            <Component> ehome.scenario.rio.devices.camera.Webcam
                <Resources>
                    <JAR>camera.jar</JAR> <JAR>rio.jar</JAR>
                </Resources>
            </Component>
            <Groups> <Group>rio</Group> </Groups>
            <Associations>
                <Association Type="requires" Name="Lookup"
                    OperationalString="Infrastructure Services"/>
                </Association>
            </Associations>

            <Parameters>
                <Parameter Name="Location" Value="Wohnzimmer" />
                <Parameter Name="URL"
                    Value="http://myhouse.ulno.net/cam1.jpg" />
            </Parameters>

            <Maintain>1</Maintain>
            <Comment>Security-Scenario Camera</Comment>
        </ServiceBean>
        <ServiceBean Name="Camera2" MatchOnName="yes">
                ...
            <Parameters>
                <Parameter Name="Location" Value="Küche" />
                <Parameter Name="URL"
                    Value="http://myhouse.ulno.net/cam1.jpg" />
            </Parameters>
                ...
        </ServiceBean>

        <Include>infrastructure.xml</Include>
    </OperationalString>
</opstring>
```

Listing 4.1: Operational-String Camera

```
1  public class CameraImpl extends ServiceBeanAdapter implements Camera
2  {
3    public void initialize(ServiceBeanContext context) throws Exception
4    {
5      super.initialize(context);
6      try
7      {
8        UIDescriptor uiDesc = UIDescriptorFactory.
9          getUIDescriptor(MainUI.ROLE,
10           new UIComponentFactory(
11             new URL[] { new URL(context.getExportCodebase()
12               + "camera-ui.jar")},
13           "ehome.scenario.rio" + ".devices.camera.CameraUI") );
14       addAttribute(uiDesc);
15       determineParameters();
16     } catch (Exception error) { error.printStackTrace(); }
17   }
18
19   //Auslesen der Parameter
20   private void determineParameters()
21   {
22     Object obj = context.getInitParameter("Location");
23     if (obj!=null) location = (String) obj;
24     else location = "undefined";
25     //Wird in der Klasse Webcam erweitert.
26   }
27   public void destroy() { super.destroy(); }
28 }
```

Listing 4.2: Initialisierung von Camera-Dienstobjekten

(Listing 4.3, Zeile 25-33). Das Problem ist, dass die Parameter während der Laufzeit nur durch die Re-Installation des Operational-Strings verändert werden können.

Die JSBs `Camera1` und `Camera2` bieten die synchrone Dienstschnittstelle `Camera` (Listing 4.4) an. In dieser Schnittstelle sind die Methoden: `getImage`, `getLocation`, `setLocation` und `test` definiert.

Die JSBs `Motion1` und `Motion2` erzeugen Events von der Event-Klasse `MotionEvent` (Listing 4.5), die von der JSB `Security` empfangen werden können. `MotionEvent`-Objekte enthalten folgende Informationen:

- Eine Zeitangabe, die angibt, wann der Bewegungsmelder ausgelöst wurde.
- Der Name der JSB, die den Bewegungsmelder repräsentiert.
- Den Ort, an dem der Bewegungsmelder sich befindet.

Zur Erzeugung von Event-Objekten muss die JSB beim Rahmenwerk als Event-Erzeuger angemeldet werden. Die Methode `initialize` der Klasse `MotionImpl` wurde dazu um den in Listing 4.6 abgebildeten Code erweitert.

Um `MotionEvents` zu erzeugen und auszulösen, wurde eine zusätzliche Methode `fireEvent` (Listing 4.7) in der Klasse `MotionImpl` implementiert. Diese Methode erzeugt ein `MotionEvent` und initialisiert es.

In dieser Arbeit wurde zu Demonstrationszwecken sowohl innerhalb der Event-Klasse `MotionEvent` als auch in der Methode `fireEvent` das Event-Objekt initialisiert. Die Initialisierung muss natürlich nur an einer der beiden Stellen vorgenommen werden.

```
1  private DynamicEventConsumer consumer;
2
3  // Anmeldung
4  public void initialize(ServiceBeanContext context)
5      throws Exception {
       :
6      :
7      DiscoveryManagement dMgr = getDiscoveryManager();
8      EventDescriptor eventTemplate =
9          new EventDescriptor(null, new Long(MotionEvent.ID));
10     consumer = new DynamicEventConsumer(eventTemplate, this, dMgr);
       :
11     :
12 }
13
14 // Methode zum Empfangen von Events
15 public void notify(RemoteServiceEvent event) {
16     if (!activated)
17         return; // The Security-Service isn't active
18
19     if (event == null) // Rahmenwerks-Fehler
20         return; // There happened something stupid
21
22     if (event instanceof MotionEvent) {
23         giveAlarm((MotionEvent) event);
24     } else {
25         System.out.println("Unwanted event received: " + event);
26     }
27 }
28
29 private void giveAlarm(MotionEvent event) {
30
31     // Auslesen der Eventinformationen
32     String sender = event.getWho();
33     String location = event.getLocation();
34     String timestamp = new Date(event.getWhen()).toString();
35
36     giveAlarm(sender, location, timestamp);
37 }
```

Listing 4.3: Event-Listener Security

```
1  {
2      public ImageIcon getImage() throws RemoteException;
3      public String getLocation() throws RemoteException;
4      public void setLocation(String loc) throws RemoteException;
5      public String test() throws RemoteException;
6  }
```

Listing 4.4: Synchrone Dienstschnittstelle `Camera`

```
1  public class MotionEvent extends RemoteServiceEvent {
2      public static final long ID = 9999999993L; //unique id
3
4      /** hold the properties */
5      private long when;
6      private String who = null;
7      private String location = null;
8
9      public MotionEvent(Object source) {
10         super(source);
11         setWhen(System.currentTimeMillis());
12         if (source instanceof Motion){
13             Motion m = (Motion)source;
14             try {
15                 setWho(m.getLocation());
16                 setLocation(m.getLocation());
17             } catch (RemoteException e) {
18                 e.printStackTrace();
19             }
20         }
21     }
22
23     /** Getter und Setter */
24     public long getWhen() { return when; }
25     public void setWhen(long value) {    when = value; }
26     public String getWho() { return who; }
27     public void setWho(String value) { who = value; }
28     public String getLocation() {   return location; }
29     public void setLocation(String value) { location = value;   }
30 }
```

Listing 4.5: Implementierung des Event-Typs `MotionEvent`

```
1  private EventHandler eventHandler;
2  private EventDescriptor eventDescriptor;
3
4  public void initialize(ServiceBeanContext context)
5          throws Exception {
               .
               .
6      eventDescriptor = new EventDescriptor(MotionEvent.class,
7          new Long(MotionEvent.ID));
8      eventHandler = new DispatchEventHandler(eventDescriptor);
9
10     eventTable.put(new Long(MotionEvent.ID), eventHandler);
11     addAttribute(eventDescriptor);
               .
               .
12 }
```

Listing 4.6: Anmeldung als Event-Erzeuger

```
1  public void fireEvent() throws RemoteException{
2      try {
3          MotionEvent event = new MotionEvent(this);
4          //Add event information
5          event.setLocation(this.location);
6          event.setWho(this.serviceName);
7          //fire
8          eventHandler.fire(event);
9          invocationCount++;
10     } catch (NoEventConsumerException e) {
11         //Than it would be useless to send an Event
12         //Or somebody else should be notified
13     }catch (RemoteException e){
14         e.printStackTrace();
15     }
16 }
```

Listing 4.7: Erzeugen und Auslösen eines MotionEvents

Die Klasse `X10Motion` ist eine Verfeinerung der Implementierung der Klasse `MotionImpl`. In ihr wird ein Thread erzeugt, der auf der X10-Schnittstelle lauscht, um Ereignisse von X10-Bewegungsmeldern zu empfangen. Meldet der X10-Bewegungsmelder eine Bewegung, führt der Thread die Methode `fireEvent` aus.

Damit die JSB `Security` Events vom Typ `MotionEvent` empfangen kann, wird die Anmeldung als Listener bei der Initialisierung des Dienstobjekts durchgeführt (Listing 4.3). Wird ein Event empfangen, wird zunächst überprüft, ob der Security-Dienst aktiv ist. Da Rio manchmal auch einfach `null` als Parameter übergibt, muss dieser Fehler abgefangen werden. Im Anschluss wird überprüft, ob das Event ein `MotionEvent` ist. Zwar besitzt jede Event-Klasse eine ID, aber es kann auch passieren, dass zwei Event-Klassen dieselbe ID besitzen.

Ist das Event ein `MotionEvent`, wird die Methode `giveAlarm` ausgeführt. In dieser Methode werden die Event-Informationen aus dem `MotionEvent` extrahiert und die eigentliche Alarmmethode ausgeführt.

Die Bibliothek für die Darstellung der Benutzeroberfläche wird bei der Anmeldung mit angegeben (Listing 4.8, Zeile 9-16).

Folgende Bibliotheken wurden für die JSBs `Camera1` und `Camera2` erzeugt:

`camera.jar`: `CameraImpl, Webcam, Webcam_Stub`

`camera-dl.jar`: `Camera, Webcam_Stub`

`camera-ui.jar`: `CameraUI`

Die Klasse `Webcam_Stub` wurde automatisch generiert. Von ihr werden zur Laufzeit Proxy-Objekte erzeugt. Die für die anderen JSBs verwendeten Archive sind ähnlich aufgebaut.

Die JSB `Security` besitzt Beziehungen mit den JSBs `Kamera1`, `Kamera2`, `EMail` und `Siren`. Diese Beziehungen sind vom Typ `requires`. Das Dienstobjekt der JSB `Security` wird über Zustandsänderungen der Beziehungen benachrichtigt. Dazu implementiert die Klasse `SecurityImpl` die Schnittstelle `AssociationListener` (Listing 4.9).

```
1  public class CameraImpl extends ServiceBeanAdapter
2          implements Camera {
3
4      public void initialize(ServiceBeanContext context)
5              throws Exception {
6
7          super.initialize(context);
8          try {
9              UIDescriptor uiDesc = UIDescriptorFactory.
10             getUIDescriptor(MainUI.ROLE,
11                 new UIComponentFactory(new URL[] {
12                     new URL(context.getExportCodebase() +
13                         "camera-ui.jar")}, "ehome.scenario.rio"
14                             +    ".devices.camera.CameraUI"));
15
16             addAttribute(uiDesc);
17
18             determineParameters();
19
20         } catch (Exception error) {
21             error.printStackTrace();
22         }
23     }
24
25     //Auslesen der Parameter
26     private void determineParameters() {
27         Object obj = context.getInitParameter("Location");
28         if (obj!=null)
29             location = (String) obj;
30         else
31             location = "undefined";
32
33         //Wird in der Klasse Webcam erweitert.
34     }
35
36     public void destroy() {
37         super.destroy();
38     }
39 }
```

Listing 4.8: Initialisierung von `Camera`-Dienstobjekten

```
1  //Anmeldung als AssociationListener
2
3  public void initialize(ServiceBeanContext context)
4          throws Exception {
5      super.initialize(context);
6
7      try {
8              ...
9          //Registriere Listener
10         getAssociationManagement().register(this);
11
12         //Objekte um Kameras zu benutzen
13         cameras = new Hashtable();
14             ...
15     } catch (Exception error) {
16         error.printStackTrace();
17     }
18 }
19
20 //Methode der Schnittstelle Associationlistener.
21
22 public void discovered(Association ass, Object service) {
23
24     if (service == null) return;
25
26     if (service instanceof Camera){
27         Camera cam = (Camera) service;
28         cameras.put(ass.getName(), cam);
29         ...
30     }
31     else if (service instanceof EMail){ ... }
32     else if (service instanceof Siren){ ... }
33 }
34
35 public void changed(Association ass, Object service) {
36     discovered(ass, service);
37 }
38
39 public void broken(Association ass, Object service) {
40
41     if (service == null) return;
42
43     if (service instanceof Camera){
44         Camera cam = (Camera) service;
45         cameras.remove(ass.getName());
46     }
47     else if (service instanceof EMail){
48         mailer = null;
49     }
50     else if (service instanceof Siren){
51         siren = null;
52     }
53 }
```

Listing 4.9: Implementierung der JSB Security als AssociationListener

Datei (-typ)	Codezeilen
*.java-Dateien	3891
davon Gerätetreiber	3226
davon Erweiterungs-Dienste	665
Operational-Strings (*.xml)	540
build.xml	375
build.properties	6
Summe	4812

Tabelle 4.1: Geschriebene Quellcodezeilen für den vereinfachten Sicherheitsdienst in Rio.

4.1.3 Zusammenfassung

Zur synchronen Kommunikation mit Dienstobjekten werden in Rio Java-Schnittstellen benutzt, die die Funktionen eines Dienstobjekts beschreiben. Um Nachrichten zu versenden, besitzt Rio ein eigenes Event-Modell. Dieses Event-Modell wurde von Jini übernommen. Ein Nachteil des Modells ist, dass zwar Event-Klassen definiert werden können, aber IDs nötig sind, um Events zu filtern, obwohl eine Nutzung des Typ-Modells viel einfacher und sinnvoller wäre.

Jede JSB wird in einem Operational-String beschrieben. Es gibt keine eigentliche Installation für Komponenten. Das Rahmenwerk lädt die Bestandteile einer JSB, wenn diese benötigt werden. Neue JSBs mit gleicher Funktionalität lassen sich sehr leicht durch Kopieren des Beschreibungsteils im Operational-String erstellen. Die Operational-Strings bieten eine gute Grundlage für eine Deployment-Beschreibung.

Damit ein Dienstobjekt ein anderes über eine synchrone Dienstschnittstelle mit den von Rio angebotenen Methoden benutzen kann, muss zwischen den JSBs, von denen die Dienstobjekte erzeugt wurden, eine Beziehung bestehen. Um Beziehungen sinnvoll zu benutzen, darf im eHome-Bereich meistens nur ein Dienstobjekt jeder JSB erzeugt werden. Es ist nur dann sinnvoll, mehrere Dienstobjekte einer JSB zu erzeugen, wenn es nicht wichtig ist, welches Dienstobjekt einer JSB benutzt wird.

Ein weiterer Nachteil ist, dass Beziehungen nicht zur Laufzeit hinzugefügt werden können, dadurch ist dieses Konzept ungeeignet, Beziehungen zwischen Dienstobjekten herzustellen. Zur Kommunikation zwischen Dienstobjekten wird auf Jini zurückgegriffen.

In Rio wird die Anmeldung von Dienstobjekten durch ein Programmier-Modell erleichtert. Eine JSB wird durch eine Klasse repräsentiert, die von der Klasse `ServiceBeanAdapter` abgeleitet wird. Dadurch muss der Komponenten-Entwickler nur noch die Methoden der Schnittstelle `ServiceBean` erweitern, um Ressourcen, die das Dienstobjekt benötigt, zu binden und wieder freizugeben. Darüber hinaus ist die Anmeldung einer Klasse als Eventlistener mit einem hohen programmiertechnischen Aufwand verbunden.

Tabelle 4.1 zeigt den Entwicklungsaufwand des vereinfachten Sicherheitsdienstes in geschriebenen Codezeilen für Rio. Zu beachten ist, dass Zeile zwei und drei nur eine Aufteilung der Dateien aus Zeile 1 darstellen.

Die Nachteile von Rio im eHome-Bereich sind zu groß, um es als Rahmenwerk für eHome-Systeme einzusetzen. In Bereichen, in denen Dienstobjekte parallel dieselbe Auf-

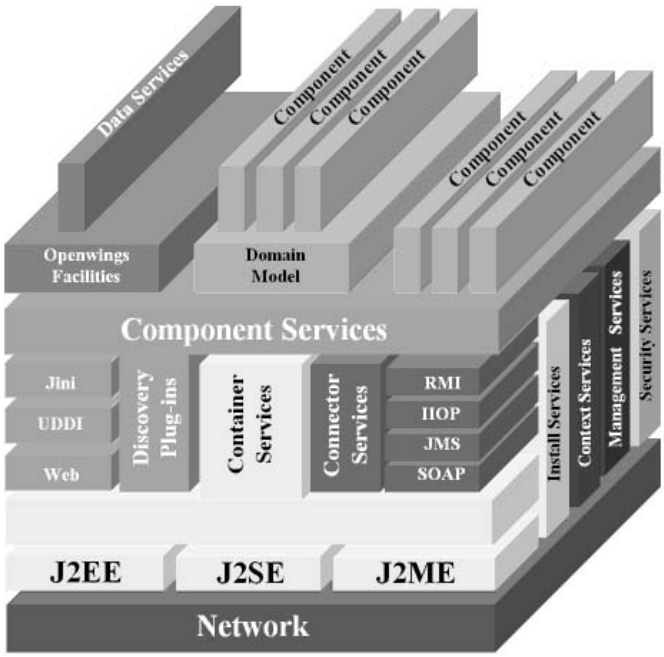

Abbildung 4.9: Architektur des Openwings-Rahmenwerks (aus [Bie02])

gabe mit anderen Eingabeparametern ausführen müssen und ein Lastausgleich sowie eine dynamische Ausnutzung von Rechner-Ressourcen wichtig sind, ist Rio eine gute Wahl. Die Implementierung eines plattformspezifischen Deployers für Rio wäre jedoch mit zu hohem Aufwand verbunden.

4.2 Openwings

4.2.1 Beschreibung

Motorola (General Dynamics Decision Systems) und Sun Microsystems haben im Juni 1999 das *Openwings Konsortium* [Gen] als offene Community gegründet. Das Ziel der Community ist es, ein Komponenten-Rahmenwerk zu spezifizieren, das unabhängig von verwendeter Middleware, Datenbanken, Rechnerarchitekturen und Betriebssystemen ist. Bis heute haben sich über 100 Unternehmen dieser Community angeschlossen [CB03b]. Als Anwendungsgebiete sind ubiquitäre Netzwerke, wie auch eHome-Systeme, aber auch militärische Anwendungen angegeben, in denen Netzwerke eine sehr dynamische Struktur aufweisen. Neben der Spezifikation wird auch eine Referenzimplementierung entwickelt. Diese wird in dieser Arbeit verwendet.

In Abbildung 4.9 ist die Architektur des Rahmenwerks dargestellt:

1. Auf der untersten Ebene befindet sich das Netzwerk. Dieses stellt eine Menge von Rechnern und Geräten dar, die mit einem Component Service kommunizieren kön-

nen. Dienstobjekte, die auf unterschiedlichen Plattformen ausgeführt werden, können mithilfe von *Konnektoren* miteinander kommunizieren. Für das Rahmenwerk ist es nicht von Bedeutung, auf welchem Rechner ein Dienstobjekt ausgeführt wird. Der *Openwings Container* (*Container Service*) stellt selbst einen Dienst dar, der über das Netzwerk aufgerufen werden kann.

2. Auf der nächst höheren Schicht sind die auf einem Betriebssystem laufenden JVMs verschiedener Java Editionen dargestellt. Welche JVM benutzt werden kann, und damit auch, welche Rechnerarchitekturen und Betriebssysteme benutzt werden können, hängt von den verwendeten Discovery Plugins und Connector Services ab. Die Referenzimplementierung benötigt eine vollwertige Java 2 Standard Edition JVM, da sie Jini als Discovery Plug-In, RMI [Sun05] für synchrone Konnektoren und JMS [HBS+02] für asynchrone Konnektoren verwendet. Inwieweit Java wirklich plattformunabhängig ist, wurde in Abschnitt 4.1 schon erläutert. Die Dienste des Rahmenwerks müssen nicht in Java implementiert werden, aber Java-Schnittstellen anbieten. In der Referenzimplementierung sind die Dienste selbst auch in Java realisiert.

3. Auf der darüber liegenden Schicht befinden sich die von Openwings angebotenen Rahmenwerks-Dienste. Diese können auch von außerhalb des Rahmenwerks ausgeführter Software benutzt werden. Sowohl diese Software als auch die von einem Container Service (s.u.) ausgeführten Prozesse, können den Component Service verwenden, um Dienstobjekte zu erzeugen und zu verwenden. Außerdem können sie über einen Component Service alle Dienste des Rahmenwerks auffinden und nutzen. Deshalb liegt der Component Service über allen anderen Diensten. Folgende Dienste werden angeboten:

   Container Service: Der *Container Service* bietet Zugriff auf verschiedene Discovery Plugins und Connector Services und eine Laufzeitumgebung für ausführbare Komponenten. Mittels der Discovery Plugins können Dienstobjekte aufgefunden und eine Kommunikation mit diesen mittels des Connector Service (s.u.) aufgebaut werden. Die Spezifikation sieht die Verwendung von Jini (siehe [Sunc]), UDDI und Webservices als Discovery Services vor. Für die Implementierung von Connector Services sind RMI, IIOP, JMS und SOAP vorgesehen.

   Connector Service: Ein *Connector Service* abstrahiert von der Verwendung einer Verbindungstechnik und kann somit Verbindungen über verschiedene Verbindungstechniken anbieten. Er generiert automatische Konnektoren für verschiedenen Dienstschnittstellen und Verbindungstechniken und die für die Verbindung notwendigen Proxy-Objekten (siehe [BC03a]).

   Component Service: Der *Component Service* unterstützt die Erzeugung von Dienstobjekten in einem Container, um sie dort zur Verfügung zu stellen, die An- und Abmeldung von Dienstschnittstellen von Dienstobjekten und die Erzeugung zugehöriger Konnektoren, die die Kommunikation mit einem Dienstobjekt ermöglichen.

Install Service: Der *Install Service* ist für die Installation von Komponenten verantwortlich. Eine Komponente muss über einen Installable Component Descriptor (ICD, s.u.) verfügen, damit sie installiert werden kann.

Context Service: Openwings erlaubt es, mehrere Plattformen, auf denen Openwings ausgeführt wird, zu einem Kontext zu verbinden. Jeder Kontext wird von einem *Context Service* verwaltet.

Management Service: Bietet eine Anbindung von Openwings an verschiedene Software und Konzepte, die Managementaufgaben unterstützen: zum Beispiel CIM, DMTF, FMA [SBC03] Der Management Service wird in dieser Arbeit nicht verwendet.

Security Service: Dieser bietet Code-, Transport- und Dienstsicherheitskonzepte. Für die Codesicherheit wird genauso, wie in Rio, das Java Code Security Concept [Sun98] verwendet. Die Transportsicherheit bezieht sich auf die Verschlüsselung der Kommunikation zwischen Dienstobjekten unter Verwendung von Konnektoren. Die Dienstsicherheit wird über ein Rollenkonzept und Zugriffsrechte geregelt. [BT03]

4. Auf der obersten Ebene der Architektur sind die im Rahmenwerk installierten *Komponenten* dargestellt. Diese werden in die drei Gruppen unterteilt: Openwings Facilities, Domain Model und sonstige Komponenten. Komponenten des Domain Modells sind in dieser Arbeit Komponenten, die von allen eHome-Diensten benutzt werden, um ihre Funktion zu erbringen.

   Openwings Facilities sind Komponenten, die grundlegende Dienste bereitstellen, die nicht Bestandteil der Spezifikation sind, wie zum Beispiel auch der *Data Service*, welcher der Anbindung an Datenbanken dient.

   Alle anderen Komponenten, die nicht zu einer dieser beiden Gruppen gehören, fallen unter sonstige Komponenten; zum Beispiel auch die in dieser Arbeit entwickelte Komponente `SecurityProvider_im`.

Openwings verfügt über zwei *Werkzeuge*, mit denen Komponenten und Prozesse, die innerhalb des Rahmenwerks ausgeführt werden, verwaltet werden können. Der *Openwings Explorer* (Abbildung 4.10) ist eine grafische Benutzeroberfläche und die *Openwings Shell* (Abbildung 4.11) ist das dazu passende Kommandozeilen-Werkzeug.[2]

Komponenten werden in Openwings nicht wie in Rio über einen Http-Server bereitgestellt, sondern in jar-Archiven verpackt zur Installation durch das Rahmenwerk angeboten. Für eine mögliche Dateistruktur dieser siehe das Openwings Tutorium [Ope03b].

Schnittstellen werden in eigenen nicht ausführbaren Komponenten bereitgestellt. Im Gegensatz zu Rio ist es in Openwings sinnvoll möglich, mehrere Dienstobjekte von einer Komponente zu erzeugen.

In Openwings gibt es zwei Arten von Komponenten:

Nicht ausführbare Komponenten: Eine nicht ausführbare Komponente, enthält Dateien, die von anderen Komponenten benutzt werden können. Dazu gehören Dienstschnittstellen.

[2] Das Openwings Tutorium [Ope03b] beschreibt den Umgang mit diesen beiden Werkzeugen.

Abbildung 4.10: Openwings Explorer

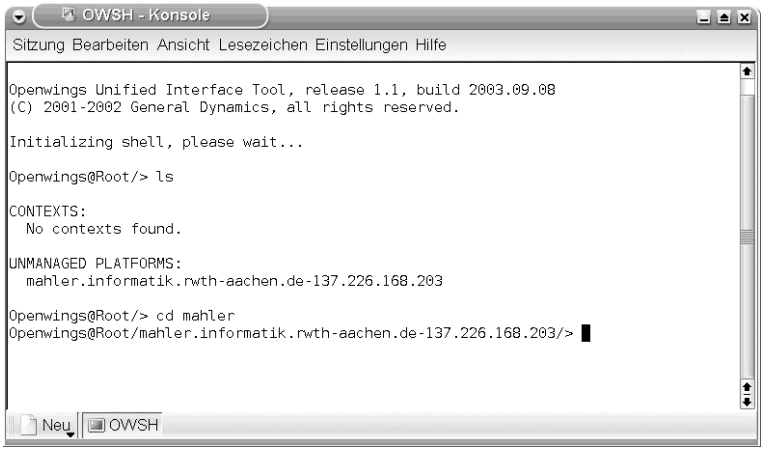

Abbildung 4.11: Openwings Shell

Ausführbare Komponenten: Eine Komponente, die ausführbare Dateien enthält, wird als ausführbar bezeichnet. Eine ausführbare Datei kann in Openwings eine ausführbare Java-Klasse oder eine Skript-Datei, die vom jeweiligen Betriebssystem ausgeführt werden kann, sein. Der Container Service startet einen Prozess, um eine Komponente auszuführen, und verwaltet diesen Prozess .

Mithilfe ausführbarer Komponenten können Dienstobjekte erzeugt werden, die Dienstschnittstellen anmelden können. Ausführbare Komponenten werden nach der Art der Dienstschnittstellen, die von ihnen erzeugte Dienstobjekte anbieten oder nutzen, unterteilt in:

Provider: bietet synchrone Dienstschnittstellen an.

User: nutzt synchrone Dienstschnittstellen.

Publisher: erzeugt Events mit einer asynchronen Dienstschnittstelle.

Subscriber: wird mithilfe asynchroner Dienstschnittstellen über Events benachrichtigt.

Diese Bezeichnungen werden an den Namen der Komponente angehängt.

Komponenten-Entwickler können in Openwings wie in Rio *Schnittstellen* in Java spezifizieren. In der Interface Definition Specification [BNC03] werden folgende Typen von Schnittstellen spezifiziert:

Dienstschnittstellen: Dienstobjekte bieten *Dienstschnittstellen*[3] an, damit ihre Dienste benutzt werden können. In Openwings gibt es nicht nur wie in Rio synchrone sondern auch asynchrone Dienstschnittstellen:

- In einer *synchronen Dienstschnittstelle* werden Methoden spezifiziert, die benutzt werden können, um ein Dienstobjekt zu verwenden.

  Als Dienstschnittstelle dürfen prinzipiell beliebige Java-Schnittstellen verwendet werden. Openwings verfügt über keine allgemeine Schnittstelle, von der alle Dienstschnittstellen abgeleitet werden. Allerdings stellt auch Openwings weitere Anforderungen an die in einer synchronen Dienstschnittstelle spezifizierten Methoden. Die Bedingungen sind dieselben wie in Rio. Eine synchrone Dienstschnittstelle muss die folgenden Bedingungen erfüllen:

  – Alle Methoden der Schnittstelle müssen Ausnahmen vom Typ `RemoteException` auslösen können, da bei der Verwendung von Dienstobjekten innerhalb eines Netzwerks jederzeit Fehler auftreten können. Dadurch ist der Komponenten-Entwickler wie in Rio gezwungen, über Probleme der verteilten Programmierung nachzudenken.
  – Jedes als Parameter verwendete Objekt muss serialisierbar sein. Dasselbe gilt für Rückgabeobjekte. Bei serialisierbaren Objekten kann es sich auch um Remote-Referenzen handeln [Edw00].

[3] In der Sprechweise von Openwings werden angebotene Dienstschnittstellen als Dienst bezeichnet. Da der Begriff Dienst in allen drei Rahmenwerken unterschiedlich belegt ist, soll er hier nicht in diesem Sinn verwendet werden.

- Openwings erlaubt den Komponenten-Entwicklern die Spezifikation beliebiger *asynchroner Dienstschnittstellen*, die zur asynchronen Kommunikation zwischen Dienstobjekten verwendet werden können.

  Ein Event-Erzeuger benutzt eine asynchrone Dienstschnittstelle, um Ereignisse zu erzeugen. Ein Event-Listener implementiert eine asynchrone Dienstschnittstelle, um auf Ereignisse, die mit dieser Schnittstelle erzeugt werden, zu reagieren.

  Bei der Spezifikation asynchroner Dienstschnittstellen gelten dieselben Regeln wie bei synchronen Dienstschnittstellen. Es gibt allerdings keine Rückgabewerte.

Legacy System Schnittstellen: Diese Art von Schnittstellen wird von Konnektoren verwendet, um Dienste von Legacy Systemen verfügbar zu machen.

Data Object Schnittstellen: Mit einer Data Object Schnittstelle kann auf Objekte zugegriffen werden, die in einer Datenbank gespeichert sind. Da die Entwicklung des Data Services zunächst eingestellt wurde, findet diese Art von Schnittstellen zur Zeit keine Verwendung. In OSGi bietet der Configuration Admin die Möglichkeit Konfigurationsdaten zu speichern.

Attribut Schnittstellen: Mithilfe von Attribut-Schnittstellen können Parameter gesetzt werden, die bei der Suche nach Dienstobjekten benutzt werden können, um bestimmte Dienstobjekte aufzufinden.

Policy Schnittstellen: Konfigurationsdaten werden in Openwings in Policy-Dateien abgelegt. Sie werden zur Speicherung und Bereitstellung von Konfigurationsdaten benutzt. Jede *Policy* besitzt eine Policy-Schnittstelle, mit der Konfigurationsdaten aus dieser Datei abgefragt und geändert werden können.

Jede Policy-Schnittstelle definiert einen Policy-Typ. In Openwings sind folgende definiert:

InstallableComponentDescriptorPolicy: In Policies dieses Typs werden Konfigurationsdaten zur Installation von Komponenten abgelegt.

ProvideServicePolicy, UseServicePolicy, PublishServicePolicy, EventServicePolicy: Dies sind Policies, die verwendet werden können, um angemeldete Dienstschnittstellen zu beschreiben.

Management Policies: Diese können Konfigurationsdaten für Management Services bereitstellen. Da Management Services in dieser Arbeit nicht untersucht wurden, wird dieser Policy-Typ nicht weiter betrachtet.

Um Policy-Schnittstellen zum Lesen und Schreiben von Policy-Daten zu benutzen, müssen eine Reihe von Klassen erzeugt und implementiert werden.

In Openwings müssen Schnittstellen, die von anderen Diensten benutzt werden sollen, in nicht ausführbaren Komponenten angeboten werden. Dies liegt daran, dass in Openwings nur Beziehungen zwischen ausführbaren und nicht ausführbaren Komponenten geknüpft

werden können. Für Schnittstellen bedeutet dies, dass Schnittstellen nicht in ausführbaren Komponenten enthalten sein dürfen, da sie von mehreren Komponenten verwendet werden. Da Openwings über ein Versionskonzept für Komponenten verfügt, ist eine Versionierung von Schnittstellen möglich.

Jede Software kann mithilfe der Dienste des Rahmenwerks Dienstobjekte erzeugen und anmelden. Openwings kann Komponenten nach der Installation auch automatisch starten, um Dienstobjekte zu erzeugen. Besitzt eine Komponente mehrere Installable Component Descriptoren, wird die Komponente nicht mehrfach installiert, aber mehrfach gestartet. Dadurch können auch mehrere Dienstobjekte mit unterschiedlichen Konfigurationen erzeugt werden. Das Problem dieser Methode ist, dass nicht nachträglich Konfigurationsdaten hinzugefügt werden können, um weitere Dienstobjekte zu erzeugen. Eine Möglichkeit zur Abhilfe bestünde darin, die Komponente einfach erneut zu installieren. Doch würden dann Dienstobjekte mehrfach erzeugt. Um dies zu unterbinden, müsste in der Implementierung der `main`-Methode zunächst überprüft werden, ob das Dienstobjekt schon existiert.

Der Component Service unterstützt die Initialisierung von Dienstobjekten. `ComponentComplex` ist eine Erweiterung der Schnittstelle `Component`. Die Klasse `ComponentFactory` und die Schnittstellen `Component` sowie `ComponentComplex` werden in der Component Service Specification [CB03a] spezifiziert.

Wie schon erwähnt, können in Openwings Konfigurationsdaten in so genannten Policy-Dateien abgelegt werden. Policy-Dateien werden im XML-Format abgelegt, siehe Abbildung 4.12.

Damit Policy-Schnittstellen verwendet werden können, um Konfigurationsdaten aus Policy-Dateien zu lesen und zu schreiben, müssen eine Reihe von Klassen und Dateien erstellt werden. In der Referenzimplementierung übernimmt dies die Klasse `PolicyBuilder`. Dem `PolicyBuilder` wird die Policy-Schnittstelle übergeben. Dieser erzeugt daraus:

1. Eine Klasse, die die Policy-Schnittstelle implementiert und Policy-Dateien lesen und schreiben kann.
2. Ein XML-Schema, welches das Format der XML-Datei festlegt.
3. Eine Beispielkonfiguration.
4. Eine Klasse, die ein Komandozeilen-Tool zum Anlegen von Policy-Dateien, implementiert.

Beim Laden und Ändern von Policy-Daten wird zuerst vom Component Service ein `PolicyLoader` angefordert. Dieser liefert über die `PolicyLoaderFactory` ein `PolicyLoader`-Objekt zurück. Dieses wird benutzt, um die Policy-Datei zu laden. Dazu wird dem `PolicyLoader` die Policy-Schnittstelle und der Dateiname der Policy-Datei übergeben. Diese Methode gibt ein Policy-Objekt zurück, auf das mit der Policy-Schnittstelle zugegriffen werden kann, um die Konfigurationsdaten zu lesen.

Die Policy-Schnittstelle verfügt auch über Methoden um Konfigurationsdaten zu ändern. Nach einer solchen Änderung kann die Methode `save` dazu benutzt werden, die Policy-Daten abzuspeichern.

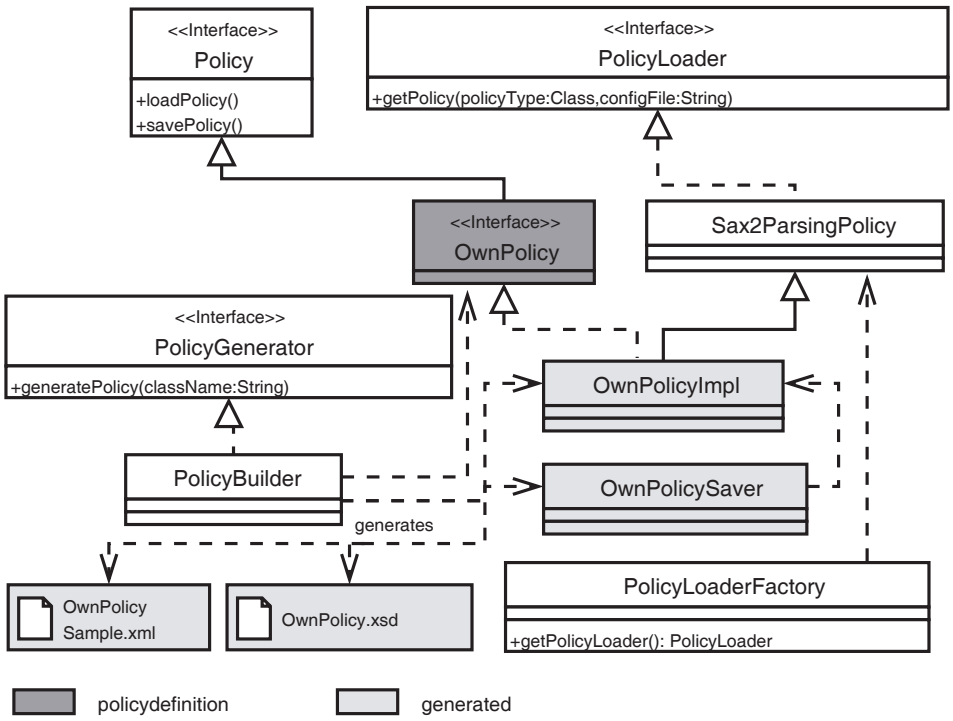

Abbildung 4.12: Openwings: Policy-Konzept

In einem *Kontext* können Policy-Objekte abgelegt werden. Klassen, die die Schnittstelle `PolicyListener` implementieren und sich als `PolicyListener` bei einem Kontext anmelden, werden benachrichtigt, wenn Policy-Objekte erzeugt, aktualisiert oder gelöscht werden.

Der Nachteil des Policy-Konzepts ist, dass ein Policy-Objekt nicht auf Änderungen der Policy-Datei reagiert.

Ein *Konnektor* ermöglicht die Kommunikation mit Dienstobjekten über eine Dienstschnittstelle unter Verwendung einer Middleware. Bei der Implementierung der Dienstobjekte braucht ein Entwickler die verwendete Middleware nicht zu berücksichtigen. Der Connector Service bietet die Möglichkeit, Konnektoren zu generieren. Ein Konnektorgenerator generiert einen Konnektor für eine bestimmte Dienstschnittstelle und eine bestimmte Middleware. Die Referenzimplementierung verfügt über Konnektorgeneratoren für RMI, IIOP und JMS. Bei der Kompilierung einer Komponente werden auch die dazugehörigen Konnektoren generiert.

Um Geräte, die über eine proprietäre Verbindungstechnologie angeschlossen sind, einzubinden, wäre es möglich, Konnektoren für die jeweiligen Verbindungstechnologien zu erzeugen. Allerdings wird kaum ein Gerät selbst ein Dienstobjekt bei einem Container Service anmelden und selbst, wenn ein Gerät sich anmelden könnte, werden die Schnittstellen von Geräten von Hersteller zu Hersteller unterschiedlich sind. Es gibt zum Beispiel noch keine allgemeine Schnittstelle für Kameras. Ein weiteres Problem besteht darin, dass ein Gerät selber oft seinen Verwendungszweck gar nicht kennt. So kann ein Lautsprecher als Alarmgeber aber auch zur Wiedergabe von Musik benutzt werden. Die Verwendung von Konnektoren, um Geräte direkt anzusprechen, ist also nicht sinnvoll.

Ein ähnliches Vorgehen sieht Openwings für die Einbindung von Legacy Systemen vor. Dazu muss zunächst eine Legacy System Schnittstelle spezifiziert werden, für die dann, je nach verwendeter Middleware, Konnektoren erstellt werden müssen. Bei der Einbindung von Legacy Systemen, unter Verwendung von Konnektoren, treten dieselben Probleme wie gerade beschrieben auf.

Eine Form der Policies ist der *Installable Deployment Descriptor* kurz *ICD*. In diesem werden die Bestandteile einer Komponente, Merkmale zur Identifikation einer Komponente und Informationen, um Komponenten auszuführen, beschrieben.

Damit eine Komponente installiert werden kann, muss sich in Ihrem Jar-Archiv mindestens die Datei `./policies/InstallableComponentDescriptor-Policy.xml` [BC03b] befinden. Diese Datei enthält einen Installable Component Descriptor im XML-Format. Eine ausführbare Komponente kann auch mehrere Installable Deployment Descriptoren enthalten. In dieser Arbeit könnte dies dazu benutzt werden, um mehrere Dienstobjekte einer Komponente zu erzeugen. Problematisch hierbei ist, dass alle Installable Component Descriptoren in dem zu installierenden Jar-Archiv enthalten sein müssen. Da im eHome-Bereich immer wieder neue Dienstobjekte erzeugt und wieder gelöscht werden müssen, zum Beispiel wenn der Benutzer sein Handy im Haus einschaltet, um auf vorhandene Dienste zuzugreifen, ist diese Methode zu statisch.

Openwings bietet einen grafischen Editor (Abbildung 4.13) an, der die Erstellung und Bearbeitung eines ICD vereinfacht. Dieser Editor kann Policy-Dateien mit ICDs im XML-Format laden, bearbeiten und abspeichern.

Ein ICD hat folgende Parameter (siehe auch Abbildung 4.13):

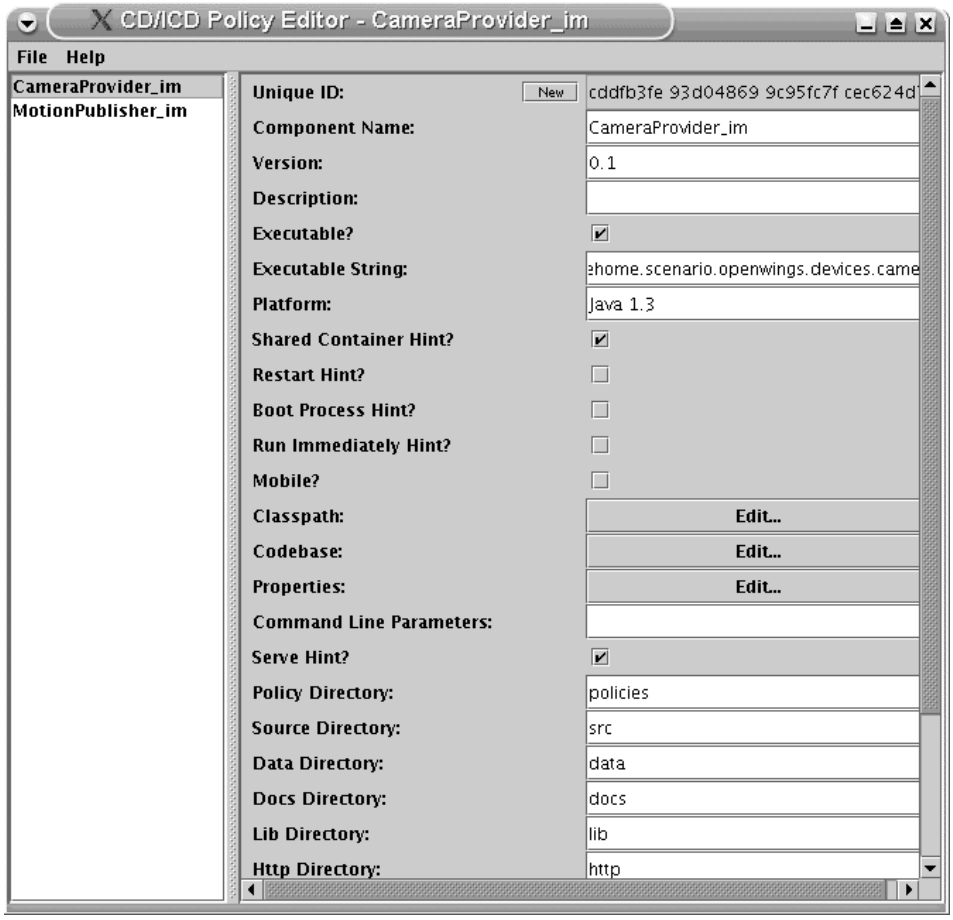

Abbildung 4.13: Installable Component Description Editor

Unique ID: 16-Byte Zahl, die eine Komponente eindeutig identifiziert. Diese kann automatisch erzeugt werden.

Component Name: Lesbarer Name der Komponente, der auch eindeutig sein sollte, da er im Rahmenwerk zur Identifikation der Komponente benutzt wird.

Version: Eine Zahl zur Identifizierung der Version der Komponente.

Icon: Im Openwings Explorer werden für jede Komponente Icons angezeigt. Soll einer Komponente ein eigenes Icon zugewiesen werden, wird die Icon-Datei hier angegeben.

Description: Eine einfache textuelle Beschreibung der Komponente, die der Benutzer lesen kann.

Executable?: Zeigt an, ob die Komponente ausführbar ist.

Executable String: Hier wird die auszuführende Klasse oder die auszuführende native Datei einer ausführbaren Komponente spezifiziert.

Platform: Für ausführbare Komponenten, die eine ausführbare Java-Klasse verwenden, wird hier die zu verwendende JVM angegeben. Wird eine native Datei zur Ausführung benutzt, wird hier das Betriebssystem angegeben.

Shared Container Hint?: Gibt an, ob von der Komponente gestartete Prozesse zusammen mit anderen Prozessen im selben Container laufen können.

Restart Hint?: Ist dieser Wert wahr, wird ein nicht ordnungsgemäß terminierender Prozess (eine ausgelöste Ausnahme wurde nicht abgefangen oder der Prozess endet mit einem Rückgabewert der ungleich 0 ist), der von einer Komponente ausgeführt wurde, erneut gestartet. Es ist nicht sichergestellt, dass der Prozess erneut gestartet wird. Dieses Verhalten ist im eHome-Bereich nicht sehr sinnvoll, da der Grund des Absturzes nicht erkannt werden kann.

Boot Process Hint?: Ist dieser Wert wahr, wird die Komponente beim Starten des Rahmenwerks mit gestartet. Ähnlich wie bei Restart Hint, ist nicht sichergestellt, dass die Komponente beim Start des Rahmenwerks ausgeführt wird.

Run Immediately Hint?: Gibt an, ob das Rahmenwerk sofort ein Dienstobjekt erzeugen soll, nachdem die Komponente installiert wurde. Auch wenn dieser Wert wahr ist, heißt das nicht, dass ein Dienstobjekt beim Start des Rahmenwerks erzeugt wird.

Serve Hint: Ist dieser Wert wahr, wird das Jar-Archiv, das die Komponente enthält, auf einem Http-Server installiert.

Mobile?: Zeigt an, ob ein Prozess von einer Plattform zu einer anderen übertragen werden kann. Die Übertragbarkeit eines Prozesses von einer Plattform zu einer anderen ist nicht trivial.

Classpath: Für eine ausführbare Komponente enthält der *Classpath* die zur Ausführung der Komponente benötigten Bibliotheken. Für nicht ausführbare Komponenten gibt der Classpath die von der Komponente angebotenen Bibliotheken an, die von anderen Komponenten benutzt werden können.

Resolvable Classpath: Der Inhalt des Resolvable Classpath wird vom Install Service an den Classpath angehängt. Im Unterschied zum Classpath kann der Resolvable Classpath Referenzen enthalten.

Codebase: Die hier aufgeführten Bibliotheken werden auf einem Http-Server zur Verfügung gestellt. Dadurch können die hier enthaltenen Bibliotheken auf allen Plattformen verwendet werden. Bibliotheken, die von anderen Komponenten benutzt werden sollen, sollten hier angegeben werden.

Resolvable Codebase: Der Inhalt von Resolvable Codebase wird vom Install Service an die Codebase angehängt. Im Unterschied zur Codebase kann die Resolvable Codebase Referenzen enthalten.

Properties: Die unter Properties spezifizierten (Key,Value)-Paare werden bei ausführbaren Komponenten beim Aufruf einer Komponente an die JVM weitergegeben. Dadurch können sie innerhalb des Prozesses benutzt werden. Werden für eine nicht ausführbare Komponente Properties spezifiziert, können diese Properties von anderen Komponenten referenziert werden.

Command Line Parameters: Hier werden die Parameter angegeben, die als Argumente für die `main`-Methode benutzt werden. Im Explorer lassen sich diese auch direkt eingeben.

bin, policies, data, docs, lib, http und source: Diese verweisen auf Verzeichnisse unterhalb des Basisverzeichnisses der Komponente.

Openwings erlaubt die Verwendung von Referenzen in Properties, Resolvable Codebase und Resolvable Classpath eines ICD. Referenzen können wie Variablen einer Programmiersprache benutzt werden, die aber nur einmalig mit neuen Werten belegt werden können.

Da Komponenten auf die lokale Festplatte einer Plattform installiert werden, kann sich der absolute Pfad auf unterschiedlichen Rechnern unterscheiden. Deshalb werden innerhalb des ICD einer Komponente Referenzen auf die absoluten Pfade benötigt. Dies ist eher eine Schwäche von Openwings. In der Resolvable Codebase der Komponente `Camera_im` wird die Referenz `Camera_im.libdir` verwendet, um die Bibliothek camera.jar einzubinden.

Wird in einer Komponente A eine Referenz auf eine Komponente B gesetzt, entsteht eine Beziehung zwischen den beiden Komponenten. Openwings kennt nur eine Art von Beziehungen zwischen Komponenten. Eine Komponente A hängt von einer anderen Komponente B ab, wenn Komponente A den Classpath, die Codebase oder Parameter der Komponente B benutzt. D.h. im ICD von Komponente A kommt eine Referenz vor, die auf Komponente B verweist. Diese Beziehung ist in der Sprechweise von Rio vom Typ

`requires`. Eine Komponente kann nur in den Zustand `Resolved` übergehen, wenn alle Komponenten, von denen sie abhängt, sich im Zustand `Resolved` befinden. In Openwings können aber keine Abhängigkeiten zwischen zwei ausführbaren Komponenten hergestellt werden.

Darüber hinaus können noch Referenzen auf Dateien oder auf Properties des Rahmenwerks angegeben werden. Diese werden hier nicht aufgeführt. Eine vollständige Liste aller Möglichkeiten befindet sich in [Ope03b], Using Properties.

Der für die Installation von Komponenten zuständige *Install Service* löst die in einem ICD benutzten Referenzen auf und ersetzt sie durch konkrete Werte. Die in den Properties verwendeten Referenzen werden direkt ersetzt. Deshalb kann Openwings auch nicht auf Änderungen der Properties reagieren. Hat sich der Wert einer Referenz geändert, muss die Komponente neu installiert werden, damit der Wert der Referenz geändert wird.

Der Install Service ermöglicht auch das Debugging von Referenzen. Tritt beim Auflösen einer Referenz ein Fehler auf, erzeugt der Install Service eine Datei, die eine Fehlerbeschreibung enthält. Diese Datei wird auch erzeugt, wenn eine referenzierte Komponente noch nicht im Zustand `Resolved` ist. Der Installer Service versucht in gewissen Zeitabständen immer wieder erneut die Referenzen aufzulösen.

Für das Starten von Komponenten ist der Container Service zuständig. Die Installation einer Komponente und der Start einer ausführbaren Komponente können auf verschiedene Arten angestoßen werden ([Ope03b] Installing Your Component):

1. Eine zu installierende Komponente wird in ein so genanntes *hotinstall*-Verzeichnis abgelegt. Der Install Service merkt automatisch, wenn in einem solchen Verzeichnis Komponenten hinzugefügt werden und installiert diese.

2. Über eine Autostart-Datei können zu installierende Komponenten angegeben werden.

3. Eine Komponente kann mit dem Openwings Explorer oder der Openwings Shell installiert und nach erfolgreicher Installation ausgeführt werden.

4. Der Install Service verfügt über eine Dienstschnittstelle, mit deren Hilfe Komponenten installiert werden können. Ein Container Service verfügt über eine Dienstschnittstelle, um ausführbare Komponenten zu starten. Diese Dienstschnittstellen könnten von einem Deployer benutzt werden, um Komponenten zu installieren und durch den Start von ausführbaren Komponenten Dienstobjekte zu erzeugen.

5. In einer ausführbaren Komponente können auch mehrere ICD-Dateien enthalten sein, die das mehrfache Starten einer Komponente unterstützen. Das Problem liegt darin, dass diese Policy-Dateien alle in der Komponente enthalten sein müssen.

Die Installation von Diensten wird in Openwings nicht unterstützt. Das Rahmenwerk kann nur einzelne Komponenten installieren und starten. Um mehrere Dienstobjekte einer Komponente automatisch zu erzeugen, können einer Komponente mehrere ICDs hinzugefügt werden.

Während der Installation einer Komponente durchläuft die Komponente mehrere Zustände. Abbildung 4.14 zeigt die möglichen Zustände einer Komponente und die dazugehörigen Transitionen. Folgende Transitionen sind möglich:

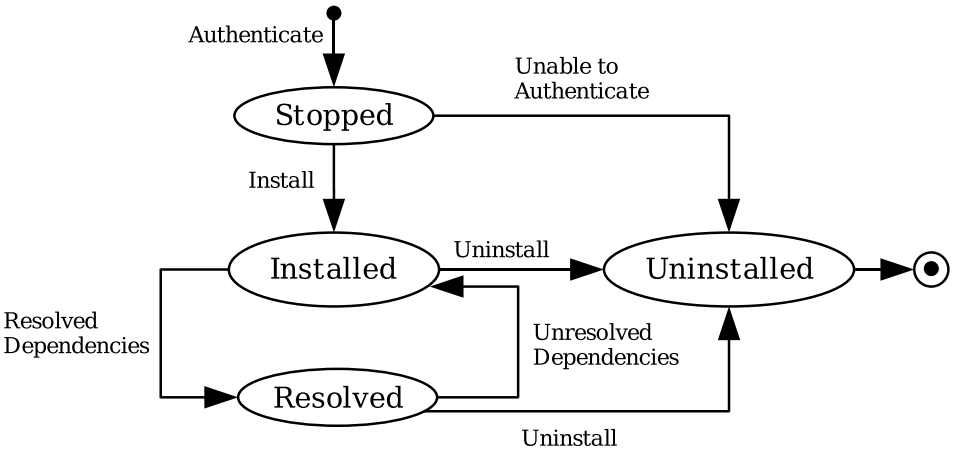

Abbildung 4.14: Openwings: Lebenszyklus einer Komponente

authenticate: Zuerst wird die Signatur der Jar-Datei [Sun99b], die die zu installierende Komponente enthält überprüft. Dadurch kann sichergestellt werden, dass eine Jar-Datei aus einer vertrauenswürdigen Quelle stammt.

install: Dann legt Install Service einem neuen Ordner für eine Komponente an, der als Basisordner für die Komponente dient. Darin wird die Jar-Datei der Komponente entpackt.

resolve: In diesem Schritt versucht der Install Service die im ICD verwendeten Referenzen durch konkrete Werte zu ersetzen. Nachdem alle Referenzen ersetzt werden konnten, wechselt die Komponente in den Zustand `Resolved`. Nicht ausführbare Komponenten können von diesem Zeitpunkt an von anderen Komponenten verwendet werden. Ausführbare Komponenten können ausgeführt werden. Wurde im ICD der Run Immediately Hint (s.o.) auf wahr gesetzt, wird die Komponente direkt vom Container Service ausgeführt.

uninstall: Bei der Deinstallation einer Komponente werden alle Dateien, die diese Komponente bereitstellt, gelöscht. Werden noch Prozesse der Komponente ausgeführt, wird die Komponente nicht sofort deinstalliert, sondern nur für eine spätere Deinstallation vorgemerkt. Dasselbe gilt, wenn eine Komponente von anderen Komponenten referenziert wird.

Die Spezifikation von Openwings sieht vor, dass zur Deinstallation vorgemerkte Komponenten beim Start des Install Service deinstalliert werden. Im eHome-Bereich ist das nicht sinnvoll, da das System prinzipiell ohne Unterbrechung laufen soll. Dadurch wird auch der Install Service nur selten neu gestartet.

Wird eine schon installierte Komponente erneut (mit derselben Version, Bezeichnung und Unique ID) installiert, werden die alten Dateien der Komponente überschrieben. Dienstobjekte, die von der Komponente gestartet wurden, laufen weiter. Sobald sie eine Datei erneut laden, bekommen sie die neue Version der Datei.

Der *Context Service* bietet eine Möglichkeit mehrere Plattformen zu verbinden. Die Plattformen innerhalb eines Kontexts bilden eine Art Vertrauensgemeinschaft. Dienstobjekte können innerhalb eines Kontexts uneingeschränkt mit Discovery-Mechanismen aufgefunden werden. Läuft ein Discovery-Manager außerhalb eines Kontexts, kann er keine Dienstobjekte innerhalb des Kontexts auffinden. Komponenten können nicht nur auf einzelnen Plattformen installiert und ausgeführt werden, sondern auch auf Kontexten. Der Context Service entscheidet dann, auf welcher Plattform eine Komponente installiert oder ausgeführt wird. Fällt eine Plattform aus, wird versucht, alle auf dieser Plattform installierten Komponenten und erzeugten Prozesse auf andere Plattformen zu verschieben. Einen Prozess so zu entwickeln, dass er verschiebbar ist, erfordert einen sehr hohen Aufwand, der in dieser Arbeit nicht durchgeführt wurde. Weiterhin können in einem Kontext Policy-Objekte abgelegt werden, die als Konfigurationsdaten verwendet werden können. Siehe für weitere Informationen zu Kontexten [Bie03].

4.2.2 Implementierung

Gegenüber der Implementierung des vereinfachten Sicherheits-Dienstes in Rio musste die Struktur etwas verändert werden. In Abbildung 4.15 sind die implementierten Komponenten und die Beziehungen zwischen den verwendeten Klassen und Schnittstellen dargestellt. In dieser Arbeit wurden ausführbare Komponenten benutzt. Diese verfügen über jeweils eine ausführbare Java-Klasse, um Dienstobjekte zu erzeugen. Dadurch kann der Container Service durch Starten der Komponenten Dienstobjekte erzeugen. Ausführbare Klassen sind in Abbildung 4.15 mit dem Stereotyp `executable` versehen.

Da in Openwings mehrere Dienstobjekte von einer Komponente erzeugt werden können, mussten für die Verwendung von zwei Kameras und zwei Bewegungsmeldern nicht jeweils zwei Komponenten erzeugt werden. Die Benennung der Klassen wurde entsprechend der Namenskonventionen von Openwings angepasst.

Als nicht ausführbare Komponenten wurden in dieser Arbeit die Komponenten `Security_im`, `EMail_im`, `Camera_im`, `Siren_im` und `MotionListener_im` implementiert, um die jeweiligen Dienstschnittstellen anzubieten.

Die Dienstobjekte `Camera1` und `Camera2`, bieten die synchrone Dienstschnittstelle `Camera` (Listing 4.10) an. Das `Security`-Dienstobjekt, ruft die Methode `getImage` eines `Camera`-Dienstobjekts über diese Schnittstelle auf, um Bilder zu erhalten, die es über die synchrone Dienstschnittstelle `EMail`, des `EMail`-Dienstobjekts versenden kann. Das `Security`-Dienstobjekt kennt nur die Dienstschnittstelle `Camera` bzw. `EMail`, welche die Dienstobjekte anbieten. Für das `Security`-Dienstobjekt ist die Implementierung der Dienstobjekte nicht so wichtig.

In der Komponente `CameraProvider_im` wurde die Dienstschnittstelle `Camera` in der Klasse `CameraProvider` implementiert. Wie schon in der Einleitung zu diesem Abschnitt erwähnt, müssen Dienstschnittstellen in eigenen Komponenten angeboten werden. Deshalb wird die Schnittstelle `Camera` von der Komponente `Camera_im` angeboten.

Im Unterschied zu Rio (Listing 4.4) wurde die Methode `getName` hinzugefügt. Diese dient dazu, die Bezeichnung eines `Camera`-Dienstobjekts abzufragen. Im Gegensatz zu Rio besitzt Openwings keine einfache Möglichkeit, die Bezeichnung eines Dienstobjekts zu erhalten. In der Implementierung für Rio wurde einfach der Name der JSB verwen-

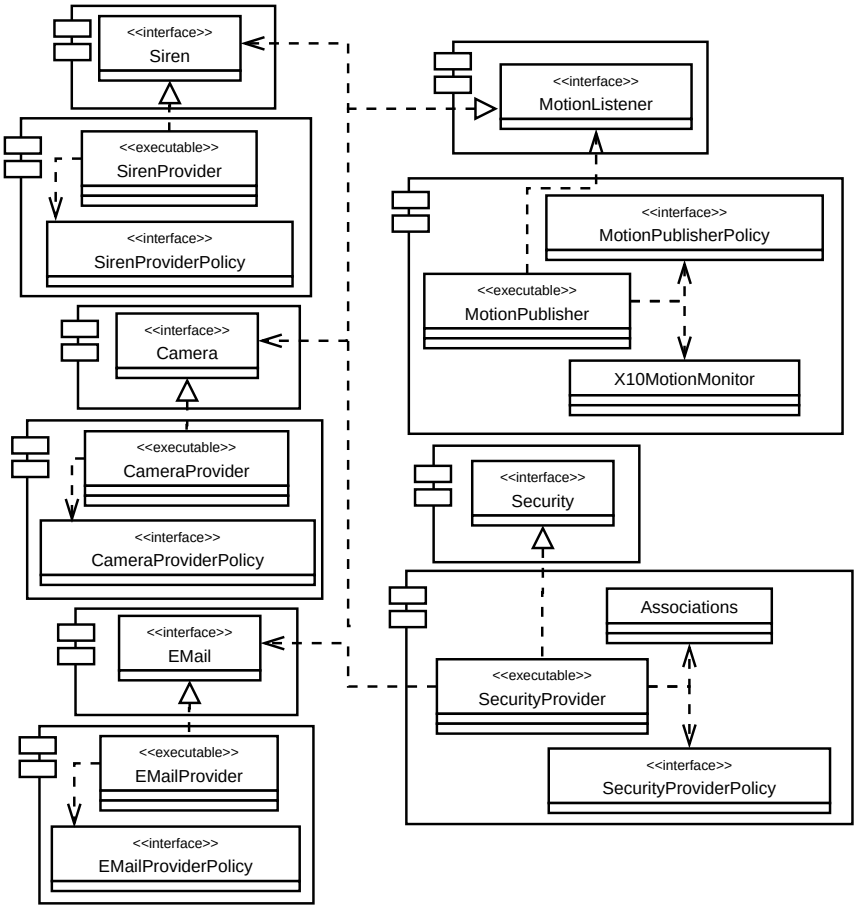

Abbildung 4.15: Openwings Komponenten für vereinfachten Sicherheits-Dienst

```
1  package ehome.scenario.openwings.devices.camera;
2
3  import javax.swing.ImageIcon;
4  import java.rmi.RemoteException;
5
6  public interface Camera{
7
8      public String getName() throws RemoteException;
9      public ImageIcon getImage() throws RemoteException;
10     public String getLocation() throws RemoteException;
11     public void setLocation(String loc) throws RemoteException;
12     public String test() throws RemoteException;
13 }
```

Listing 4.10: Synchrone Dienstschnittstelle Camera

```
1  package ehome.scenario.openwings.sensors.motion;
2
3  import java.rmi.RemoteException;
4
5  public interface MotionListener {
6      public void motionDetected(
7          String sender,
8          String location,
9          long timestamp) throws RemoteException;
10 }
```

Listing 4.11: Asynchrone Dienstschnittstelle für Ereignisse eines Bewegungsmelders

det. Würden in Rio mehrere Dienstobjekte einer JSB erzeugt, müsste auch in Rio diese Methode hinzugefügt und implementiert werden. Unter Verwendung von Parametern kann Openwings einer Schnittstelle eine Bezeichnung hinzufügen, die als Bezeichnung für Dienstobjekte benutzt werden kann.

Die Dienstobjekte `Motion1` und `Motion2` bieten die asynchrone Dienstschnittstelle `MotionListener` (Listing 4.11) an. Sie erzeugen mithilfe dieser Schnittstelle Ereignisse. Das `Security`-Dienstobjekt benutzt diese Schnittstelle, um Ereignisse zu empfangen. Bei der Anmeldung der asynchronen Dienstschnittstelle `MotionListener` wird mehrfach versucht, die Schnittstelle anzumelden. Bei erfolgreicher Anmeldung wird ein Objekt zurück geliefert, das zum Erzeugen von Ereignissen benutzt werden kann.

In Openwings werden also nicht, wie in Rio, Instanzen einer Eventklasse erzeugt, und in diese die Informationen zu einem Ereignis abgelegt, sondern die Informationen beim Methodenaufruf als Parameter übergeben. Beim Auslösen eines Ereignisses mit der Methode `motionDetected` wird die Bezeichnung des Bewegungsmelders, der Ort, den der Bewegungsmelder überwacht und der Zeitpunkt des Ereignisses übergeben.

In der Arbeit wurde die Policy-Schnittstelle `CameraProviderPolicy` (Listing 4.12) definiert, um Konfigurationsdaten für `Camera`-Dienstobjekte bereitzustellen. In den ausführbaren Komponenten `CameraProvider_im` und `MotionPublisher_im` werden sie benutzt, um die Policy-Datei anzugeben, die die Konfigurationsdaten zur Initialisierung eines Dienstobjektes enthält.

Die Komponente `Camera_im` enthält die ausführbare Klasse `CameraProvider`, die zum Starten der Komponente benutzt wird. Beim Ausführen der Komponente `Camera_im` wird die `main`-Methode von `CameraProvider` (Listing 4.13) durch einen Container Service aufgerufen. In der `main`-Methode wird zunächst überprüft, ob genau ein Parameter übergeben wurde. Dieser Parameter spezifiziert die zu verwendende Policy-Datei. Die Initialisierung von Dienstobjekten wird im Konstruktor der Klasse durchgeführt. Sie könnte genauso gut in der `main`-Methode selbst durchgeführt werden.

Die einzelnen Schritte der Implementierung lassen sich an den Listings 4.14 und 4.15 nachvollziehen:

Zuerst muss ein Component Service Objekt erzeugt werden, das die Anmeldung eines Dienstobjekts ermöglicht. Dazu wird eine `ComponentFactory` benutzt. Component Service Objekt implementiert die Schnittstelle `Component` oder `ComponentComplex`. Es kann benutzt werden, um ein Dienstobjekt zu initialisieren (Zeile 11).

```
1  package ehome.scenario.openwings.devices.camera;
2
3  import net.openwings.policy.Policy;
4
5  public interface CameraProviderPolicy extends Policy {
6
7      public void setServiceName (String value);
8      public String getServiceName ();
9
10     public void setLocation (String value);
11     public String getLocation ();
12
13     public void setURL (String value);
14     public String getURL ();
15 }
```

Listing 4.12: Policy-Schnittstelle für Camera-Dienstobjekte

```
1  public static void main(String[] args) {
2
3      // Create a new service provider.
4      CameraProvider cameraProvider = null;
5
6      //Minimale Parameterüberprüfung
7      if (args.length==1)
8          //Spezielle Konfiguration
9          cameraProvider = new CameraProvider(args[0]);
10     else
11         //Standard Konfiguration
12         cameraProvider = new CameraProvider("");
13 } //end main
```

Listing 4.13: main-Methode der Klasse `CameraProvider`

```
1  public class CameraProvider implements Camera, ProcessShutdown {
2      private String policyFilename = "",
3                     servicename = null,
4                     location = null,
5                     url = null;
6      ProvideServiceParameters params = null;
7      private PolicyLoader policyLoader;
8
9      public CameraProvider(String policyFilename) {
10
11         /**Get a complex component service**/
12         ComponentComplex component =
13             ComponentFactory.getComponentComplex();
14
15         /**Register the shutdown callback **/
16         ContainerAccess containerAccess =
17             component.getContainerAccess();
18
19         if (containerAccess != null) {
20             containerAccess.registerProcessShutdown(this);
21         }
22         // This execption block will catch any errors trying encountered
23         // in providing the service
24         try {
25
26             //Anmeldung der synchronen Dienstschnittstelle siehe
27             //Listing 4.16
28             /**Get Policy **/
29             policyLoader = component.getPolicyLoader();
30             this.policyFilename = policyFilename;
31             CameraProviderPolicy policy =
32                 (CameraProviderPolicy) policyLoader.getPolicy(
33                     CameraProviderPolicy.class,
34                     policyFilename);
35             determineParameters(policy); //get Attributes from policy
36
37             log("Service started.");
38
39         } catch (Exception e) {
40             e.printStackTrace();
41             component.shutdown(); // Shutdown Gracefully
42         }
43     }
44
45     public void shutdown() {
46         ComponentFactory.getComponent().shutdown();
47         log("Camera provider Location: " + location + "shut down");
48         System.exit(0);
49     }
50 }
```

Listing 4.14: Initialisierung eines Dienstobjekts von `CameraProvider_im`

```
1  private void determineParameters(CameraProviderPolicy policy) {
2      servicename = policy.getServiceName();
3      location = policy.getLocation();
4      url = policy.getURL();
5  }
```

Listing 4.15: Auslesen der Parameter einer Camera Provider Policy

```
1 //Distribute the Camera interface on the cameraProvider
2 Object distributedObject =
3     component.distributeObject(Camera.class, this);
4
5 //Publish Camera interface on the distributedObject.
6 UniqueID serviceID = component.
7     provideService(Camera.class, distributedObject);
```

Listing 4.16: Anmeldung der synchronen Dienstschnittstelle `Camera`

In Zeile 15 wird ein Objekt registriert, das die Methode `shutdown` implementiert. Dazu wird zunächst ein Container Service angefordert und dann bei diesem das Objekt, das die Schnittstelle implementiert, angemeldet. Der Container Service ruft `shutdown` auf, wenn der Prozess beendet wird.

In Zeile 28 wird die zu dem Dienstobjekt innerhalb der Komponente abgelegte Policy-Datei geladen und anschließend zur Konfiguration des Dienstobjekts verwendet.

Die Anmeldung der synchronen Dienstschnittstelle `Camera` ist in Listing 4.16 und die der asynchronen Dienstschnittstelle `MotionListener` in Listing 4.17 dargestellt. Listing 4.16 zeigt die Anmeldung der Dienstschnittstelle `Camera`.

Bei der Anmeldung der asynchronen Dienstschnittstelle `MotionListener` (Listing 4.17) wird mehrfach versucht, die Schnittstelle anzumelden.

In Openwings werden die Informationen, die das Ereignis beschreiben, als Parameter übergeben. Es werden keine Eventobjekte erzeugt. Dadurch entfällt die Initialisierung von Eventobjekten und das anschließende Extrahieren der Information auf der Seite des Ereignis-Empfängers (Listing 4.7).

In der Arbeit wurden keine Dienstparameter verwendet, da keine grafischen Benutzerschnittstellen für Dienste entwickelt wurden und kein Bedarf bestand, Attribute zu benutzen, um die Menge der verwendeten Dienstschnittstellen einzugrenzen. Eine alternative Implementierung des Security-Dienstobjekts hätte auch nach der zu einem `Motion`-Dienstobjekt gehörigen Kamera mittels Attributen suchen können. Eine Suche nach einer Kamera, in einem bestimmten Raum sähe wie in Listing 4.19 dargestellt aus. Die drei anderen `Parameter`-Klassen werden ähnlich benutzt.

`Security`-Dienstobjekte werden als Listener für die synchronen Dienstschnittstellen `Camera`, `EMail` und `Security` sowie für die asynchrone Dienstschnittstelle `MotionListener` angemeldet. Dazu wurden die Methoden `serviceProvided` und `providedServiceRemoved` (Listing 4.20), sowie `servicePublished` und `publishedServiceRemoved` (Listing 4.21) in der Klasse `SecurityProvider` implementiert.

Die Komponente `SecurityProvider_im` ist so implementiert, dass Referenzen auf Dienstobjekte verwaltet werden. Dadurch muss nicht immer wieder nach Dienstobjekten gesucht werden. Wird eine Dienstschnittstelle hinzugefügt, die vom `Security`-Dienstobjekt benutzt werden kann, wird ein Proxy-Objekt gespeichert, das die Verwendung des Dienstobjekts über die angemeldete Dienstschnittstelle erlaubt. Das Proxy-Objekt wird wieder gelöscht, wenn die Dienstschnittstelle abgemeldet wird.

Für `Security`-Dienstobjekte ist keine Reaktion auf das Hinzufügen oder Entfernen von Bewegungsmeldern vorgesehen. In dieser Arbeit wurden Security-Dienstobjekte

```
ServiceResult result = null;

// This block catches InvalidServiceException which should
// not occur if the published interface was correctly written.
try {
    // Try getting a result back a few times before continuing
    do {
        result = component.publishService(MotionListener.class);
        if (result == null) {
            log("No asynchronious Connector Service found...");
            Thread.sleep(1000);
        }
    } while (result == null && count++ < 60);

    // Got a service result, so set the service value and start this thread.
    if (result != null) {
        service = (MotionListener) result.getService();
    } else {
        log(
            "Could not publish service - check to make sure "
                + "message server process is running.");
    }
}catch (Exception e) {
            e.printStackTrace();
            shutdown();
}
\end{lstlisting}
```

Listing 4.17: Anmeldung der asynchronen Dienstschnittstelle MotionListener

```
public void motionDetected(
    String sender, String location,
    long timestamp) {

    if (!activated) return; //Security not activated

    giveAlarm(sender, location,
              new Date(timestamp).toString());
}
```

Listing 4.18: Benutzung der Schnittstelle `MotionListener` in `SecurityProvider`

```
Attribute[] a =
    new Attribute[] { new Attribute("location", location) };
UseServiceParameters usp = new UseServiceParameters();
usp.setRequiredAttributes(a);

ComponentComplex componentComplex =
  ComponentFactory.getComponentComplex();

ServiceResult[] sr =
    componentComplex.useService(Camera.class, usp);
```

Listing 4.19: Suche nach Dienstobjekten mit Use Service Parametern

```
1  private Map cameras;
2
3  private EMail mailer = null;
4  private UniqueID mailerID = null;
5
6  private Siren siren = null;
7  private UniqueID sirenID = null;
8
9  public void serviceProvided( java.lang.Class serviceInterface,
10                              ServiceResult serviceResult) {
11     log("Service Provided was invoked");
12     if (serviceInterface.equals(Camera.class)) {//doppeltes Hashing
13         try {
14             String name =
15                 ((Camera) (serviceResult.getService())).getName();
16             cameras.put(serviceResult.getServiceID(), name);
17             cameras.put(name, serviceResult.getService());
18             log(">> Found Camera:  " + name + " Id: "
19                     + serviceResult.getServiceID().toString());
20         } catch (RemoteException e) {
21             log("Could not reach new Camera");
22             e.printStackTrace();
23         }
24     } else if (serviceInterface.equals(EMail.class)) {
25         mailer = (EMail) serviceResult.getService();
26         mailerID = serviceResult.getServiceID();
27         log(">> Found EMail:  " + mailerID.toString());
28     } else if (serviceInterface.equals(Siren.class)) {
29         siren = (Siren) serviceResult.getService();
30         sirenID = serviceResult.getServiceID();
31         log(">> Found Siren:  " + sirenID.toString());
32     } else {
33         log(">> Found Unknown Service");
34     }
35 }
36
37 public void providedServiceRemoved(UniqueID service) {
38     if (service.equals(sirenID)) {
39         siren = null; sirenID = null;
40         log(">> Lost Siren:   " + service.toString());
41     }else if (service.equals(mailerID)) {
42         mailer = null; mailerID = null;
43         log(">> Lost Mailer:   " + service.toString());
44     }
45     else {
46         String name = (String) cameras.get(service);
47         if (name != null) { //doppeltes Hashing
48             cameras.remove(name);
49             cameras.remove(service);
50             log(">> Lost Camera:   " + name + " Id: " +
51                     service.toString());
52         }
53     }
54 }
```

Listing 4.20: Implementierung von `UseServiceListener` in `SecurityProvider`

```
1  //Methods executed if asynchronous services are provided or gone.
2
3  public void serviceAvailable (java.lang.Class intf,
4                                ServiceResult result)
5  {
6    log(intf.toString() + " service is now available...");
7  }
8
9  public void serviceUnavailable (UniqueID uid)
10 {
11   log("MotionListener is unavailable...");
12 }
```

Listing 4.21: EventServiceListener

```
1  Datei: CameraProviderPolicyCamera1.xml
2
3  <?xml version="1.0" encoding="UTF-8"?>
4  <CameraProviderPolicy
5      xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
6      xsi:noNamespaceSchemaLocation='CameraProviderPolicy.xsd'
7
8      URL="http://myhouse.de/cam1.jpg"
9      location="Wohnzimmer"
10     serviceName="Camera1"
11 />
```

Listing 4.22: Policy des Dienstobjekts `Camera1`

auch nur als `EventServiceListener` angemeldet, um Meldungen beim An- und Abmelden von Dienstobjekten, die Ereignisse mit der Schnittstelle `MotionListener` erzeugen, zu erhalten. Für die Funktion von `Security`-Dienstobjekten ist dies nicht notwendig. Deshalb werden in den Methoden der Schnittstelle `EventServiceListener` (Listing 4.21) nur Log-Messages ausgegeben. Eine mögliche Erweiterung wäre, `Security`-Dienstobjekte einfach zu deaktivieren, wenn keine Bewegungsmelder mehr vorhanden sind. Die Möglichkeit, Dienstobjekte beim An- und Abmelden von Ereignis-Auslösern zu benachrichtigen, ist ein echter Vorteil gegenüber Rio.

In Listing 4.22 ist die Policy-Datei, die die Konfiguration für das Dienstobjekt `Camera1` enthält, angegeben. Um diese Policy-Datei zu lesen, wird der in Listing 4.14 Zeile 28 ff. angegebene Code verwendet.

Da in Openwings keine Abhängigkeiten zwischen zwei ausführbaren Komponenten hergestellt werden, kann zum Beispiel auch keine Beziehung zwischen den Komponenten `SecurityProvider_im` und `CameraProvider_im` hergestellt werden. Die Komponente `Security_im` benötigt die Schnittstelle `Camera`, um Dienstobjekte, die diese Schnittstelle implementieren, zu benutzen.

Deshalb wurde jede Schnittstelle in einer eigenen Komponente angeboten, zum Beispiel wird die Schnittstelle `Camera` von der Komponente `Camera_im` angeboten. Statt einer Verbindung zwischen `SecurityProvider_im` und `CameraProvider_im` wurde eine Beziehung von `SecurityProvider_im` zur Komponente `Camera_im`

hergestellt. Dazu wurde im Classpath der Komponente `SecurityProvider_im` und der Komponente `Camera_im` die folgenden zwei Zeilen hinzugefügt:

- `${Camera_im.classpath}` *//In der Komponente SecurityProvider_im*
- `${Security_im.classpath}` *//In der Komponente Camera_im*

4.2.3 Zusammenfassung

Openwings benutzt wie Rio Java-Schnittstellen zur Spezifikation von Dienstschnittstellen. Openwings unterstützt auch asynchrone Dienstschnittstellen. Dies ist aber im eHome-Bereich kein nennenswerter Vorteil gegenüber dem von Rio unterstützten Event-Modell.

Neben Dienstschnittstellen können in Openwings auch Schnittstellen für die Bereitstellung von Konfigurationsdaten definiert werden, so genannte Policy-Schnittstellen. Der Nachteil ist, dass Openwings zwar Policy-Daten in Dateien ablegen kann, aber Veränderungen an diesen Dateien nicht überwacht werden. Außerdem ist die Verwendung von Policy-Generatoren aufwändig, da die Unterstützung durch Editoren nur minimal ist. Deshalb ist es nicht sinnvoll, für jede Komponente eigene Policy-Schnittstellen zu generieren, auch wenn dadurch komplexere Datenstrukturen für Konfigurationsdaten benutzt werden können. Noch fehlt in Openwings ein allgemeiner Editor für Policy-Daten. Deshalb wurden in dieser Arbeit die Beziehungen zwischen Bewegungsmeldern und Kameras auch in einer eigenen Datei abgelegt, obwohl diese auch in einer Policy abgelegt werden könnten.

Bei der Installation von Komponenten und Erzeugung von Dienstobjekten fehlt es Openwings, genauso wie Rio, an der notwendigen Dynamik. Referenzen, die einmal aufgelöst wurden, werden erst bei erneuter Installation einer Komponente aufgelöst, und damit mit neuen Werten belegt. Außerdem kann ein Install Service zwar einen Container Service dazu veranlassen, eine Komponente mehrfach zu starten, um mehrere Dienstobjekte zu erzeugen, dies ist jedoch nur bei der Installation möglich.

Der Container Service überwacht die auf ihm laufenden Dienste (Prozesse, die Dienste erzeugen). Löst einer dieser Dienste eine Ausnahme aus, wird die Komponente neu gestartet. Eine derartige Fehlerbehandlung von Openwings ist nicht ausreichend für die automatische Erkennung von Diensten, die fehlerhaft arbeiten, oder nicht mehr erreichbar sind.

In Tabelle 4.2 ist wieder der Entwicklungsaufwand in Codezeilen für den vereinfachten Sicherheitsdienst in Openwings angegeben. Dabei fällt die enorme Menge an zu erstellendem xml-Code auf. Zwar lassen sich die xsd-Dateien automatisch generieren und auch in den build-Dateien müssen in der Regel immer nur zwischen 50 und 100 Zeilen verändert werden, doch sind auch die verbleibenden 4106 Zeilen für die Konfigurations- und Strukturbeschreibung durch Policies ein erheblicher Aufwand. Für eine einfache und für den Entwicklungsprozess zeitoptimale Anwendung müsste der Großteil dieses Codes auch durch Werkzeuge erzeugt werden.

Openwings stellt Konzepte bereit, um eine automatische Installation von Komponenten und eine automatische Instanzierung von Dienstobjekten zu ermöglichen. Allerdings ist noch ein großer Aufwand für die Entwicklung eines plattformspezifischen Deployers auf Basis von Openwings notwendig.

Die Stärken von Openwings sind die Unterstützung verschiedener Kommunikations- und Discovery-Technologien. Um weitere Protokolle zum Suchen von Dienstobjekten

Datei (-typ)	Codezeilen
*.java-Dateien	4777
davon Gerätetreiber	3997
davon Erweiterungs-Dienste	800
Policies und *.xml-Dateien	10605
davon 12x owbuild.xml	5191
und 13x build.xml	1201
Policies und *.xml-Dateien ohne (ow-)build.xml	4106
build.properties	6
generierte *.xsd-Dateien	7509
Summe mit allen *.xml, ohne generierte	15388
Summe mit *.xml ohne (ow-)build.xml und generierte	8996

Tabelle 4.2: Geschriebene Quellcodezeilen für den vereinfachten Sicherheitsdienst in Openwings.

hinzuzufügen, können Discovery-Plugins entwickelt werden. Openwings würde sich eignen mehrere OSGi-Plattformen miteinander zu verbinden. Auf Basis von Openwings ließe sich ein globaler Deployer entwickeln, der plattformspezifische Deployer über die verschiedensten Protokolle ansprechen kann.

4.3 OSGi

4.3.1 Beschreibung

Die *Open Service Gateway initiative (OSGi)* ist ein Zusammenschluss von über 50 Unternehmen, die gemeinsam einen offenen Standard für Service Gateways erarbeiten. Sie wurde im März 1999 gegründet. Die OSGi Service Plattform Specification (kurz OSGi-Spezifikation) definiert eine Umgebung, in der Dienste erbracht werden können. Sie ist im März 2003 in der Version 3 erschienen [The05, Ope03a]. Seit September 2006 gibt es die Spezifikation in der Version 4. In dieser Arbeit wurden allerdings nur OSGi in der Spezifikation 2 und 3 verwendet. Das Komponenten-Rahmenwerk wird in der Framework Specification beschrieben, die Teil der OSGi Spezifikation ist.

In OSGi wird nur das Komponentenmodell selbst als Rahmenwerk bezeichnet. Alle anderen Hilfsmittel werden als Dienste bezeichnet. Die Menge dieser Dienste beruht auf Erfahrungen mehrerer Unternehmen und wird innerhalb eines Gremiums festgelegt. Für den hier durchgeführten Rahmenwerksvergleich wurde der mBedded Server 5.1 der Firma Prosyst [Prob] verwendet. Dieser implementiert die OSGi-Spezifikation Version 2 [Ope02]. Mit Oscar [OSG04], Knopflerfish [Kno04], Apache Felix [KLZ+05] und Equinox [FGH+] gibt es auch mittlerweile viele freie Implementierungen des OSGi-Rahmenwerks.

Jede Implementierung von OSGi bietet ihre eigenen Management-Werkzeuge an. In Abbildung 4.16 ist die von Prosyst mBedded Server 5.1 angebotene Management Console dargestellt.

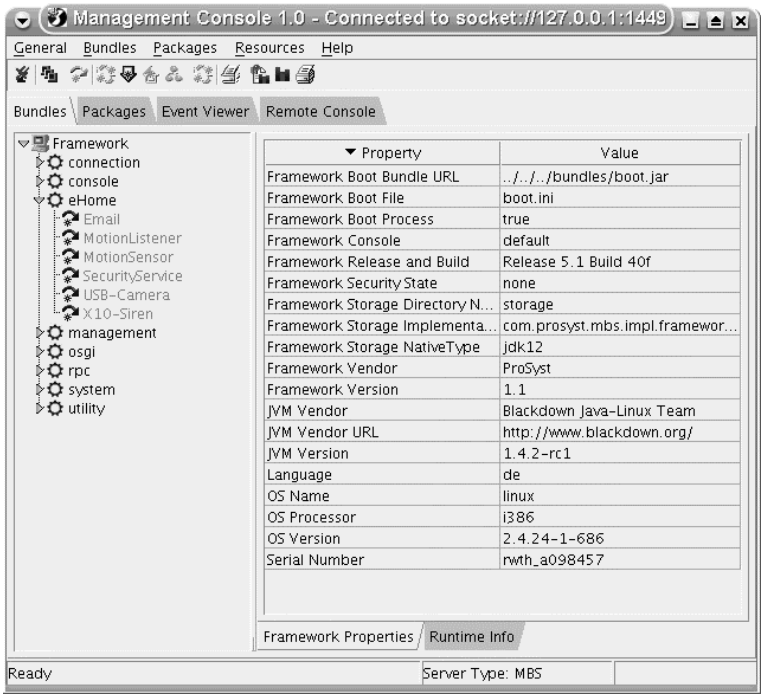

Abbildung 4.16: OSGi Management Console

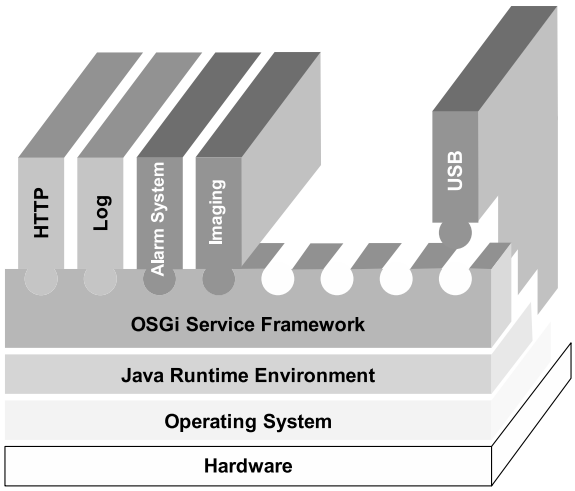

Abbildung 4.17: Architektur der OSGi-Plattform

In Abbildung 4.17 ist die Architektur einer OSGi-Plattform dargestellt. Auf der untersten Ebene befindet sich die Hardware der Plattform, an die Geräte über proprietäre Verbindungstechnologien angeschlossen werden können. Anders als Openwings und Rio, unterstützt OSGi die Verwendung dieser Verbindungstechnologien. Auf der darüber liegenden Schicht liegt das Betriebssystem, das den Zugriff auf die Hardwareschnittstellen erlaubt, und auf das die darüber liegende JRE aufbaut. Die JRE bietet eine Laufzeitumgebung für Java-Objekte. OSGi bietet keinen eigenen Container als Laufzeitumgebung für Dienstobjekte, sondern benutzt die von der JRE bereitgestellte Laufzeitumgebung. Auf der obersten Schicht befinden sich die installierten Bundles. In OSGi werden Komponenten als *Bundles* bezeichnet. Im Folgenden werden die Dienste des OSGi-Rahmenwerks kurz vorgestellt:

Package Admin Service: Ermöglicht es einem Administrator in die Auflösung von Abhängigkeiten zwischen Bundles auf Paketebene einzugreifen.

Permission Admin Service: OSGi bietet genauso wie Openwings ein Sicherheitskonzept an. Dabei unterstützt es ebenfalls die von Java angebotenen Mechanismen zur Codesicherheit. Darüber hinaus bietet OSGi ein rollenbasiertes Sicherheitskonzept, ähnlich dem von Openwings, an. Die unterstützten Sicherheitskonzepte sind für den eHome-Bereich ausreichend. Der Einsatz wurde in dieser Arbeit nicht untersucht. [CG01], Kapitel 9. bietet eine detaillierte Einführung in die Sicherheitskonzepte von OSGi.

Configuration Admin: Mithilfe dieses Dienstes können Dienstobjekte mit Parametrisierungsdaten versorgt werden, die zum Beispiel zur Konfigurierung von Dienstobjekten benutzt werden können.

Device Access Service: Dieser Dienst ist hauptsächlich für das Auffinden von Treibern für Geräte verantwortlich.

Preferences: *Preferences* sind Konfigurierungsdaten, die innerhalb des Rahmenwerks abgelegt werden können und auf die jedes Dienstobjekt zugreifen kann. Sie bieten eine Möglichkeit Konfigurationsdaten dienstübergreifend abzulegen. So kann zum Beispiel die bevorzugte Schriftart und Schriftgröße für die Darstellung von Informationen abgelegt werden ([Ope02], Kapitel 11).

Die hier aufgeführten Dienste sind die Dienste, die für die Erzeugung und Konfiguration von Dienstobjekten wichtig sind. Darüber hinaus sind innerhalb des OSGi-Rahmenwerks ab Spezifikation 3 eine Reihe weiterer Dienste definiert, die die Implementierung von Diensten vereinfachen, zum Beispiel ein XML-Parser ([Ope03a], Kapitel 17) und ein Logging-Dienst ([Ope03a], Kapitel 9).

Wie schon erwähnt, werden Komponenten in OSGi Bundles genannt. Diese werden innerhalb des OSGi-Rahmenwerks installiert und vom Rahmenwerk verwaltet.

Ein Bundle wird genauso wie eine Openwings-Komponente in einer Jar-Datei verpackt zur Installation angeboten. Bei der Installation eines Bundles werden die Bestandteile des Bundles installiert und Abhängigkeiten zwischen Bundles aufgelöst. Jedes Bundle muss eine Manifest-Datei enthalten, die das Bundle beschreibt. In OSGi lassen sich die Inhalte eines Bundles in drei Gruppen unterteilen:

Pakete: Ein Bundle kann mehrere Java-Pakete enthalten. Diese Pakete können auch Bestandteil von Bibliotheken sein. Die `SecurityProvider`-Komponente enthält zum Beispiel die Pakete: `ehome.scenario.osgi.logic.security` und `ehome.scenario.osgi.logic.security.impl`

Im Classpath eines Bundles wird der Suchpfad für Java-Klassen, die ein Bundle benutzen kann, angegeben. Ein Bundle kann seine Pakete exportieren, sodass andere Bundles sie importieren können, um die darin enthaltenen Klassen und Schnittstellen zu benutzen.

Native Bibliotheken: Neben Java-Klassen kann ein Bundle auch nativen Code enthalten. In einer Manifest-Datei können abhängig vom Betriebssystem, auf dem das Rahmenwerk läuft, unterschiedliche native Bibliotheken eingebunden werden, die innerhalb des Bundles verwendet werden können. Zum Beispiel kann für Windows 98 eine andere native Bibliothek benutzt werden als unter Linux, um eine USB-Kamera anzusprechen. Da die Bibliotheken abhängig vom Betriebssystem angegeben werden, kann das Bundle ohne Anpassung auf beiden Betriebssystemen installiert werden.

Hilfsmittel: Hilfsmittel sind alle anderen innerhalb eines Bundles enthaltenen Dateien, die vom Rahmenwerk für Dienstobjekte zugänglich gemacht werden sollen. Dienstobjekte können über die Schnittstelle `BundleContext` auf diese Dateien zugreifen.

Genauso wie Rio und Openwings verwendet OSGi Java zur Spezifikation von Schnittstellen. In OSGi werden dafür nur *synchrone Dienstschnittstellen* verwendet. Dabei kann jede Java-Schnittstelle als Schnittstelle eines Dienstobjekts benutzt werden. Es gibt keine Einschränkungen bezüglich der Typen von Parametern und Rückgabewerten. Da OSGi

ein Einzelplatzrahmenwerk ist, müssen auch keine `RemoteExceptions` abgefangen werden.

Um Dienstobjekte eines Bundles zu erzeugen, kann ein sogenannter *Aktivator* benutzt werden. Ein Aktivator ist eine Klasse, die die Schnittstelle `BundleActivator` implementiert. Diese Schnittstelle spezifiziert zwei Methoden `start` und `stop`, die beim Starten und Stoppen eines Bundles aufgerufen werden. In der `start`-Methode können Dienstobjekte erzeugt und angemeldet werden. Die `stop`-Methode dient dazu, die verwendeten Ressourcen wieder freizugeben. Da alle in dieser Arbeit implementierten Bundles gestartet werden können, muss in jeder Komponente eine Klasse `Activator` implementiert werden.

Den Methoden `start` und `stop` wird vom Rahmenwerk ein Objekt, das die Schnittstelle `BundleContext` implementiert, als Parameter übergeben. Dieses Objekt ermöglicht es dem Aktivator auf das Rahmenwerk zuzugreifen, um:

- Dienstobjekte anzumelden,
- Dienstobjekte anderer Bundles zu benutzen,
- Bundles zu installieren, deinstallieren und aktualisieren,
- die Menge aller installierten Bundles abzufragen,
- eine Datei innerhalb des Bundles zu erzeugen (siehe [CG01]),
- Dateien als Hilfsmittel zu benutzen und anzulegen (siehe [CG01]) und
- Listener zum Empfang von Events anzumelden.

Der Activator kann das `BundleContext`-Objekt an beliebige Objekte weitergeben. Dadurch können auch andere Objekte als der Activator selbst zum Beispiel Dienstobjekte anmelden. Ein mit einem `BundleContext`-Objekt angemeldetes Dienstobjekt gehört zu dem jeweiligen Bundle. Wird das Bundle gestoppt, wird das Dienstobjekt automatisch abgemeldet.

Um ein Dienstobjekt zu implementieren, müssen die Methoden der Dienstschnittstelle implementiert werden und es muss Code zur Anmeldung und Initialisierung des Dienstobjekts geschrieben werden.

Die Konfigurationsdaten, mit denen das Dienstobjekt angemeldet wird, werden als Properties bezeichnet. Properties sind Mengen von (Key,Value)-Paaren und können verwendet werden, um Dienstobjekte zu finden und um weitere Informationen zu deren Benutzung zu erhalten.

Die Verwendung von Properties ähnlich der Verwendung von Dienstpararametern in Openwings. Im Gegensatz zu Openwings existieren in OSGi aber keine eigenen Klassen für Properties. Es werden also nur Strings abgespeichert. Um hier also Typinfomationen speichern zu können müsste man dies über den Umweg einer Serialisierung erreichen. Außerdem werden Properties nicht verwendet, um Benutzeroberflächen zu generieren. Sie dienen lediglich der Information über Dienstobjekte.

Hier wurden die Konfigurationsparameter als Properties verwendet. Oft ist es aber nicht erwünscht, dass alle Konfigurationsdaten zum Auffinden von Dienstobjekten benutzt werden können. Dann muss eine zusätzliche Hash-Tabelle erzeugt werden, die nur

```
1  ServiceReference[] refs = null;
2  Lamp l;
3
4  //suche Dienstobjekte
5  //bei der Suche können mehrere Dienstobjekte gefunden werden.
6  refs = context.getServiceReferences(Lamp.getClass(),
7                                      "(Location=Wohnzimmer)");
8
9  if (refs != null){ //Dienstobjekte gefunden
10     for (int i=0; i<refs.length){
11         //fordere Dienstobjekt an
12         l = (Lamp) context.getService(ref[0]);
13         //benutze das Dienstobjekt
14         l.switchOff();
15         //zeige an, dass das Dienstobjekt nicht weiter benutzt wird.
16         l=null;
17         context.releaseService(ref[0]);
18     }
19 }
```

Listing 4.23: Verwendung eines Dienstobjekts

die Properties enthält. Sollen keine Properties bei der Anmeldung benutzt werden, kann auch `null` als Parameter übergeben werden.

Nur der Aktivator eines Bundles bekommt beim Start des Bundles ein `BundleContext`-Objekt vom Rahmenwerk. Damit ein Dienstobjekt Dienstobjekte anderer Bundles benutzen kann, muss es ein `BundleContext`-Objekt besitzen. Dazu kann der Activator zum Beispiel das `BundleContext`-Objekt an das Dienstobjekt übergeben. In dieser Arbeit wird dem Konstruktor der Klasse `ServiceProvider` ein `BundleContext`-Objekt übergeben, damit das `Security`-Dienstobjekt andere Dienstobjekte benutzen kann.

Listing 4.23 zeigt die einzelnen Schritte bei der Verwendung eines Dienstobjekts. In dieser Arbeit wurde die Suche nach Dienstobjekten nur bei der Implementierung des Deployers 6.7 benutzt. Hier wird folgendes Beispiel benutzt: Alle Lampen im Wohnzimmer sollen ausgeschaltet werden. Eine Lampe implementiert die Dienstschnittstelle `Lamp` und besitzt eine Property `Location`. Die Schnittstelle `Lamp` enthält die Methode `switchOff`.

Zuerst wird mithilfe der Methode `getServiceReferences` nach den entsprechenden Dienstobjekten gesucht. Dieser Methode kann die Bezeichnung einer Dienstschnittstelle übergeben werden, die die Dienstobjekte angemeldet haben sollen. Zusätzlich kann ein Filter angegeben werden, der das Suchergebnis einschränkt.

Um alle Lampen im Wohnzimmer auszuschalten, muss nach allen Dienstobjekten gesucht werden, welche die Schnittstelle `Lamp` angemeldet haben und die Property `Location` mit dem Wert Wohnzimmer besitzen: Zur Einschränkung der Schnittstellen wird die Schnittstelle `Lamp` angegeben. Um das Suchergebnis auf Lampen im Wohnzimmer einzuschränken, wird der Filter `(Location=Wohnzimmer)` angegeben. Filter werden in Form einer LDAP-Anfrage ([Ope02], Seite 77 ff.) angegeben. In OSGi können beliebig komplexe Filter benutzt werden.

Auch in OSGi können Dienstobjekte auf Veränderungen innerhalb des Rahmenwerks reagieren. OSGi bietet dazu ein Event-Modell an. Die Verwendung der Event-Klassen

ist ähnlich der Verwendung von Event-Klassen in Rio. In Rio können Dienstobjekte aber kaum auf Veränderungen innerhalb des Rahmenwerks reagieren. In OSGi können im Gegensatz zu Rio keine eigenen Event-Klassen definiert werden. Es gibt drei Arten von Events:

ServiceEvents: Sobald ein Dienstobjekt angemeldet, abgemeldet oder die Properties eines Dienstobjekts verändert werden, versendet das Rahmenwerk ein `ServiceEvent`. Ein `ServiceEvent`-Objekt enthält eine Referenz auf ein Dienstobjekt, und eine Typinformation, die anzeigt, ob ein Dienstobjekt angemeldet, abgemeldet oder ob dessen Properties geändert wurden.

BundleEvents: Das Rahmenwerk sendet ein `BundleEvent`, wenn der Zustand eines Bundles sich innerhalb des Lebenszyklus des Bundles ändert. Dadurch können Dienstobjekte auf die Installation und Deinstallation von Bundles reagieren.

FrameworkEvents: Wenn das Rahmenwerk gestartet wurde, oder wenn ein Fehler innerhalb des Rahmenwerks aufgetreten ist, sendet es ein `FrameworkEvent`.

OSGi besitzt genauso wie Openwings keine Möglichkeit, mehrere Dienste im Sinne einer zusammenhängenden Deployment-Konfiguration zu beschreiben. Es werden einzelne Bundles beschrieben, die andere Bundles referenzieren können, damit Dienstobjekte, die das Bundle erzeugt, deren Ressourcen nutzen können. Ein Bundle ist in einer einzelnen Jar-Datei verpackt, deren Manifest-Datei die Beschreibung des Bundles enthält, die das Rahmenwerk benutzt, um das Bundle zu installieren und zu starten. Die Manifest-Datei enthält im Allgemeinen keine Informationen über zu instanzierende Dienstobjekte.

Die *Manifest-Datei* eines Bundles befindet sich innerhalb des Jar-Archivs in der Datei `META-INF/MANIFEST.MF`. Der strukturelle Aufbau einer Manifest-Datei wird in ([Ope01], Abschnitt 2.3) festgelegt. In der Manifest-Datei werden eine Menge von (Key,Value)-Paaren angegeben. Die OSGi-Spezifikation beschreibt die Bedeutung einer großen Menge von Schlüsseln. Im Folgenden sind die für die Installation und den Start eines Bundles wichtigen Schlüssel und deren Bedeutung aufgelistet:

Bundle-Name: Hier kann einem Bundle ein für Benutzer lesbarer Name gegeben werden. Es ist nicht sichergestellt, dass dieser Name eindeutig ist. Zur eindeutigen Identifizierung eines Bundes wird die Bundle-Location verwendet.

Bundle-Version: Kennzeichnet die Version eines Bundles.

Bundle-Activator: Verweist auf den Aktivator, der zum Starten und Stoppen des Bundles benutzt wird. Von einem Bundle, das keinen Aktivator besitzt, können keine Dienstobjekte erzeugt werden.

Bundle-Classpath: Jedes Bundle besitzt seinen eigenen Classpath. Dieser wird wie der üblicherweise von Java verwendete Classpath angegeben. Die einzelnen Jar-Archive, die die Pakete enthalten, die das Bundle benutzt oder exportiert, werden hier durch Kommas getrennt angegeben. Zusätzlich können in dieser Liste auch Verzeichnisse aufgeführt werden. Beginnt eine Verzeichnisangabe mit einem Punkt, liegt das Verzeichnis innerhalb des Bundles. Im Gegensatz zu Openwings können Jar-Archive anderer Bundles nicht in den Classpath aufgenommen werden. Importierte Pakete brauchen nicht im Classpath angegeben zu werden.

Import-Package: Pakete anderer Bundles müssen importiert werden, damit sie von Dienstobjekten des Bundles benutzt werden können. Durch den Import eines Pakets wird eine Beziehung zu einem anderen Bundle aufgebaut.

Export-Package: Damit Pakete von anderen Bundles importiert werden können, müssen sie exportiert werden. Üblicherweise werden Pakete exportiert, die Schnittstellen enthalten, um ein Dienstobjekt zu benutzen, oder Pakete, die Klassenbibliotheken enthalten, die bei der Implementierung eines Dienstobjekts eines anderen Bundles benötigt werden.

Bundle-NativeCode: Unter diesem Schlüssel können die verwendeten nativen Bibliotheken aufgelistet werden. OSGi unterstützt dabei die Verwendung unterschiedlicher Bibliotheken in Abhängigkeit von der Umgebung, in der das Rahmenwerk läuft.

Bundle-UpdateLocation: Ist der Ort, von dem Bundle-Updates installiert werden, wenn die Methode `update` aufgerufen wird.

DynamicImport: Pakete werden nur dann importiert, wenn sie auch gebraucht werden. Ein Problem entsteht, wenn das Paket nicht gefunden werden kann. Das Bundle muss dann so implementiert sein, dass der entsprechende Code, der das andere Paket benutzt, gar nicht aufgerufen wird, wenn die Pakete nicht verfügbar sind. Im eHome-Bereich könnte diese Art des Imports von Paketen sinnvoll sein. Zum Beispiel kann der Kunde einen Dienst in verschiedenen Ausbaustufen erwerben. Für jede dieser Ausbaustufen eine eigene Komponente zu implementieren, wäre aufwändig. Der einfachere Weg wäre, in der Konfiguration entsprechende Informationen zu hinterlegen, und dem Kunden die Möglichkeit eines späteren Updates zu geben. Dadurch wird für verschiedene Ausbaustufen nur eine Komponente benötigt.

In der Manifest-Datei dürfen beliebige weitere Schlüssel verwendet werden. Auf in der Manifest-Datei abgelegte Informationen kann mit der Methode `getHeaders`, die in der Schnittstelle `Bundle` spezifiziert ist, zugegriffen werden. `getHeaders` gibt ein `Dictionary`-Objekt, das alle Informationen der Manifest-Datei als Menge von (Key,Value)-Paaren enthält, zurück.

Alle Schlüssel innerhalb einer Manifest-Datei sind optional. Ist ein Schlüssel nicht enthalten, gilt der Eintrag als nicht spezifiziert. Eine Ausnahme bildet der Bundle-Classpath, der standardmäßig mit "." belegt wird.

In OSGi können Abhängigkeiten zwischen Bundles auf Paketebene beschrieben werden. Ein Bundle exportiert Pakete, die von anderen Bundles importiert werden können. Exportieren zwei Bundles dasselbe Paket, entscheidet das Rahmenwerk, welches benutzt wird. Durch den Import und Export von Paketen werden Beziehungen zwischen Bundles aufgebaut. In diesem Abschnitt wird der Im- und Export von Paketen und der Aufbau von Beziehungen behandelt.

Ein Bundle, das Dienstobjekte erzeugt und eine Menge von Dienstschnittstellen anmeldet, muss sicherstellen, dass diese Dienstschnittstellen für andere Bundles zugängig

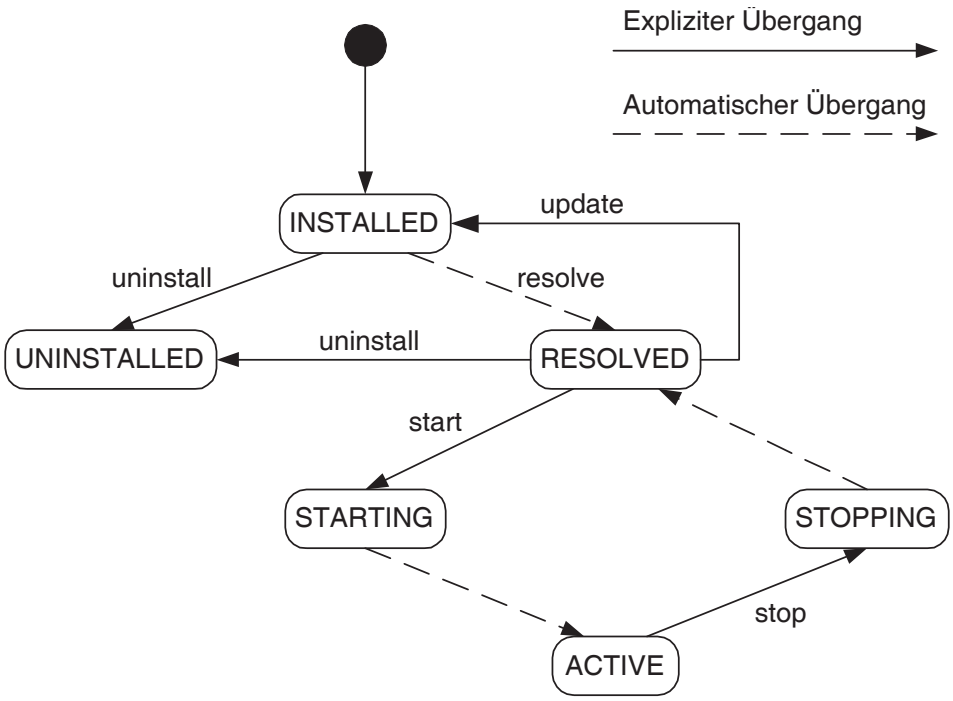

Abbildung 4.18: OSGi: Lebenszyklus eines Bundles

gemacht werden. Dazu können die Pakete, in denen die Schnittstellen enthalten sind, exportiert werden. Diese Pakete können von anderen Bundles, die die Dienstobjekte benutzen, importieren. Dabei ist es nicht möglich, eine einzelne Klasse eines Pakets zu exportieren. Deshalb wurde in dieser Arbeit die Implementierung eines Dienstobjekts in einem anderen Paket abgelegt als die Schnittstelle. Die Schnittstelle `Camera` befindet sich im Paket `ehome.scenario.osgi.devices.camera`. Bei der Angabe von zu im- oder exportierenden Bundles können auch Versionsnummern angegeben werden (siehe [Ope01]).

Das Rahmenwerk sorgt dafür, dass die exportierten Pakete eines Bundles verfügbar bleiben. Deshalb wird bei der Deinstallation eines Bundles ein exportiertes Paket solange nicht gelöscht, bis die Methode `refreshPackages` der Klasse `PackageAdmin` aufgerufen oder das Rahmenwerk neu gestartet wird.

Die OSGi-Implementierung von Prosyst führt automatisch `refreshPackages` aus, wenn ein Bundle deinstalliert wird.

Neben Schnittstellen, die zur Benutzung von Dienstobjekten exportiert werden, können Bundles auch Pakete enthalten, die bei der Implementierung von Dienstobjekten eines anderen Bundles benutzt wurden. Benötigt ein Bundle ein Paket eines anderen Bundles, kann es dieses importieren. Ein Bundle kann nur die Pakete eines anderen Bundles importieren, die von diesem exportiert werden.

In Abbildung 4.18 ist der Lebenszyklus eines OSGi-Bundles dargestellt. Der Lebens-

zyklus eines Bundles beginnt mit der Installation des Bundles und endet mit dessen Deinstallation. Im Folgenden werden die möglichen Transitionen und die damit verbundenen Zustandsänderungen beschrieben:

install: Um ein Bundle zu installieren, muss der Ort angegeben werden, von dem das Bundle heruntergeladen werden kann. Dieser Ort wird als Bundle-Location bezeichnet. Von dort wird das Jar-Archiv, in dem die Dateien des Bundles enthalten sind, heruntergeladen. Zunächst überprüft das Rahmenwerk, ob von dieser Bundle-Location schon ein Bundle installiert wurde. Existiert schon ein Bundle mit derselben Bundle-Location, wird das Bundle nicht installiert, da das Rahmenwerk davon ausgeht, dass das Bundle schon installiert wurde. Die Bundle-Location dient der eindeutigen Identifikation eines Bundles. Dann überprüft das Rahmenwerk, anhand verschiedener Kriterien, ob das Jar-Archiv ein gültiges Bundle enthält. Das Bundle wird anschließend installiert. Dabei werden die Dateien innerhalb des Archivs entpackt. Damit ist das Bundle installiert. Zur Installation eines Bundles kann die Methode `installBundle` der Schnittstelle `BundleContext` benutzt werden. Diese führt bei Bundles, die Pakete importieren, automatisch die in der Transaktion `resolve` enthaltenen Schritte durch.

uninstall: Bundles, die nicht mehr benötigt werden, können deinstalliert werden. Wurden von einem Bundle Pakete exportiert, die von anderen Bundles benutzt werden, werden diese Pakete weiterhin angeboten. Sobald das Rahmenwerk erneut gestartet wird, oder die Methode `refreshPackages` des Package Admins aufgerufen wurde, werden alle Pakete eines Bundles im Zustand `UNINSTALLED` entfernt. Dadurch kann es vorkommen, dass ein Dienst, der vor einem Neustart des Rahmenwerks noch funktionierte, nach dem Neustart nicht mehr funktioniert. Solche Probleme müssen von Administrations-Diensten behoben werden. In Prosyst wird die Methode `refreshPackages` immer aufgerufen, wenn ein Bundle deinstalliert wird. Die Methode `uninstall` der Schnittstelle `Bundle` stoppt ein Bundle, das sich nicht im Zustand `RESOLVED` oder `INSTALLED` befindet, bevor es deinstalliert wird. Diese Methode kann zur Deinstallation eines Bundles benutzt werden. Bundles, die sich einmal im Zustand `UNINSTALLED` befinden, bleiben in diesem Zustand, bis sie endgültig vom Rahmenwerk entfernt werden.

update: Eine Version eines Bundles kann während das Rahmenwerk ausgeführt wird, durch eine neuere ersetzt werden. Die neue Version muss abwärtskompatibel zur älteren Version sein. Denn das Rahmenwerk sorgt dafür, dass nur noch die neuen Versionen, der im Bundle enthaltenen Dateien, erreichbar sind. Wurden Pakete durch das Bundle exportiert, bleiben die alten Versionen der Pakete solange erhalten, bis das Rahmenwerk neu gestartet oder die Methode `refreshPackages` des Package Admins aufgerufen wurde. Die Methode `update` der Schnittstelle `Bundle` stoppt ein Bundle bevor es upgedatet wird, wenn es sich nicht im Zustand `RESOLVED` befindet.

resolve: Diese Transition wird automatisch vom Rahmenwerk ausgeführt, wenn ein Bundle installiert oder aktualisiert wurde. `resolve` versucht die in der Manifest-Datei spezifizierten Abhängigkeiten aufzulösen. Konnten alle Abhängigkeiten auf-

gelöst werden, wechselt das Bundle in den Zustand `RESOLVED`. Ausführbare Bundles können dann gestartet werden.

start: Befindet sich ein Bundle nicht im Zustand `RESOLVED`, versucht das Rahmenwerk zunächst Abhängigkeiten aufzulösen. Gelingt dies nicht, wird der Startvorgang abgebrochen. Sonst wird, falls ein Aktivator vorhanden ist, die `start`-Methode des Bundle-Aktivators ausgeführt. Während des Start-Vorgangs befindet sich das Bundle im Zustand `STARTING`. Konnte die `start`-Methode erfolgreich ausgeführt werden, wechselt das Bundle in den Zustand `ACTIVE`. Auch Bundles, die keinen Aktivator besitzen, können gestartet werden. Dann wird einfach keine `start`-Methode ausgeführt. Um ein Bundle zu starten, kann die Methode `start` der Schnittstelle `Bundle` benutzt werden.

stop: Beim Stoppen eines Bundles wird zunächst die Methode `stop` des Aktivators ausgeführt, insofern das Bundle einen besitzt. Dann werden automatisch alle angemeldeten Dienstobjekte des Bundles abgemeldet und alle benutzten Referenzen auf Dienstobjekte anderer Bundles freigegeben. Außerdem werden alle Listener für Events abgemeldet.

Der *Configuration Admin* stellt eine Datenbank zur Verfügung, in die `Configuration`-Objekte abgelegt werden können. Der Configuration Admin selbst ist ein Dienstobjekt, das die Schnittstelle `ConfigurationAdmin` implementiert. Mithilfe dieser Schnittstelle können `Configuration`-Objekte erzeugt, aktualisiert und gelöscht werden. Darüber hinaus kann über diese Schnittstelle nach `Configuration`-Objekten gesucht werden. Innerhalb eines `Configuration`-Objekts ist eine Menge von (Key,Value)-Paaren enthalten. Es gibt drei Schlüssel, deren Bedeutung festgelegt ist:

service.pid: ID, mit deren Hilfe ein `Configuration`-Objekt identifiziert werden kann. Sie dient auch dazu, ein `Configuration`-Objekt einem Managed Services zuzuordnen.

service.factoryPid: `Configuration`-Objekte, die diesen Schlüssel besitzen, werden keinem Managed Services zugeordnet, sondern einer Managed Service Factory. Auch diese `Configuration`-Objekte müssen eine `service.pid` besitzen.

service.bundleLocation: Mithilfe dieses Schlüssels wird ein `Configuration`-Objekt an ein Bundle gebunden.

Die `service.pid` und die `service.factoryPid` können, nachdem das `Configuration`-Objekt erzeugt wurde, nicht mehr verändert werden.

Der Configuration Admin selbst kann keine Dienstobjekte erzeugen. Um Dienstobjekte zu initialisieren, wird zunächst ein einzelnes Dienstobjekt erzeugt, das Konfigurationsdaten vom Configuration Admin erhält. Mit diesen Konfigurationsdaten kann es Dienstobjekte erzeugen und initialisieren. Um ein einzelnes Dienstobjekt zu erzeugen, muss das Dienstobjekt, das ein `Configuration`-Objekt erhalten soll, die Schnittstelle `ManagedService` implementieren. Soll ein Dienstobjekt mehrere `Configuration`-Objekte erhalten, um mehrere Dienstobjekte zu initialisieren, muss es die Schnittstelle `ManagedServiceFactory` implementieren.

Managed Service Factory können analog zu Managed Services verwendet werden. Eine Managed Service Factory wird verwendet, wenn mehrere `Configuration`-Objekte zur Initialisierung mehrerer Dienstobjekte benutzt werden sollen. Das Dienstobjekt, das die Instanzierung der Dienstobjekte übernimmt, muss die Schnittstelle `ManagedServiceFactory` implementieren. Bei der Anmeldung des Dienstobjekts, wird die Schnittstelle `ManagedServiceFactory` und eine `service.factoryPid` angegeben.

Um eine Managed Service Factory mit `Configuration`-Objekten zu versorgen, sucht der Configuration Admin nach `Configuration`-Objekten mit der jeweiligen `service.factoryPid`. Für jedes gefundene Dienstobjekt ruft der Configuration Admin die Methode `update` auf. Anders als bei Managed Services, wird dieser Methode nicht nur das `Configuration`-Objekt übergeben, sondern auch eine `service.pid`.

Sobald ein `Configuration`-Objekt, das eine `service.factoryPid` besitzt, angelegt, aktualisiert oder gelöscht wird, sucht der Configuration Admin nach einem `ManagedServiceFactory`-Dienstobjekt mit der entsprechenden `service.factoryPid`. Wurde es angelegt oder aktualisiert, wird die Methode `update` aufgerufen. Dabei werden das `Configuration`-Objekt und die `service.pid` als Parameter übergeben. Sonst wird die Methode `deleted` mit der `service.pid` aufgerufen.

Der *Device Access Service (DAS)* unterstützt die Einbindung von Geräten in das OSGi-Rahmenwerk. Dabei unterstützt dieser Dienst mehrere Verbindungstechnologien, über die er den Anschluss eines neuen Geräts automatisch erkennt und versucht, ein Dienstobjekt zu erzeugen, um ein solches Gerät anzusteuern.

Für jede Verbindungstechnologie wird ein Basistreiber bereitgestellt. Die Basistreiber werden vom Device-Manager benutzt, um den Anschluss von Geräten zu erkennen und Informationen über die Geräte zu erhalten, die bei der Suche nach Treibern verwendet werden können. Wird ein neues Gerät vom Device-Manager erkannt, erzeugt dieser ein `Device`-Objekt und meldet es im Rahmenwerk an. Dieses Objekt repräsentiert das Gerät innerhalb des Rahmenwerks, es kann aber noch nicht verwendet werden. Die Informationen, die das Gerät beschreiben, sind in den Properties, die bei der Anmeldung des Dienstobjekts hinzugefügt werden, enthalten. Für USB-Geräte sind dies u.a. die Herstellernummer (Vendor-ID) und eine Seriennummer (Product-ID), die zusammen die eindeutige Identifizierung eines USB-Geräts ermöglichen.

Um ein angeschlossenes Gerät benutzen zu können, müssen Dienstobjekte erzeugt werden. Dazu dienen so genannte Treiber. Als Treiber werden im OSGi-Kontext Dienstobjekte bezeichnet, die die Schnittstelle `Driver` implementieren. Jeder Treiber besitzt eine ID, die das Auffinden des Treibers ermöglicht. Treiber sind für die Erzeugung von Dienstobjekten, die Geräte ansteuern, verantwortlich. Jedes Bundle, das zur Ansteuerung von einem Gerät entwickelt wurde, muss einen Treiber im Rahmenwerk anmelden. Mithilfe der Methode `match` entscheidet der Device-Manager, welches `Driver`-Dienstobjekt er mit der Erzeugung und Anmeldung eines Dienstobjekts, das das Gerät ansteuert, beauftragt. Dazu ruft der Device-Manager die `match`-Methode aller Treiber auf. Der Treiber, der den größten Wert zurück gibt, wird ausgewählt.

Hat der Device-Manager einen Treiber ausgewählt, ruft er die Methode `attach` des Treibers auf. Diese Methode kann entweder ein Dienstobjekt erzeugen und `null` zurück

```
1  package ehome.scenario.osgi.devices.camera;
2
3  import javax.swing.ImageIcon;
4
5  public interface Camera{
6      public String getName();
7      public ImageIcon getImage();
8      public String getLocation();
9      public void setLocation(String loc);
10     public String test();
11
12 }
```

Listing 4.24: Synchrone Dienstschnittstelle Camera

geben, oder die ID eines besseren Treibers, den der Device-Manager stattdessen benutzen soll.

Unschön ist hier, dass der DAS eine Zahl benutzt, die beschreiben soll, wie gut ein Treiber zu einem Gerät passt. Es ist aber nicht geklärt, was es bedeutet, wenn die Zahl bei einem Treiber größer ist, als bei einem anderen.

Um Treiber benutzen zu können, die von noch nicht installierten Bundles bereitgestellt werden, kann ein *Driver-Locator* benutzt werden. Ein Driver-Locator ist ein Dienstobjekt, dass für bestimmte Geräte Bundles kennt, die Treiber für diese Geräte bereitstellen.

Auch unter Verwendung des DAS müssen weiterhin Bundles entwickelt werden, um Geräte anzusprechen. Der DAS bietet eine weitere Möglichkeit, um Dienstobjekte zu erzeugen. Der Treiber übernimmt die Rolle der Factory als Erzeuger von Dienstobjekten. Der Auslöser für die Erzeugung eines Dienstobjekts ist hier der Anschluss eines Geräts. Ein Problem bei der Nutzung des Device-Managers ist, dass die von den Treibern erzeugten Dienstobjekte weitere Konfigurationsinformationen benötigen. So liefert eine an eine USB-Schnittstelle angeschlossene Kamera im Allgemeinen keine Informationen über die Lage der Kamera. Auch hier hilft es, diese fehlenden Daten über den Configuration-Admin zur Verfügung zu stellen.

4.3.2 Implementierung

Für die Implementierung des Sicherheits-Szenarios wurden die fünf Komponenten an das OSGi-Rahmenwerk angepasst. Für die Erzeugung von Dienstobjekten wurden eine Klasse `Activator` und `Factory` bzw. `Manager` hinzugefügt. In Abbildung 4.19 sind die Klassen dargestellt, die für die Implementierung des Sicherheits-Dienstes benutzt wurden.

In Listing 4.24 ist die synchrone Dienstschnittstelle `Camera` angegeben, die im Bundle `CameraProvider` enthalten ist.

In dieser Arbeit wurde die dynamische Instanzierung benutzt. Deshalb wurde der Code zur Anmeldung der für den Sicherheits-Dienst implementierten Dienstobjekte in der `Factory`- bzw. `Manager`-Klasse des jeweiligen Bundles implementiert.

Um ein `Camera`-Dienstobjekt anzumelden, wird der in Listing 4.26 aufgeführte Code verwendet. In dieser Arbeit wurde dieser Code in einer Factory verwendet, um beim Hinzufügen von Konfigurationsdaten Dienstobjekte zu erzeugen. Aber auch der Aktiva-

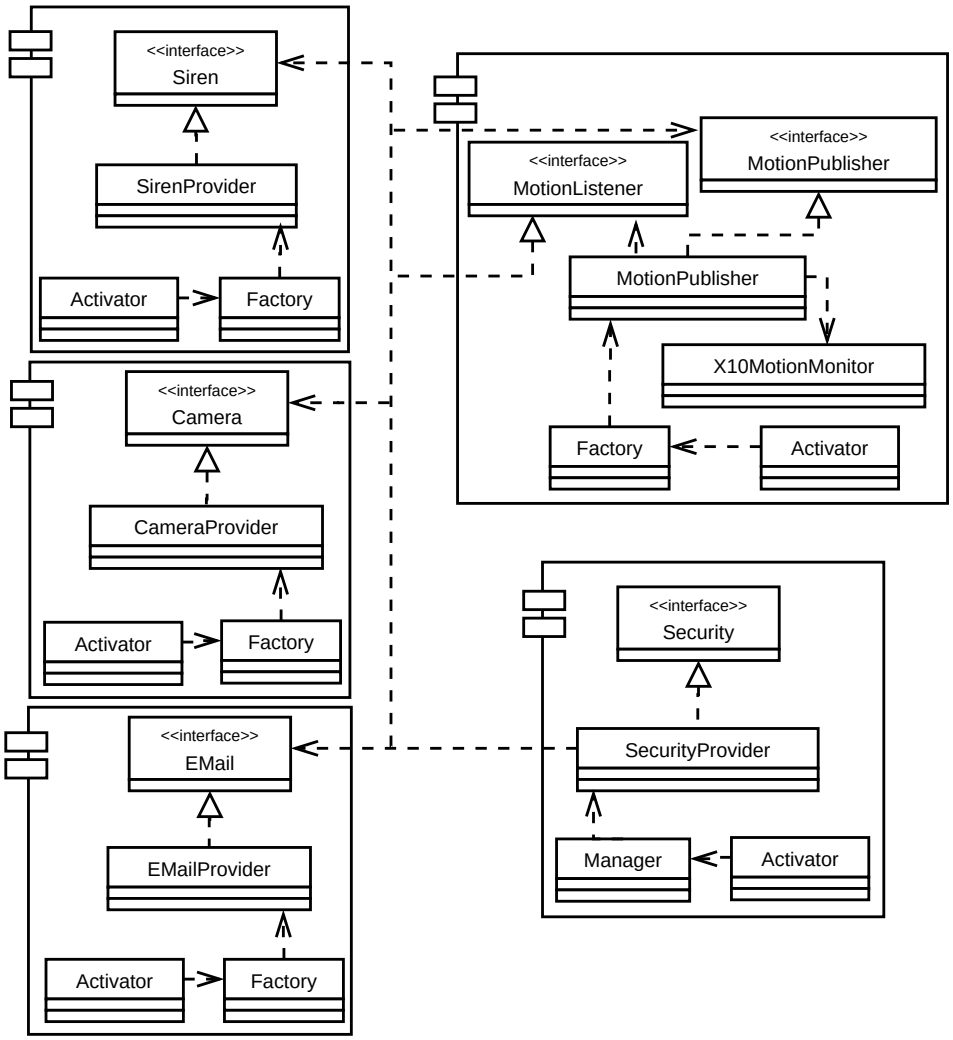

Abbildung 4.19: Komponentenübersicht des vereinfachten Sicherheits-Dienstes in OSGi

```
1  public CameraProvider(Dictionary config) {
2      determineParameters(config);
3      log("Service started.");
4  }
5
6  public void determineParameters(Dictionary config) {
7
8      /*Parametername: Name; has to be unique; required*/
9      servicename = (String) config.get("service.name");
10     log("Name: " + servicename);
11
12     /*Parametername: Location; Standardvalue: null; optional*/
13     location = (String) config.get("Location");
14     log("Location: " + location);
15
16     /*Parametername: URL; required*/
17     url = (String) config.get("URL");
18     log("URL: " + url);
19 }
```

Listing 4.25: Erzeugung eines `Camera`-Dienstobjekts

```
1  //Erstellung der Konfiguration
2  Hashtable config = new Hashtable();
3  config.put("service.name","Camera1");
4  config.put("Location","Bad");
5  config.put("XML","CAMERA1");
6
7  //erzeugen des Dienstobjekts
8  Camera service = new CameraProvider(config);
9
10 //anmelden des Dienstobjekts
11 ServiceRegistration serviceReg =context.
12     registerService(Camera.class.getName(), service, config);
```

Listing 4.26: Anmeldung eines `Camera`-Dienstobjekts

tor selbst kann Dienstobjekte erzeugen und anmelden. Jedes Objekt kann mithilfe eines `BundleContext`-Objekts Dienstobjekte anmelden.

Zunächst wird ein Konfigurationsobjekt erzeugt (Zeile 1-5). Dann wird mit dem Konfigurationsobjekt ein Dienstobjekt erzeugt (Zeile 7-8). Der Konstruktor der Klasse `CameraProvider` verwendet die Konfigurationsdaten zur Initialisierung des Dienstobjekts (Listing 4.25). Im Anschluss wird das Dienstobjekt angemeldet (Zeile10-12). Dazu wird die Methode `registerService` mit einer Dienstschnittstelle und dem Konfigurationsobjekt als Parameter aufgerufen. Soll das Dienstobjekt mehrere Dienstschnittstellen anmelden, kann der Methode auch ein Array von Dienstschnittstellen übergeben werden.

`Motion`-Dienstobjekte benutzen die Schnittstelle `MotionListener`, um Ereignisse zu versenden (Listing 4.27). Jedes Dienstobjekt, das Ereignisse von einem `Motion`-Dienstobjekt empfangen will, muss diese Schnittstelle implementieren, um auf Ereignisse zu reagieren. Da OSGi keinen zentralen Ereignisdienst besitzt, müssen sich interessierte Dienstobjekte bei den `Motion`-Dienstobjekten selbst anmelden.

```
1  protected void  motionDetected() {
2
3      long timestamp = System.currentTimeMillis();
4
5      //Sende Ereignis an alle Listener
6      Enumeration enum = listeners.elements();
7      MotionListener ml = null;
8
9      while(enum.hasMoreElements()){
10         try{
11             ml = (MotionListener) enum.nextElement();
12             ml.motionDetected(servicename,location,timestamp);
13         }catch(Exception e){
14             e.printStackTrace();
15         }
16     }
17 }
```

Listing 4.27: Versenden von Ereignissen

```
1  Hashtable listeners;
2
3  public void addListener(String name, MotionListener listener) {
4      if (name == null || listener == null) return;
5      if (listeners.get(name) != null) return;
6      listeners.put(name,listener);
7  }
8
9  public void removeListener(String name) {
10     listeners.remove(name);
11 }
```

Listing 4.28: Implementierung der Schnittstelle `MotionEventPublisher`

Jedes `Motion`-Dienstobjekt implementiert die Dienstschnittstelle `MotionEventPublisher` (Listing 4.28), um anderen Dienstobjekten die An- und Abmeldung zu ermöglichen. Zur Anmeldung wird die Methode `addListener` benutzt, und zur Abmeldung die Methode `removeListener`. Der Parameter `name` wird benutzt, um angemeldete Dienstobjekte zu unterscheiden. Bei der Anmeldung wird zusätzlich das Objekt selbst als Parameter übergeben, damit das `Motion`-Dienstobjekt Ereignisse verschicken kann. Bei der Versendung des Ereignisses wird die Methode `motionDetected` bei allen angemeldeten Dienstobjekten nacheinander ausgeführt (Listing 4.27).

Ein `Security`-Dienstobjekt meldet sich bei allen `Motion`-Dienstobjekten an, um Ereignisse zu erhalten. Wird ein `Motion`-Dienstobjekt hinzugefügt, meldet es sich auch dort an. Wenn eines entfernt wird, meldet es sich wieder ab (Listing 4.29). Wird das `Security`-Dienstobjekt beendet, meldet es sich bei allen `Motion`-Dienstobjekten ab.

Um ein `Security`-Dienstobjekt zu erzeugen, wurde die Klasse `Manager` implementiert. Diese implementiert die Schnittstelle `ManagedService`. Der Activator des `SecurityProvider`-Bundles legt zunächst eine neue Hash-Tabelle an, und legt in dieser die Property `service.pid` ab (Listing 4.31). Dann wird ein Objekt der Klasse `Manager` erzeugt und als Dienstobjekt angemeldet.

```
1  public void serviceChanged(ServiceEvent event) {
2      int type = event.getType();
3      ServiceReference serviceRef = event.getServiceReference();
4      if (type == ServiceEvent.REGISTERED){
5          serviceAdded(serviceRef);
6      }else if (type == ServiceEvent.UNREGISTERING){
7          serviceRemoved(serviceRef);
8          context.ungetService(serviceRef);
9      }else if (type == ServiceEvent.MODIFIED){
10             //no reaction
11     }
12 }
```

Listing 4.29: Reaktion auf `ServiceEvents` des `Security`-Dienstobjekts

```
1  Manifest-Version: Version 1.0
2  Bundle-ContactAddress: phd.ehomeconfig@mail.ulno.net
3  Import-Package: org.osgi.service.cm,
4   ehome.services.osgi.runtimeinstancer,
5   ehome.scenario.osgi.devices.siren,
6   ehome.scenario.osgi.devices.camera,
7   ehome.scenario.osgi.devices.email,
8   ehome.scenario.osgi.sensors.motion
9  Export-Package: ehome.scenario.osgi.logic.security
10 Bundle-Version: 0.0.1
11 Bundle-Activator: ehome.scenario.osgi.logic.security.impl.Activator
12 Bundle-Name: SecurityService
13 Bundle-Vendor: RWTH
14 Bundle-DocURL: http://ehomeconfig.ulno.net
15 Bundle-Description: SecurityService
16 Bundle-Category: eHome
```

Listing 4.30: Manifest-Datei des Bundles Security

```
1  private static final String SERVICE_PID =
2          "ehome.scenario.osgi.logic.security";
3
4  public void start(BundleContext context) throws BundleException {
5
6      Hashtable config = new Hashtable();
7      config.put(Constants.SERVICE_PID, SERVICE_PID);
8
9      service = new Manager();
10     try {
11         service.start(context);
12     } catch (Exception e) {
13         e.printStackTrace();
14         return;
15     }
16
17     //Register Manager
18     serviceReg = context.registerService(
19         ManagedService.class.getName(), service, config);
20 }
```

Listing 4.31: Erzeugung eines ManagedService-Dienstobjekts

```
1  public synchronized void updated(java.util.Dictionary properties){
2
3      if (properties==null){ //keine Standardkonfiguration
4          return;
5      }
6
7      if (service==null){ //Erzeuge Dienstobjekt
8          service = new SecurityProvider(context);
9          service.determineParameters(properties);
10
11         serviceReg = context.registerService(
12             Security.class.getName(), service, properties);
13
14         service.activate();
15         log("new serviceobject: Security");
16     }else{ //Aktualisiere Konfiguration
17         service.deactivate();
18         service.determineParameters(properties);
19         serviceReg.setProperties(properties);
20         service.activate();
21     }
22 }
```

Listing 4.32: Erzeugung und Aktualisierung des Security-Dienstobjekts

Die Klasse `Manager` implementiert die Schnittstelle `ManagedService`. In der Methode `update` (Listing 4.32) wird zunächst überprüft, ob `null` übergeben wurde, also keine Konfigurationsdaten vorhanden sind. Dann kann das `Security`-Dienstobjekt nicht erzeugt werden, da es keine Standardkonfiguration besitzt (Zeile 3). Wenn ein `Configuration`-Objekt übergeben wurde, wird überprüft, ob das `Security`-Dienstobjekt schon erzeugt wurde. Wurde schon ein `Security`-Dienstobjekt erzeugt, wird dessen Konfiguration aktualisiert (Zeile 17-20). Sonst wird ein neues `Security`-Dienstobjekt erzeugt, initialisiert und angemeldet (Zeile 8-15).

In dieser Arbeit wurde für die Erzeugung von `Camera`-, `Motion`-, `EMail`- und `Siren`-Dienstobjekten jeweils eine Managed Service Factory implementiert. Der Activator des CameraProvider-Bundles meldet ein Factory-Dienstobjekt an (Listing 4.33). Im Folgenden wird die Implementierung der Klasse `Factory` des Bundles `CameraProvider` vorgestellt. Die Implementierung der anderen `Factories` ist weitgehend identisch.

In der Klasse `Factory` werden die erzeugten Dienstobjekte in einer Hash-Tabelle verwaltet. Als Schlüssel wird die `service.pid` des `Configuration`-Objekts verwendet, das die Konfigurationsdaten für das Dienstobjekt enthält. In der Methode `update` (Listing 4.35, Zeile 16-36) wird zunächst überprüft, ob das Dienstobjekt schon angemeldet wurde (Zeile 23). Wurde es schon angemeldet, wird die Konfiguration aktualisiert (Zeile 31-34), sonst wird ein neues Dienstobjekt angelegt und angemeldet. Zusätzlich wird es in der Hash-Tabelle abgelegt (Zeile 24-29).

In der Methode `delete` (Zeile 38-42) wird die Methode `stopService` (Zeile 44-51) aufgerufen, die das Dienstobjekt stoppt und aus der Hash-Tabelle entfernt.

```
1  private static final String FACTORY_ID =
2       "ehome.scenario.osgi.devices.camera.factory";
3  private Factory factory = null;
4
5  public void start(BundleContext context) throws BundleException{
6
7      Hashtable config = new Hashtable();
8      config.put(Constants.SERVICE_PID, FACTORY_ID);
9
10     /* Generate new Factory */
11     factory = new Factory();
12     try {
13         factory.start(context);
14     } catch (Exception e) {
15         e.printStackTrace();
16         return;
17     }
18     /* Register Factory */
19     context.registerService(
20             ManagedServiceFactory.class.getName(), factory, config
21             );
22 }
23 public void stop(BundleContext context) throws BundleException{
24     try {
25         factory.stop(context);
26         factory = null;
27     } catch (Exception e) {
28         e.printStackTrace();
29     }
30 }
```

Listing 4.33: Anmeldung der Factory innerhalb des Activators

```
1  public class Factory implements ManagedServiceFactory{
2
3      //diese Methoden werden vom Activator benutzt
4
5      public void start( BundleContext context ) throws Exception {
6          this.context=context;
7      }
8
9      public void stop(BundleContext context) throws Exception {
10         this.context=context;
11         String servicepid;
12
13         Enumeration pids = services.keys();
14         while (pids.hasMoreElements()){
15             servicepid = (String) pids.nextElement();
16             stopservice(servicepid);
17         }
18     }
19 }
```

Listing 4.34: An- und Abmeldung der Factory

```
1  public class Factory implements ManagedServiceFactory{
2      private static final String FACTORY_ID =
3              "ehome.scenario.osgi.devices.camera.factory";
4      private static final String FACTORY_NAME =
5              "CameraProviderFactory";
6
7      private Hashtable services = new Hashtable(),
8                        servicesRegs = new Hashtable();
9
10     private BundleContext context;
11
12     public java.lang.String getName(){
13         return FACTORY_NAME;
14     }
15
16     public void updated(String pid, Dictionary properties){
17         CameraProvider service = null;
18         ServiceRegistration serviceReg = null;
19
20         //get serviceobject
21         service = (CameraProvider) services.get(pid);
22
23         if (service==null){ //Generate serviceobject
24             service = new CameraProvider(properties);
25             serviceReg = context.registerService(
26                 Camera.class.getName(), service, properties
27                 );
28             services.put(pid,service);
29             servicesRegs.put(pid,serviceReg);
30         }else{ //Update serviceobject
31             service.determineParameters(properties);
32             serviceReg =
33                 (ServiceRegistration) servicesRegs.get(pid);
34             serviceReg.setProperties(properties);
35         }
36     }
37
38     public void deleted(String pid){
39         stopservice(pid);
40         services.remove(pid);
41         servicesRegs.remove(pid);
42     }
43
44     protected void stopservice(String pid){
45         CameraProvider service =
46             (CameraProvider) services.get(pid);
47         ServiceRegistration serviceReg =
48             (ServiceRegistration) servicesRegs.get(pid);
49         serviceReg.unregister();
50         service.shutdown();
51     }
52 }
```

Listing 4.35: Factory des `CameraProvider`-Bundles

4.3.3 Zusammenfassung

Der auffälligste Unterschied von OSGi zu Rio und Openwings ist, dass OSGi kein verteiltes Rahmenwerk ist. D.h. Dienstobjekte, die auf unterschiedlichen OSGi-Plattformen ausgeführt werden, können nicht direkt über Dienste des Rahmenwerks miteinander kommunizieren. Es gibt allerdings mittlerweile Bundles, die die Kommunikation über Webservices und SOAP [EF03, New02] erlauben. Auch Openwings (siehe Abschnitt 4.2 auf Seite 75) würde sich hier als Verschaltungsplattform anbieten. Auch JXTA [Gon01] wird verwendet, um verschiedene OSGi-Plattformen zu verschalten.

Der in dieser Arbeit implementierte Prototyp benutzt die Schnittstellen `Bundle` und `BundleContext` zur Steuerung des Lebenszyklus' eines Bundles. Zwar besitzt Openwings einen eigenen Dienst zur Installation von Komponenten, dieser ist aber schwieriger zu benutzen.

In OSGi kann der Aktivator eines Bundles Dienstobjekte anmelden und sie mit Konfigurationsdaten versorgen. D.h. durch den Start eines Bundles können Dienstobjekte erzeugt und initialisiert werden.

OSGi unterstützt nicht direkt die Übergabe von Parametern beim Start eines Bundles, und ein Bundle kann auch nicht mehrfach gestartet werden. Deshalb können in OSGi nicht auf dieselbe Art und Weise Dienstobjekte erzeugt werden, wie in Openwings. OSGi bietet aber mit dem Configuration Admin ein komfortableres und leistungsfähigeres Konzept zur Instanzierung von Dienstobjekten an.

OSGi unterstützt als einziges der untersuchten Rahmenwerke das Ablegen von Konfigurationsdaten in einer Datenbank. Dadurch sind die Konfigurationsinformationen leichter auffindbar und es entsteht nicht dasselbe Problem, wie bei den von Openwings benutzten Policies. Der von OSGi angebotene Configuration Admin versorgt Dienstobjekte mit Konfigurationsdaten, die diese benutzen können, um Dienstobjekte zu initialisieren. Werden Konfigurationsdaten aktualisiert oder gelöscht, werden diese Dienstobjekte benachrichtigt und können entsprechend reagieren. OSGi besitzt also hier eine Dynamik, die Rio und Openwings fehlt. Der Deployer könnte also den Configuration Admin benutzen, um Dienstobjekte anzulegen und die Konfiguration eines Dienstobjekts zu aktualisieren. Die vom Configuration Admin verwalteten Konfigurationsdaten sind dann jeweils einem Dienstobjekt zuzuordnen. Um Konfigurationsdaten abzulegen, die von mehreren Dienstobjekten benutzt werden können, wie zum Beispiel die Schriftgröße in der Informationen dargestellt werden sollen, bietet OSGi die Möglichkeit Preferences anzulegen.

Die erste Version des Deployers wurde auch nach dieser Vorgehensweise implementiert. Allerdings war die Übergabe der Konfigurationsdaten, wegen ihrer Speicherung als Strings in einem objektorientierten Kontext zu fehleranfällig, weshalb in der neuen Deployervariante ein eigenes Bundle zum Zugriff auf die Konfigurationsdaten eingesetzt wird. Dennoch wurde die Idee einer gemeinsamen Datenbank aller Dienste dabei wiederverwendet.

Um auf Dienstobjekte innerhalb des Rahmenwerks zuzugreifen oder um Bundles zu installieren, deinstallieren und zu aktualisieren, wird ein `BundleContext`-Objekt benötigt. Es gibt keinen OSGi-Dienst, der Software, die nicht innerhalb des OSGi-Rahmenwerks ausgeführt wird, den direkten Zugriff auf das Rahmenwerk erlaubt.

OSGi benutzt genauso wie Rio und Openwings Java als Spezifikationssprache für Schnittstellen. Allerdings kennt OSGi nur synchrone Dienstschnittstellen. OSGi be-

Datei (-typ)	Codezeilen
*.java-Dateien	4620
davon Gerätetreiber	3794
davon Erweiterungs-Dienste	826
Manifest-Dateien *.MF	356
build.xml	289
Summe	5265

Tabelle 4.3: Geschriebene Quellcodezeilen für den vereinfachten Sicherheitsdienst in OSGi.

sitzt kein eigenes Konzept zur asynchronen Kommunikation. Zur Benachrichtigung von Dienstobjekten über Veränderungen innerhalb des Rahmenwerks werden zwar Event-Klassen benutzt, die Implementierung eigener Event-Klassen wird aber nicht unterstützt.

Tabelle 4.3 zeigt die geschriebenen Quellcodezeilen für OSGi. Die Werte ähneln den Beobachtungen von Rio (siehe Tabelle auf Seite 74) und sind also auch bei den Verbindungsbeschreibungen nicht höher als erwartet.

OSGi ist ein sehr schmales und mittlerweile weit verbreitetes Komponentenrahmenwerk mit einer breiten Community-Unterstützung und hat als einziges der vorgestellten Rahmenwerke auch außerhalb dieser Arbeit Anwendungen im eHome-Bereich (siehe inHaus [inH05]). Die Entwicklung der eHome-Dienste gestaltet sich deshalb auch im OSGi-Kontext einfacher als bei Rio und Openwings.

4.4 Vergleich

In Tabelle 4.4 sind die Zahlen der geschriebenen Quellcodezeilen zur Realisierung des vereinfachten Sicherheitsdienstes in den gerade besprochenen Rahmenwerken aus den Tabellen 4.1, 4.2 und 4.3 in einer Tabelle zusammengeführt. Unter Strukturdateien fallen hier Manifest-Dateien, `build.xml`-Dateien, `owbuild.xml`-Dateien, Policies, die Datei `build.properties` und Operational-Strings. Das sind somit die Dateien, die helfen, die Realisierungen der Komponenten zusammenzukleben, deswegen wird hier auch das Stichwort *Glue* zur Bezeichnung benutzt. Es sollte berücksichtigt werden, dass die `owbuild.xml` und `build.xml`-Dateien alle sehr ähnlich sind, so dass der Entwicklungsaufwand geringer einzuschätzen ist, als die hohe Zahl suggeriert, da `owbuild.xml` und `build.xml`-Dateien – wie in Tabelle 4.2 zu sehen – alleine ungefähr 6500 Zeilen Code ausmachen. Dennoch ist die Entwicklung der Policies mangels geeigneter Werkzeugunterstützung für Openwings sehr aufwändig und muss so als Nachteil bei einer Auswahl aus den vorgestellten Rahmenwerken gewertet werden.

Alle vorgestellten Rahmenwerke bieten die Spezifikation von Javaschnittstellen zur synchronen Kommunikation an. Auch bieten alle Mechanismen zur asynchronen Kommunikation wie Event-Kommunikation bzw. asynchrone Schnittstellen an.

Die Operational-Strings, insbesondere die Beschreibung der dort enthaltenen JSBs, von Rio stellen eine geeignete Methode zur Deployment-Beschreibung für die Parametrisierung von Komponenten dar. Sie sind, wie in der Tabelle 4.4 zu sehen ist, viel kürzer

Datei (-typ)	Rio	Openwings	OSGi
*.java-Dateien	3891	4777	4620
davon Gerätetreiber	3226	3997	3794
davon Erweiterungs-Dienste	665	780	826
Strukturdateien/Glue	921	10611	654
Summe	4812	15388	5265

Tabelle 4.4: Geschriebene Quellcodezeilen für den vereinfachten Sicherheitsdienst in unterschiedlichen Rahmenwerken.

als Policies bei Openwings. Allerdings drücken die Policies in Openwings noch mehr die Beziehungen bzw. Abhängigkeiten einzelner Komponenten untereinander aus. Die Manifest-Dateien von OSGi spezifizieren Schnittstellen und Import-Export-Beziehungen bzw. Abhängigkeiten. Sie eignen sich also zur Abhängigkeitsbeschreibung in der Deploymentkonfiguration.

Nur OSGi bietet Rekonfiguration. Rio und Openwings erfordern Neuinstallation der Komponenten, besitzen also nicht die erforderliche Dynamik für eHome-Systeme. Rio eignet sich besser für verteilte Systeme, die einen Lastausgleich benötigen und viele gleiche Dienstobjekte verwalten müssen, als für den Einsatz in eHome-Systemen. Allerdings hat die Konfigurierung von OSGi-Komponenten mittels des Configuration Admin den Nachteil, dass sie nur Strings als Daten speichert und somit keine strukturierten Datentypen erlaubt. Deshalb wird dieses Konzept für die endgültige Realisierung einer eHome-Werkzeugsuite nicht verwendet und in Kapitel 6 neu entwickelt.

Die Entwicklung eines Deployers wird in Rio und Openwings mit mehr Aufwand als in OSGi verbunden sein. Sowohl Rio als auch Openwings sind ungeeignet für kleine Java-Maschinen. OSGi ist klein genug, um auf solch eingeschränkter Hardware zu arbeiten.

Openwings bietet sich zur Verschaltung unterschiedlicher Rahmenwerke an. Es könnte sich evtl. für Verschaltung mehrerer OSGi-getriebener Plattformen verwenden lassen.

OSGi ist im Gegensatz zu Rio und Openwings kein Rahmenwerk für verteilte Systeme. Es ist aber dafür klein, kompakt und sehr verbreitet. Die Entscheidung in dieser Arbeit für die Nutzung von OSGi wurde im Jahr 2003 gefällt. Seine Verbreitung hat seit der Entscheidung sehr zugenommen. OSGi ist heute ein sehr beliebtes lokales Komponentenrahmenwerk. So gibt es viele Projekte im Entwicklungsbereich, die dieses Rahmenwerk einsetzen (siehe Projekte wie Eclipse [Ecl03], eDOBS [Uni05], Fujaba 5 [Zün99]), es läuft auf Maschinenadaptern von Kühlschränken und einigen Waschmaschinen (dies beruht auf mündlichen Informationen von Mitarbeitern aus [inH05]), aber auch Standardanwendungen wie sip-communicator [IPB+06] setzen es ein. Obwohl es ein Einzelplatz-Rahmenwerk ist, kann es leicht durch Openwings oder andere Techniken wie Webservices [New02], JXTA [Gon01] oder wie hier in der Arbeit bereits geschehen CoObRA [Sch03] um Kommunikation in verteilten Systemen erweitert werden.

Wegen leichter Entwicklung, geringem Kodierungsaufwand, hoher Verbreitung und einzigem System mit echtem Bezug zu eHome-Systemen wird für die weiteren Untersuchungen dieser Arbeit OSGi als Rahmenwerk gewählt. Natürlich fließen die Erkenntnisse über zu verwendende Konfigurationsdaten aus Rio und Openwings mit in das neu zu er-

stellende Konfigurationsbeschreibungsmodell (eHome-Modell, siehe Abschnitt 5.2) im nächsten Kapitel ein.

Kapitel 5

Generative Konfigurierung

Wie bereits im vorangegangen Kapitel beschrieben, ist die manuelle Erstellung einer Konfigurationsbeschreibungsdatei noch mit einem erheblichen Aufwand verbunden. Auch die Anbindung der Kommunikationsschnittstellen im Rahmenwerk stellt einen nicht zu vernachlässigenden Entwicklungsaufwand dar. Im zweiten Schritt der Vereinfachung des Entwicklungsprozesses von eHome-Systemen wird ein großer Teil der Konfigurationsbeschreibung automatisch erzeugt also generiert.[1] Dafür müssen die Dienstkomponenten in einer bestimmten Art und Weise implementiert werden. Dieses Kapitel beschreibt die Konzepte, die für diese Erstellung der Konfigurierungsinformationen wichtig sind. Diese Konzepte werden hier wegen der Generierung der Konfigurationsbeschreibung und den Parallelen zur generativen Programmierung (siehe auch [CE99]) unter dem Begriff "Generative Konfigurierung" zusammengefasst. Das folgende Kapitel (Kapitel 6 auf Seite 145) beschreibt die Implementierung.

5.1 eHome-SCD-Prozess

Neben der Komponentenentwicklung im Bereich der eHome-Systeme beschäftigt sich die Arbeitsgruppe für integrierte eHome-Systeme an der RWTH-Aachen, in der auch diese Arbeit entstand, auch mit dem damit verbundenen Geschäftsprozess. Eines der Hauptprobleme, welches dabei gelöst werden soll, ist die Etablierung von eHome-Systeme in einem Massenmarkt. Neben der Aufarbeitung des eigentlichen Geschäftsprozesses in [Kir05], muss auch der Entwicklungsprozess neu formuliert und angepasst werden. Dafür wird in einen sich wiederholenden Entwicklungsprozess ein sogenannter Spezifizierungs-, Konfigurierungs- und Deployment-Prozess (SCD-Prozess) eingeführt, der erheblich leichter zu handhaben ist, als der gesamte Entwicklungsprozess und somit einen kundenorientierten Prozess ermöglicht (siehe Abbildung 5.1).

Der erste Schritt zur Reduktion der repetitiven Entwicklung im klassischen Entwicklungsprozess ist die Bewegung des Implementationsanteils weg von einer dienstbezogenen Entwicklung hin zu einer a-priori Entwicklung wiederverwendbarer Komponenten. Die Softwareentwicklung beschäftigt sich dann also nicht mehr mit der Entwicklung von

[1] Für die Unterstützung von Softwareentwicklung durch Werkzeuge vergleiche auch [Nag96, Bör93].

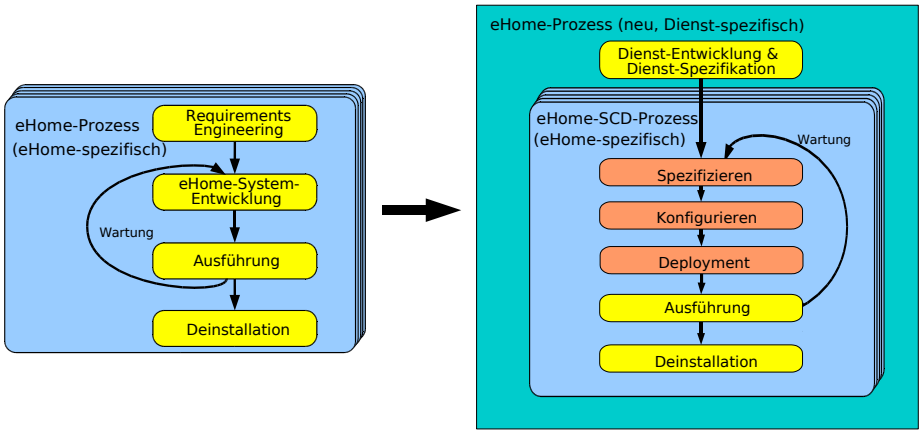

Abbildung 5.1: Vom klassischen repetitiven Entwicklungsprozess hin zur generativen Konfigurierung

dienst- und umgebungsspezifischen Softwaremonolithen, sondern mit der a-priori Entwicklung von wiederverwendbaren Softwarekomponenten, die abstrakt genug sind, um durch einfache Konfigurierung beim Kunden eingepasst werden zu können. Dann bleibt als repetitiver Anteil ein Prozess übrig, der aus den Phasen Spezifikation, Konfigurierung und Deployment besteht. Aber auch der verbleibende Konfigurierungsanteil ist weiterhin sehr aufwändig, wenn er manuell durchgeführt wird. Wie dieser Abschnitt aber zeigen wird, ist er teilweise automatisierbar, wenn die wiederverwendbaren Komponenten und deren formelle Beschreibung gewissen Anforderungen genügen. Durch die Einführung der Funktionalitäten-basierten Komposition wird die teilweise automatische Bestimmung der benötigten Unterdienste und Geräte zu einem gewählten Erweiterungsdienst in einer bestimmten Umgebung möglich. Der entstehende unterstützte Prozess wird hier in dieser Arbeit *eHome-SCD-Prozess*, abgeleitet von Specification, Configuration, and Deployment Process, oder auch nur *SCD-Prozess* genannt.

Die Idee des SCD-Prozesses ist es, eine iterative Kette prozeduraler Techniken zu etablieren, die die Erstellung eines eHome-Systems so weit wie möglich automatisiert. Das bedeutet, dass hier ein Weg gesucht wird, Regeln für die a-priori Entwicklung der wiederverwendbaren Dienste festzulegen und die Spezifizierung, die Konfigurierung und das Deployment eines eHome-Systems in einer normalen Wohnung, also die Transformation einer normalen Wohnung in ein eHome, zu unterstützen. Dies wird durch den Bau spezieller Werkzeuge, Wiederverwendung und Konfigurierung von Softwarekomponenten, die zur Laufzeit eHome-Dienste anbieten, erreicht.

Dabei ist zu beachten, dass die Transformation einer normalen Wohnung in ein eHome nach diesem Prozess nur noch initial, also beim ersten Einsatz, die Entwicklung von Dienstsoftware erfordert. Diese kann durch den Anbieter im Vorfeld entwickelt werden. Anschließend ist beim Kunden nur die Konfigurierung und das Deployment erforderlich.

Als Ergebnis sind die Spezifikation der Heim-Umgebung, die Auswahl und Parametrisierung der gewünschten eHome-Erweiterungsdienste und die Anbringung der benötigten Geräte die einzigen Aktivitäten, die manuell durchgeführt werden müssen. Aktuelle Forschung im Bereich des Konfigurationsmanagements beschäftigt sich hauptsächlich mit manuellem Konfigurationsmanagement und Software-Deployment [WC98, Hoe01]. Im Falle dieser Arbeit müssen die Konfigurierung teilweise und das Deployment vollkommen automatisch durchgeführt werden und Funktionalitäten-basierte Komposition unterstützen. Deshalb werden in dieser Arbeit auch Versionierungsaspekte zugunsten der semantischen Unterstützung vernachlässigt.

Die drei Phasen des SCD-Prozesses sind dann:

1. Spezifizierung der eHome-Umgebung und der benötigten Dienste
2. Automatische Konfigurierung der ausgewählten Dienste
3. Deployment der Konfiguration auf das Service-Gateway im eHome

Sie werden in den folgenden drei Abschnitten beschrieben.

5.1.1 Spezifizierung der eHome-Umgebung und der benötigten Dienste

Während dieser Phase wird die physische architekturbezogene Information über das eHome erfasst – wie die Räume im eHome angeordnet sind, wie sie verbunden sind, welche Bereiche es gibt, wo sich Türen und Fenster befinden und ob und wo bereits elektronisch steuerbare Geräte installiert sind. Gegebenenfalls ist hier in dieser Phase bereits eine Konfiguration vorhanden und es sind schon eHome-Dienste installiert. Um neue Dienste in die Konfigurationsbeschreibung aufzunehmen, müssen diese ausgewählt werden und zur Konfigurationsbeschreibung hinzugefügt werden. Auswählbar sind hier nur top-level Dienste. Weiterhin müssen im Vorfeld vom Anbieter die Dienstbeschreibungen, Gerätebeschreibungen und die semantischen Abhängigkeiten der Dienste im Bezug auf Funktionalitäten spezifiziert worden sein.

5.1.2 Automatische Konfigurierung der ausgewählten Dienste

Die Dienste, die in der Spezifizierungsphase selektiert wurden, werden in dieser Phase automatisch konfiguriert. Dies bedeutet, dass die Geräte, die noch zur Diensterfüllung benötigt werden, der Konfigurationsbeschreibung mit dem Hinweis hinzugefügt werden, dass diese Geräte noch vom Kunden anzuschaffen sind. Unterdienste, die von den gewählten Diensten benötigt werden, werden der Konfigurationsbeschreibung automatisch den Funktionalitäten-Anforderungen entsprechend hinzugefügt. Weiterhin werden für jeden Dienst in der Konfigurationsbeschreibung auch den Lokalitäten korrespondierende Dienstobjekte der Konfigurationsbeschreibung hinzugefügt.

Wenn zum Beispiel der Beleuchtungsdienst so spezifiziert wurde, dass er mindestens eine Lampe und einen Schalter pro Raum benötigt, in dem er angefordert wird, so werden diese Geräte während der Konfigurierung des Lichtdienstes automatisch hinzugefügt. Weiterhin werden die Treiberdienste für die Steuerung der Lampe und die Abfrage des Schalters hinzugefügt.

5.1.3 Deployment der Konfiguration auf das Service-Gateway im eHome

Die Softwarekomponenten, die während der ersten beiden Phasen spezifiziert und konfiguriert wurden, werden in dieser Phase vollautomatisch auf das Service-Gateway des eHomes deployed. Dabei werden die Softwarekomponenten mit den konfigurierten Parametern initialisiert und gestartet.

In dieser dritten Phase des eHome-SCD-Prozesses wird die Konfigurationsbeschreibung durch das Deployment zum Leben erweckt. Bisher existiert das Deployerwerkzeug nur für das Komponentenrahmenwerk OSGi (siehe Abschnitt 4.3). Der hier implementierte Algorithmus ist sehr geradlinig und sollte sich leicht an andere Komponentenrahmenwerke anpassen lassen.

Er besteht aus fünf Schritten:

1. Das Laden der Software Komponenten (in OSGi bundles genannt)
2. Starten der Komponenten
3. Aufbau der Referenzen zwischen der Laufzeitkonfigurationsbeschreibung und den Laufzeitkomponenten
4. Die Initialisierung der Softwarekomponenten in Korrespondenz zu ihren Abhängigkeiten
5. Aufruf der Dienstobjektschnittstelle in den Softwarekomponenten in ihrem jeweiligen Dienstkontext.

Mögliche Erweiterungen dieser Deployment-Methode liegen im Bereich der dynamischen Rekonfiguration und der Anpassung an andere Komponentenrahmenwerke.

5.2 eHome-Modell

Um eine werkzeuggestützte Unterstützung der Spezifizierung und der Konfigurierung im eHome-SCD-Prozess zu ermöglichen, ist es sinnvoll die möglichen auftretenden Konfigurationen in einem Modell zu beschreiben. Dieses Modell wird anhand der im letzten Kapitel ermittelten Ergebnisse und in Bezug auf einen automatisierten Konfigurierungsprozess erstellt. Es wird in dieser Arbeit *eHome-Modell* genannt. Der gesamte SCD-Prozess bezieht sich sehr stark auf dieses Modell.

Es beinhaltet all die Informationen, welche für die Unterstützung des eHome-Prozesses und die Ausführung der eHome-Dienste nötig sind. Das instanzierte Modell, die *eHome-Modell-Instanz* bzw. kurz Konfiguration, spielt eine aktive Rolle zur Laufzeit eines eHome-Systems, da es für die aktiven eHome-Dienste inferierbare Informationen über die Heimumgebung zur Verfügung stellt.

Dies ermöglicht die Entwicklung *kontextsensitiver* Dienste, welche aktuelle Informationen über den Ort der Einwohner, die Zustände benachbarter Dienste oder von diesen gesteuerten Geräten benötigen. Diese Dienste können sich also der ihnen physikalisch zugeordneten Umgebung entsprechend verhalten und kommunizieren.

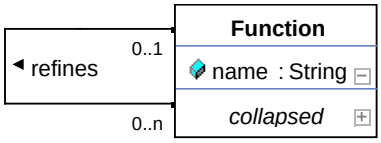

Abbildung 5.2: Der Funktionalitäten-Kontext

In anderen Worten ist also eine eHome-Modell-Instanz auch ein Kommunikationsmedium für eHome-Dienste. Ein vergleichbarer Ansatz für Kommunikation wird auch in [AGLH06] (siehe auch Kapitel 8) verfolgt.

Das eHome-Modell wird mit Hilfe der Fujaba-Werkzeug-Suite[Zün99, NNZ00, KNNZ99] entwickelt. Dabei ermöglicht Fujaba die Entwicklung eines Objekt-orientierten Modells und die Generierung von vollständigem und ausführbaren Java-Code[2], um Modellinstanzen zu erzeugen und zu bearbeiten. Das Modell hat einen statischen Teil und einen dynamischen Teil. Der statische Teil wird mit der Hilfe eines UML-Klassen-Diagramms beschrieben. Dort sind die Klassen des Modells und ihre Beziehungen untereinander aufgeführt. Während der Laufzeit der eHomeConfigurator-Werkzeugsuite werden die Objekte der Klassen des Modells entsprechend der Struktur dieses Klassendiagramms instanziert. Eine aktuelle Publikation über das eHome-Modell ist unter [NARS06] zu finden.

In den folgenden Unterabschnitten werden die verschiedenen Kontexte, die sich aus dem Modell ableiten lassen beschrieben. Diese sind der Funktionalitäten-Kontext (eng.: functionality context), der Geräte-Definitions-Kontext (eng.: device definition context), Umgebungs-Kontext (eng.: environment context), Dienst-Kontext (eng.: service context), Dienst-Objekt- oder Dienst-Instanz-Kontext (eng.: service object oder service instance context) und Personen-Kontext (eng.: person context). Alle diese Kontexte sind miteinander verbunden und bilden zusammen das eHome-Modell. Sie werden hier getrennt aufgeführt, um die kontextspezifischen Details zu betonen.[3]

5.2.1 Funktionalitäten-Kontext

Der kleinste aber auch gleichzeitig mit der wichtigste Kontext des eHome-Modells ist der Teil, der die Funktionalitäten der Dienste und implizit auch der von diesen kontrollierten Geräte beschreibt, der *Funktionalitäten-Kontext* (eng.: *Functionality Context*). Der entsprechende Kontext des eHome-Modells ist in Abbildung 5.2 dargestellt. Funktionalitäten werden durch die Klasse `Function` dargestellt, welche eine reflexive Relation besitzt, um Verfeinerungen von Funktionalitäten durch andere darzustellen. Zum Beispiel kann die Funktionalität Erkennung (detection) durch die Funktionalitäten wie Bewegungserkennung (movement detection), Raucherkennung (smoke detection) oder Glasbrucherkennung (glas breakage detection) verfeinert werden. Diese reflexive Relation der Klasse `Function` impliziert eine Baumstruktur des korrespondierenden Modellinstanzgraphen, dessen Blätter Objekte der Klasse `Function` sind. Bisher werden Funktionalitäten nur

[2] Es ist auch möglich Code anderer Programmiersprachen zu erzeugen.
[3] Ein kurzer Überblick über die verschiedenen Kontexte findet sich auch in [NM06b].

durch ihren Namen und ihre Position im Verfeinerungsgraphen definiert. Die Verfeinerungsrelation sollte so benutzt werden, dass die allgemeinste Funktionalität die Wurzel des Baumes ist und die speziellsten Funktionalitäten die Blätter. Alle in dieser Arbeit getesteten Funktionalitäten sind in Abbildung 5.3 abgebildet.

5.2.2 Geräte-Definitions-Kontext

Die Geräte, die in der Umgebungsspezifizierung benutzt werden, sind vordefiniert. Um also ein Gerät zu einer eHome-Umgebung hinzufügen zu können, muss eine Art Metaklasse für dieses instanziert werden, welche den Typ des Gerätes beschreibt. Diese Metaklasse wird hier *Geräte-Definition* (eng.: *Device Definition*) genannt. Der Klasse `DeviceDefinition` können verschiedene Attribute zugeordnet werden, wie zum Beispiel Herstellername, oder aber auch ein einstellbarer Adresscode (zum Beispiel der Housecode bei X10-Lampenfassungen, eine USB-Geräteadresse oder eine IP-Adresse). Name (`name`) und die Anzahl vorhandener virtueller Geräte (`numberOfVirtualDevices` – später wichtig in der Konfigurierungsphase) sind feste Variablen dieser Klasse.

Die gerade besprochenen Attribute können eine beliebige Struktur aufweisen. Sie sind also nicht, wie in einer früheren Version des Modells auf reine Zeichenketten-basierte Schlüssel-Wert-Paare festgelegt, sondern können auch komplexe Objektstrukturen beinhalten. Deshalb enthält die Klasse `Attribute` einen abstrakten `ValueHolder`, der dazu erweitert werden kann, beliebige Datentypen zu beinhalten.

Ein konkretes Gerät wird in der Umgebung als Instanz der Metaklasse `DeviceDefinition` spezifiziert, welches sich dann durch die Festlegung der spezifischen Attribute auszeichnet. In Abbildung 5.4 wird die Gerätedefinition-Meta-klasse durch die Klasse `DeviceDefinition` dargestellt. Das tatsächliche im Haus vorhandene Gerät wird durch die `Device`-Klasse dargestellt. Attribute haben die Form der `Attribute`-Klasse, welche auch in anderen Kontexten benutzt wird.

Die Funktionalitäten, die von Geräten zur Verfügung gestellt werden, sind selbstverständlich wichtig für den SCD-Prozess. Sie werden allerdings nicht bei der Gerätedefinition angegeben sondern bei der Spezifikation der Treiberdienste, die diese Geräte steuern. Dies spiegelt sich in der Tatsache wieder, dass sowohl die Hardware selber als auch die physikalischen Verbindungen innerhalb des SCD-Prozesses keine signifikante Rolle für die Konfigurierung der Software spielen. Die Konfigurierung wird allein auf der Basis der Softwarekomponenten durchgeführt. Einige davon sind dann in der Lage, Geräte zu steuern.

5.2.3 Umgebungs-Kontext

Der *Umgebungs-Kontext* (eng.: *Environment Context*) modelliert alle Ortsinformationen in Bezug zum Grundriss der Wohnung und dort installierter oder zu installierender Geräte. Der Umgebungs-Kontext ermöglicht die Modellierung verschiedener Umgebungen, die über *Lokationen* (eng.: *Locations*) bzw. *Lokationselementen* (eng.: *Location-Elements*) verbunden sind. Zum Beispiel kann ein Haus mit einer externen Garage über einen Flur oder eine Tür verbunden sein. Weiterhin ist auch ein Unterlokationskonzept vorgesehen. So lassen sich die Räume einer Etage als Unterlokationen modellieren oder ein Raum lässt

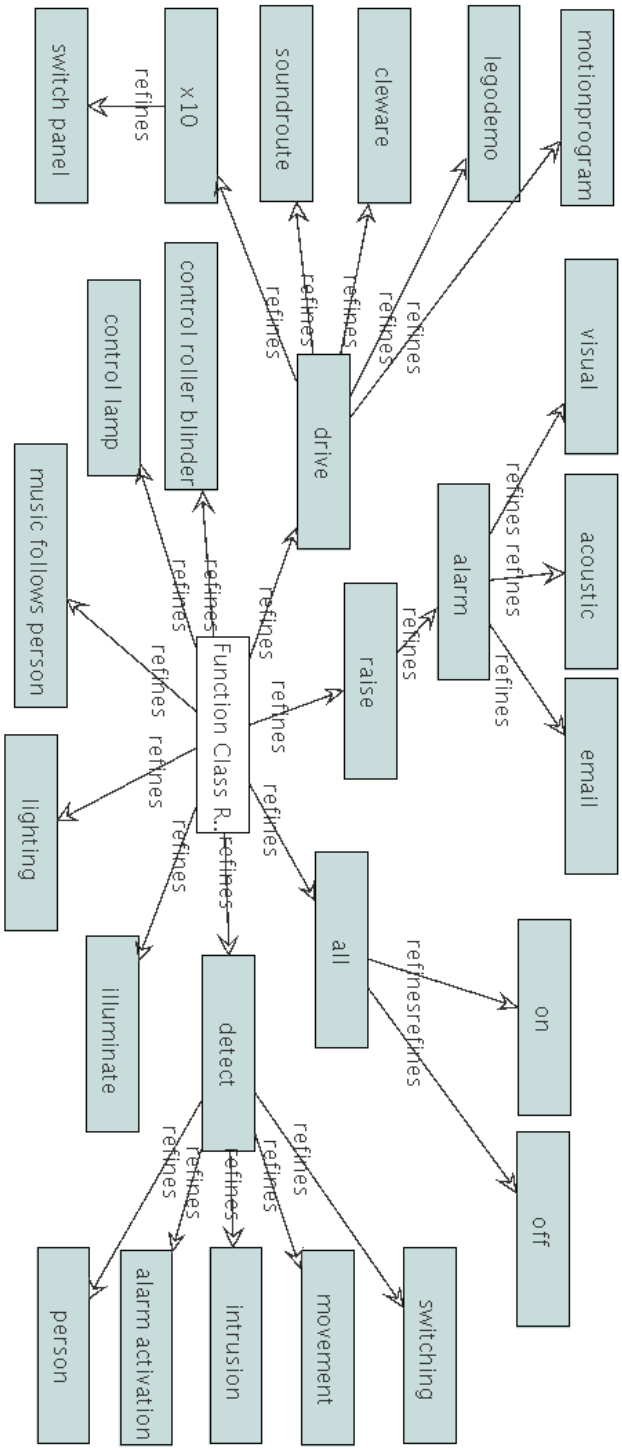

Abbildung 5.3: Übersicht aller getesteten Funktionalitäten

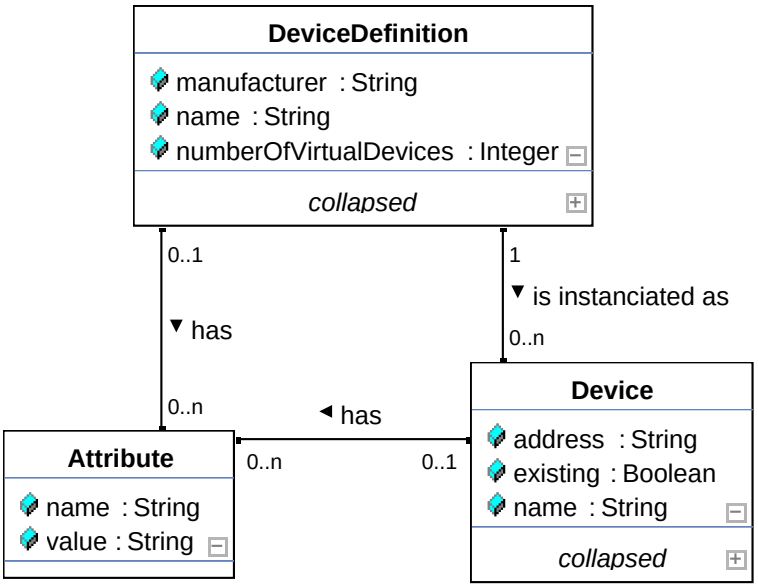

Abbildung 5.4: Der Gerätedefinition-Kontext

sich in verschiedene dienstbezogene Bereiche wie Fensterbereich, Küchen- oder Fernsehecke, und Wohnbereich aufteilen. Diese Modellierungfreiheit und Generizität stellt einen Mechanismus zur Verfügung, der mächtig genug ist, um jede logische Struktur des architektonischen Designs von Wohnungen darzustellen. In dieser Arbeit werden normalerweise nur Grundrisse einer Etage betrachtet, es ist aber auch möglich, dreidimensionale Gebilde zu beschreiben, indem verbindende und beschreibende Lokationselemente eingesetzt werden.

Der Umgebungskontext besitzt die Klasse `EnvironmentElement` als Oberklasse für alle anderen Klassen, die Lokationsinformationen der Wohnung beschreiben (siehe auch Abbildung 5.5).

Die Klasse `EnvironmentElement` vereint die allgemeinen Fähigkeiten der Klassen, die benutzt werden, um eine eHome-Umgebung zu beschreiben. Da die Klassen `Environment`, `Location` und `LocationElement` von ihr erben, stehen sie alle in Relation zur Klasse `Device`. Dies bedeutet, dass jedes Element einer Umgebung diesem zugeordnete Geräte besitzen kann. Zum Beispiel kann dadurch ausgedrückt werden, dass sich im Wohnzimmer eine Lampe, eine Stereoanlage mit Lautsprechern und ein LCD-Bildschirm, sowie Steuerschalter und -knöpfe befinden. An der Tür des Zimmers, welche auch ein Lokationselement ist, kann ein Schließsensor installiert sein und an dem Fenster ein Glasbruchsensor.

Umgebungen, Lokationen und Verbindungslokationen können zu einem komplexen Graph zusammengefügt werden, welcher die logischen Verbindungen eines architektonischen Designs beschreibt. Die Verbindungen beschreiben die Relationen zwischen den einzelnen Entitäten des architektonischen Designs. Ein Beispiel einer solchen Umgebung sieht man in Abbildung 5.6. Diese wird in Abschnitt 5.2.7 genauer erläutert.

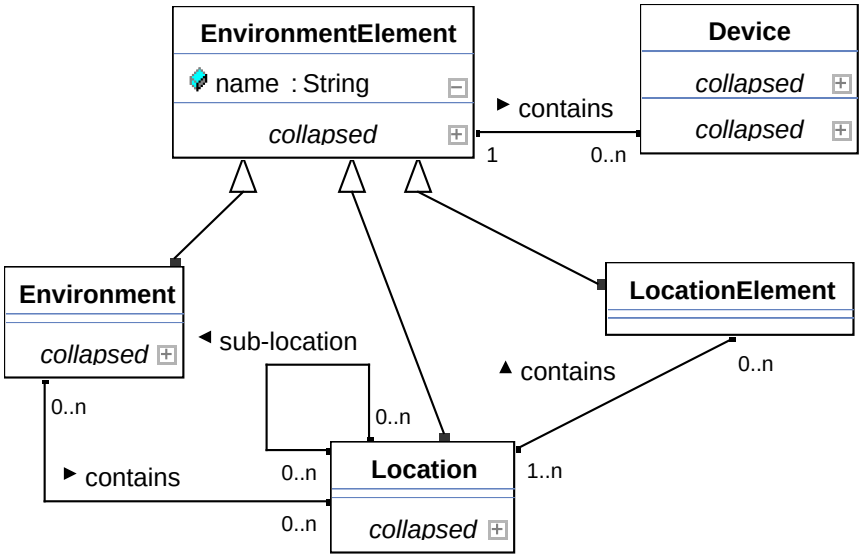

Abbildung 5.5: Der Umgebungskontext

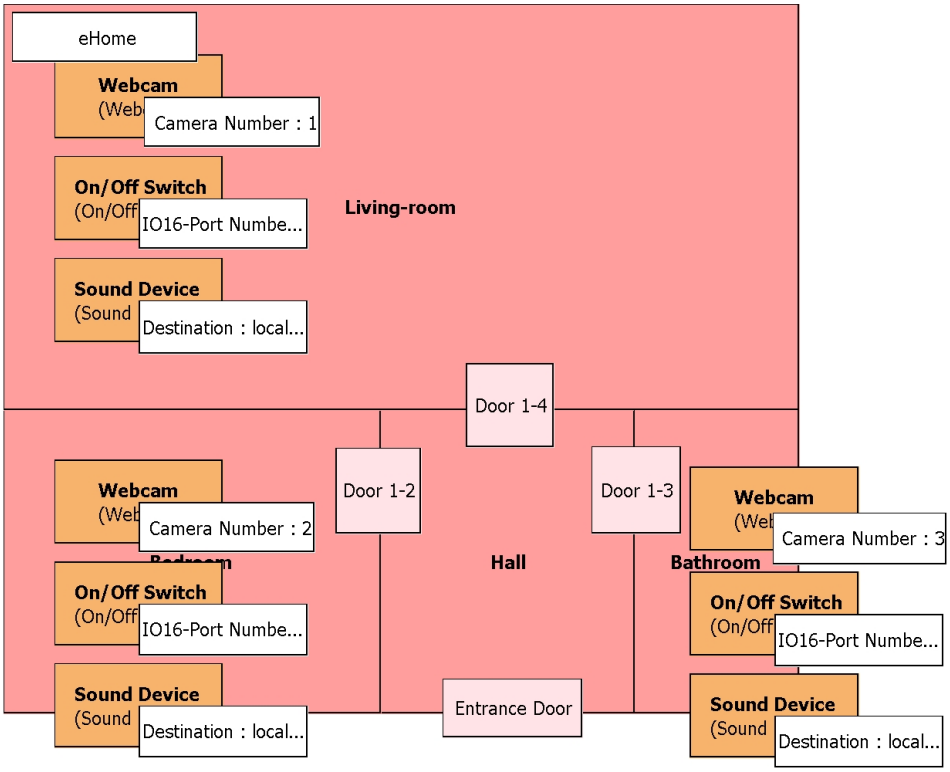

Abbildung 5.6: Umgebungselemente und Geräte in einer Umgebung

5.2.4 Dienst-Kontext

Der *Dienst-Kontext* (eng. *Service Context*) des eHome-Modells beinhaltet die eHome-Dienst-Beschreibungen, beziehungsweise genauer die Definitionen von eHome-Diensten. Für den SCD-Prozess ist es ist unerlässlich, dass die den eHome-Dienst realisierenden Softwarekomponenten, die während der automatischen Konfigurierung eingestellt und miteinander in Verbindung gesetzt werden, vorher modelliert werden, da die automatische Konfigurierung diese Beschreibungen benutzt. Der Dienstkontext beinhaltet nicht die Laufzeitkonfiguration der ausgewählten Dienste, welche tatsächlich auf das oder die Service-Gateways während der Deployment-Phase des SCD-Prozesses installiert werden.

Der Dienstkontext des Modells wird in Abbildung 5.7 dargestellt. Der Dienst selbst wird durch die `Service`-Klasse modelliert, die nicht nur Informationen wie ID, Name, Typ und Beschreibung beinhaltet sondern auch Informationen über die Resource-URL der realisierenden Softwarekomponente, die in der Deployment-Phase des SCD-Prozesses installiert und an der richtigen Stelle ausgeführt werden muss. Die Klasse `Service` ist eine abstrakte Beschreibung, der den durch sie angegebenen Dienst realisierenden Softwarekomponente, die zur Laufzeit des eHome-Systems ausgeführt wird.

Der essentielle Teil der Dienstbeschreibung wird durch Funktionalitäten spezifiziert. Die Klasse `Service` hat drei indirekte Relationen zur Klasse `Function` über die Klasse `ServiceFunctionCardinality`. Ein Dienst zeichnet sich durch die Funktionalitäten aus, die er anbietet (provides), anfordert (requires) und optional benutzen (optionally requires) kann. Wie bereits zuvor erwähnt werden die Funktionalitäten der Geräte als Teil ihrer Treiberdienste angesehen. Die Funktionalitäten erlauben die dynamische Komposition und Abhängigkeitsauflösung während des automatischen Konfigurierungsschrittes. Sie formen somit eine Abstraktionsschicht für die Dienstkomposition.

Die Klasse `ServiceFunctionCardinality` wird auch für die Dienstanforderungsspezifizierung benutzt. Die Kardinalität am angeforderten Dienst gibt die Quantität an in der eine Funktionalität benötigt wird. Abbildung 5.8 zeigt einen Dienst, der einen Schalter abfragen kann. Dieser kann von einem zweistufigen Heizsystem benutzt werden. Der zugehörige Heizsystemdienst würde dann eine `switching`-Funktionalität mit Kardinalität 2 fordern. So könnte dann der eine Schalter benutzt werden, um die Wandheizung ein- und auszuschalten und der zweite, um die Fußbodenheizung zu bedienen. Wird dieser Heizsystemdienst in einer Lokation installiert, so werden dort auch zwei Schalterdienstobjekte und, falls noch nicht vorhanden oder belegt, zwei Schalter-Geräte der Konfiguration hinzugefügt. Wenn bereits ein anderer Dienst in der Lokation die `switching`-Funktionalität mit einer Kardinalität größer oder gleich 2 anbietet, müssten keine weiteren Dienstobjekte der Konfiguration hinzugefügt werden. Wenn der Dienst die switching-Funktionalität mit einer Kardinalität echt größer als 2 anbietet, so kann das entsprechende Dienstobjekt in der Lokation auch noch von anderen Diensten genutzt werden, die diese Funktionalität benötigen.

Die Klasse `Service` hat zwei Relationen zu der Klasse `Attribute`. Diese beiden Relationen modellieren globale und lokale bzw. spezifische Attribute. Globale Attribute gelten für die gesamte eHome-Umgebung, während die lokalen Attribute in jeder Lokation unterschiedlich sein können.

Dienste sind über eine Relation mit den Umgebungsinformationen verbunden. Eine Relation zwischen einer Instanz der `Service`-Klasse und einem Umgebungselement al-

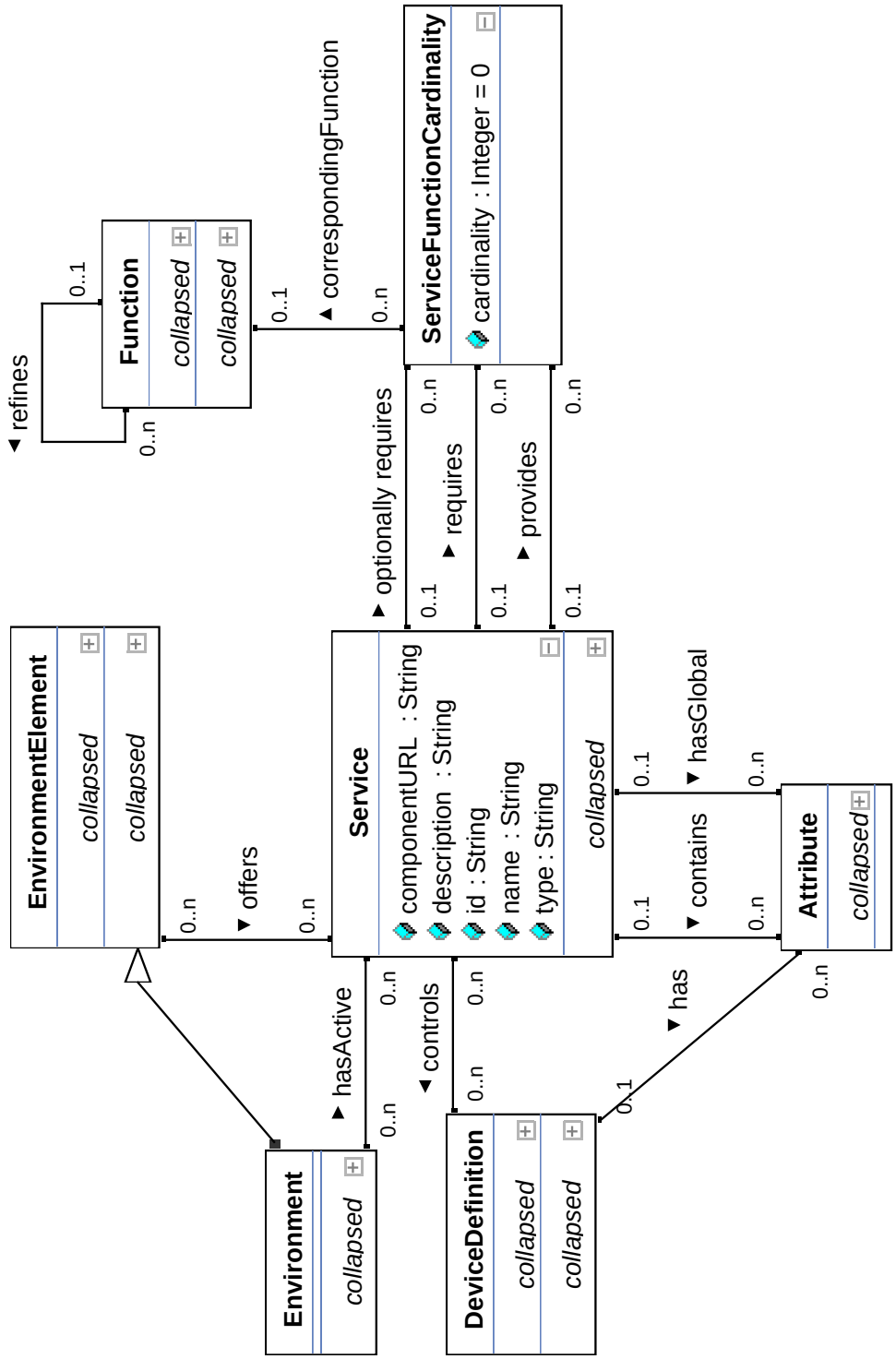

Abbildung 5.7: Der Dienstkontext

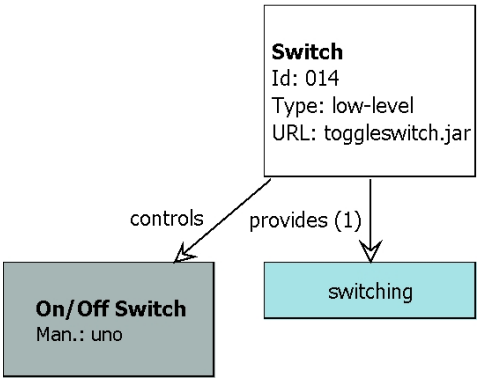

Abbildung 5.8: Ein Beispiel für einen Treiberdienst

so einer Instanz der Klasse `EnvironmentElement` oder einer abgeleiteten Klasse impliziert dass in oder an diesem Umgebungselement dieser Dienst dem eHome-Bewohner angeboten wird. Normalerweise wird es sich bei der abgeleiteten Klasse um die Klasse `Location` handeln. Diese Relationen werden bei der Dienstauswahl in der Spezifikationsphase angelegt.

Die Assoziation zwischen der `Service`-Klasse und der `DeviceDefintion`-Klasse deutet an, dass Geräte von Diensten, die die Gerätefunktionalität zur Verfügung stellen, gesteuert werden. Dies ist in der Regel bei Treiberdiensten der Fall. Die realisierenden Softwarekomponenten sind also Gerätetreiberkomponenten. Die gesteuerten Geräte selbst haben keine große Auswirkung auf die automatische Konfigurierung im SCD-Prozess. Ein Beispiel eines solchen Treiberdienstes ist in Abbildung 5.8 zu sehen.

5.2.5 Dienstobjekt-Kontext

Im Dienstkontext des eHome-Modells wird der Dienst durch seine Funktionalitäts-Abhängigkeiten modelliert. Der *Dienstobjekt-Kontext* (eng.: *service object context*) ermöglicht die Instanzierung der Laufzeitkonfigurationsinformationen der eHome-Dienste in Bezug auf ihren physikalischen Wirkungsbereich. Dies ermöglicht die tatsächliche lokale Referenz von Informationen. Also eine wirkliche Möglichkeit physikalisch verwandte Informationen aus der eHome-Modell-Instanz zu inferieren.

Während der Konfigurierungsphase des SCD-Prozesses werden die Teile der eHome-Modell-Instanz, der Konfiguration, korrespondierend zu ihrem Kontext automatisch aufgebaut. Der Dienstobjektkontext beschreibt den Aufbau der Dienstkonfiguration zur Laufzeit des eHome-Systems. Um eine bessere Übersicht zu gewährleisten, wurde der Dienstkontext auf zwei Abbildungen verteilt, siehe Abbildung 5.9 und 5.10.

Abbildung 5.9 zeigt die Klasse `ServiceObject`, welche für jede Lokation instanziert wird, in der ein Service angeboten wird oder auf Grund von Abhängigkeiten angeboten werden muss. Deshalb gibt es in Abbildung 5.10 eine Relation `is in` zwischen den Klassen `ServiceObject` und `EnvironmentElement`. Diese Relation korrespondiert zu der `offers`-Relation in Abbildung 5.7. Die Idee der Instanzierung einzelner

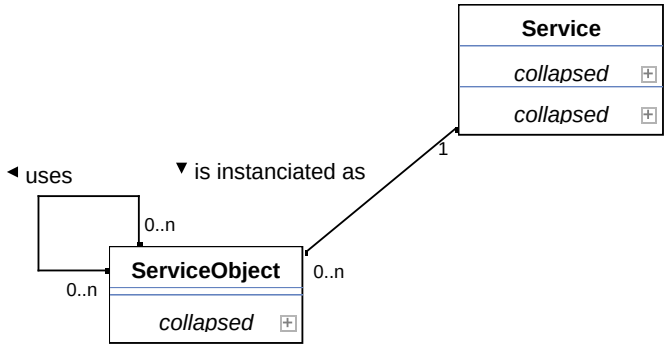

Abbildung 5.9: Die Relation zwischen Dienst und Dienstobjekt

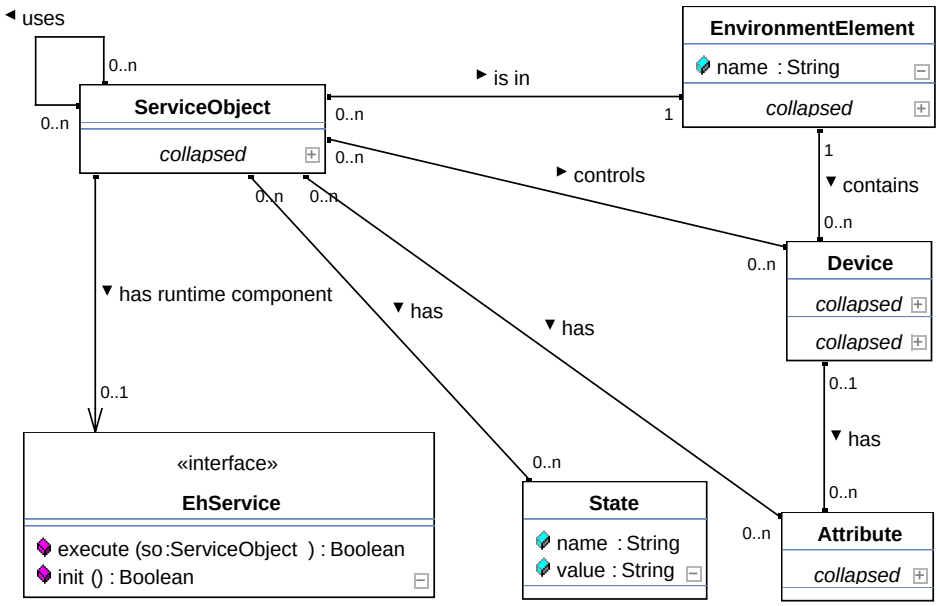

Abbildung 5.10: Der Dienstobjektkontext

Dienste durch mehrere Dienstobjekte ist die Möglichkeit, jedem Dienst für jede Lokation bzw. jeden Raum, in dem er angeboten wird, eine spezifische Konfiguration und auch einen unterschiedlichen Kontext zur Verfügung stellen zu können. Diese Instanzierung wird durch die `is instanciated as`-Relation in Abbildung 5.9 ausgedrückt.

Die für den SCD-Prozess wichtigste Relation im Dienstobjektkontext ist die Selbstrelation `uses` der `ServiceObject`-Klasse. Dadurch lässt sich eine benutzt-Beziehung zwischen Softwarekomponenten zur Laufzeit darstellen. Im Dienstkontext findet sich diese Relation nicht explizit. Dies liegt daran, dass für die Komposition der Dienste die Abstraktion der Komposition über die Funktionalitäten durchgeführt wird, um eine automatische Konfigurierung und Komposition während des SCD-Prozesses zu ermöglichen.

Der Konfigurationsgraph des Dienstobjektkontext wird nur während der Konfigurierungsphase des SCD-Prozesses aufgebaut. Die Werkzeuge, die die automatische Konfigurierung unterstützen, berücksichtigen die vom eHome-Benutzer ausgewählten top-level Dienste und ihre Funktionalitäten-Anforderungen und komponieren dann eine passende Menge von Diensten, um diese Anforderungen zu erfüllen und somit im Endeffekt die Funktionalitäten anzubieten, die der Benutzer mit den top-level Diensten gewählt hat.

Die Benutzungsrelationen zwischen den Dienstobjekten werden explizit ausgedrückt. Die Komposition benutzt die indirekten `provides`-, `requires`- und `optionally requires`-Relationen zwischen der `Service`- und der `Function`-Klasse (siehe Abbildung 5.7). Diese Relationen werden benutzt um den Benutzungs- und Abhängigkeitsgraphen der Dienstobjekte aufzubauen und so die Laufzeitstruktur und Konfiguration der eHome-Dienste im eHome-System auszudrücken.

5.2.6 Personen-Kontext

Innerhalb des *Personen-Kontexts* ist die Struktur der personenbezogenen Information festgelegt. Diese spielen eine wichtige Rolle während der Laufzeit eines eHome-Systems. Beispielsweise, wenn Dienste den Aufenthaltsort einer Person nutzen, um dieser Person einen Dienst an der Stelle anzubieten, an der sie sich gerade befindet. Diese Informationen sind aber auch in anderen Prozessen wie Geschäftsprozessen (siehe auch in [Kir05]) oder auch bei der Migration von Diensten zwischen unterschiedlichen eHome-Umgebungen interessant. Deshalb ist dieser Kontext Bestandteil des eHome-Modells.

Abbildung 5.11 zeigt die Klasse `Person`, welche die Struktur von Personen in diesem Kontext vorgibt. Über die Relation `is in` kann festgelegt werden, in welcher Lokation sich die Person gerade befindet. Diese Information kann von den Diensten benutzt werden, die ortssensitiv sind. Um einer Person einen bestimmten Dienst zuzuordnen, der in einer späteren Arbeit gegebenenfalls mit dieser Person in eine andere Umgebung migrieren kann, wird die Relation `uses` benutzt. Um Profile oder einfach strukturierte personenspezifische Informationen zu speichern werden Attribute benutzt. Deshalb existiert auch hier eine Relation zur `Attribute`-Klasse.

5.2.7 Beispiel einer Konfiguration auf Basis des eHome-Modells

Als motivierendes Beispiel für eine Konfiguration, die mit diesem Modell beschrieben werden kann, wird hier eine kleine Appartementwohnung betrachtet, die aus einer Diele (eng.: hall), einem Badezimmer (eng.: bathroom), einem Wohnzimmer (eng.: living-

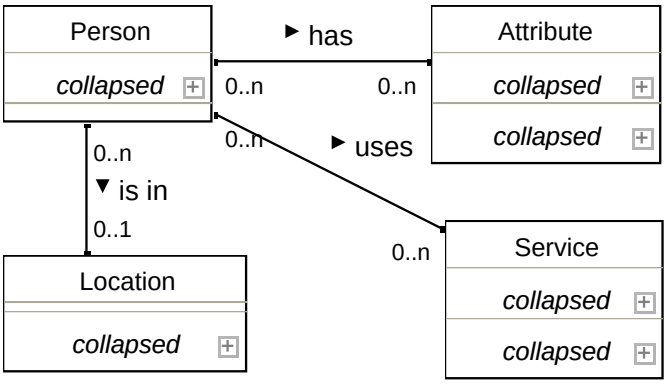

Abbildung 5.11: Der Personenkontext

room) und einem Schlafzimmer (eng.: bedroom) besteht. Die Darstellung des Umgebungskontext für dieses Beispiel war bereits in Abbildung 5.6 im Abschnitt 5.2.3 zu sehen. Das Wohnzimmer hat eine kleine Küchenecke zum Kochen. Es wird angenommen, dass in dieser Wohnung ein Studentenpärchen lebt. Diese hören gerne Musik und möchten jeweils gerne in jedem Raum in diesem Apartment immer ihre jeweilige Lieblingsmusik hören, zum Beispiel wenn einer von ihnen ein Bad nimmt. Ihre Musiksammlung befindet sich auf einem Computer und kann über Funk ins Netzwerk gestreamt werden. Sie möchten nun den `Music-Follows-Person`-Dienst in ihrem Heim einrichten. In Abbildung 5.12 ist ein Ausschnitt aus der vollständig durchkonfigurierten Konfiguration, also ein Ausschnitt der deployfähigen eHome-Modell-Instanz, als Graph dargestellt. Es handelt sich deshalb hier insbesondere um die Graphsicht des Deploymentanteils. In der Abbildung ist nur der Teil zu sehen, der relevant für das Wohnzimmer und den Dienst `Music-Follows-Person` ist, und die entsprechenden Beziehungen zwischen den Objekten und dem Umgebungselement `Living-room`. Die Knoten und Kanten repräsentieren hier Folgendes:

Objekte:

Geräteobjekte: Als Geräte sind hier `Sound Device`, `On/Off Switch` und eine `Webcam` abgebildet.

Dienstobjekte: Dienstobjekte der Unterdienste `Person Detector`, `Motion Controller` und `Switch`.

Attributobjekte: Mit Geräten und Diensten sind teilweise Attribute assoziiert. So hat im angegebenen Beispiel der On/Off Switch die `IO16_Port Number 6` und das `Sound Device` die Abspielzieladresse `localhost`.

Stateobjekte: Gestartete Dienste können auch States haben. Diese sind ähnlich wie Attribute, aber nur Laufzeitinformationen eines Dienstes. Sie müssen nicht bei der Konfigurierung des Dienstes bestimmt werden. Ein Beispiel für einen solchen State ist hier der State `person`, welcher anzeigt, welche Person gerade in Wohnzimmer detektiert wurde. Der State `switch` zeigt an, ob

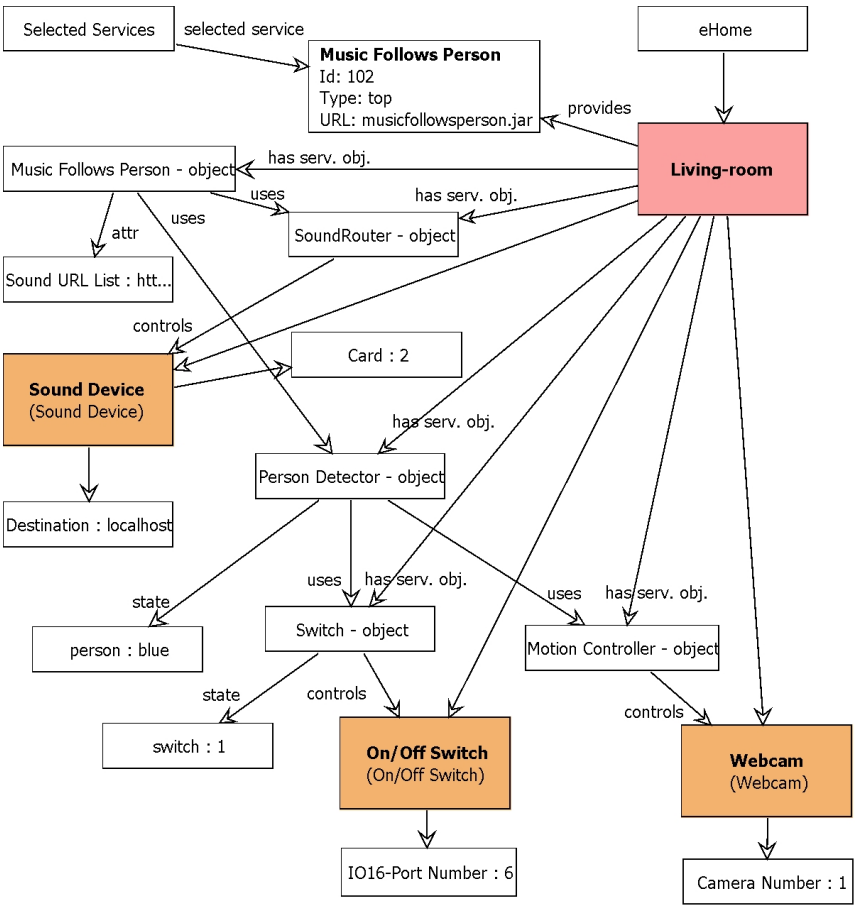

Abbildung 5.12: Teil der Deployment-Konfiguration für den MusicFollowsPerson-Dienst im Wohnzimmer

der Schalter gerade gedrückt ist oder nicht. In diesem Fall hat er den Wert 1, ist also gedrückt.

Relationen:

`uses`-Abhängigkeiten: Die `uses`-Kanten zeigen Abhängigkeitsbeziehungen zwischen den durch die Dienste benutzen Softwarekomponenten an. Das MusicFollowsPerson-Dienstobjekt benutzt das SoundRouter-Dienstobjekt

`contains`-Relation: Die `contains`-Kanten zeigen eine physikalische Enthaltensseinsbeziehung an. So befindet sich das Sound Device im Wohnzimmer, ist physikalisch gesehen im Wohnzimmer enthalten.

`has serv. object`-Relation: Die `has serv. object`-Relation zeigt die Verantwortlichkeit einer Softwarekomponente innerhalb eines Dienstobjektes zur Laufzeit für eine bestimmte Lokation an. In der Abbildung ist das Soundrouter-Dienstobjekt für die Soundausgabe im Wohnzimmer verantwortlich.

`attr`-Relation: Dies sind die Kanten zwischen Attributen und Dienstobjekten bzw. Geräten (siehe Attribut-Objekte).

`state`-Relation: Dies sind analog die Kanten für States.

5.3 Spezifikatorerzeugung

Während der Projektzeit dieser Arbeit hat das zugrundeliegende eHome-Modell mehr als zehn wesentliche Veränderungen erfahren. Weiterhin wurden oft kleinere Änderungen an der Modellstruktur durchgeführt. Jede größere Design-Entscheidung betraf strukturelle Änderungen des Klassen-Modells: Hinzufügen und Löschen von Klassen; Hinzufügen, Ändern und Löschen von Assoziationen; Hinzufügen, Ändern und Löschen von Assoziationen von Klassenattributen und -methoden (siehe auch Abbildung 5.13). Alle diese Änderungen fordern in der Regel manuelle Programmierung, um den eHomeConfigurator an diese anzupassen.

Eine solche Veränderung war die Einführungen von Kardinalitäten an den `provides`-, `requires`- und `optionally requires`-Relationen der `Service`-Klasse. Hätte jede solche Änderung weitreichende Codeänderungen am Spezifikatorteil des eHomeConfigurators zur Folge gehabt, wären sicher viele Ergebnisse dieser Arbeit nicht erreicht worden, da die Entwicklung sonst zu zeitaufwändig gewesen wäre. Deshalb wurde ein Mechanismus entwickelt, das Spezifikatorwerkzeug größtenteils direkt aus dem Modell zu generieren. Für die Modellierung des Modells und der Aktionen, die auf dem Modell durchgeführt werden können gibt es bereits Systeme, die eine automatische Codegenerierung aus diesem Modell erlauben [Sch91, KNNZ99, ERT99]. Die Frage, die verbleibt, ist, wie soll nun die Modellinstanz dem Benutzer präsentiert werden? Das heißt Model (aus Model-View-Controller) und Controller können bereits einfach graphisch modelliert werden, die Darstellung bzw. der View aber noch nicht.[4] Es gibt zwar auch hier Ansätze, wie zum Beispiel das Upgrade Framework [Jäg00, BJSW02], die diesen Aspekt

[4] Im Sinne des Model-View-Controller-Design-Patterns. [GHJV95, KP88]

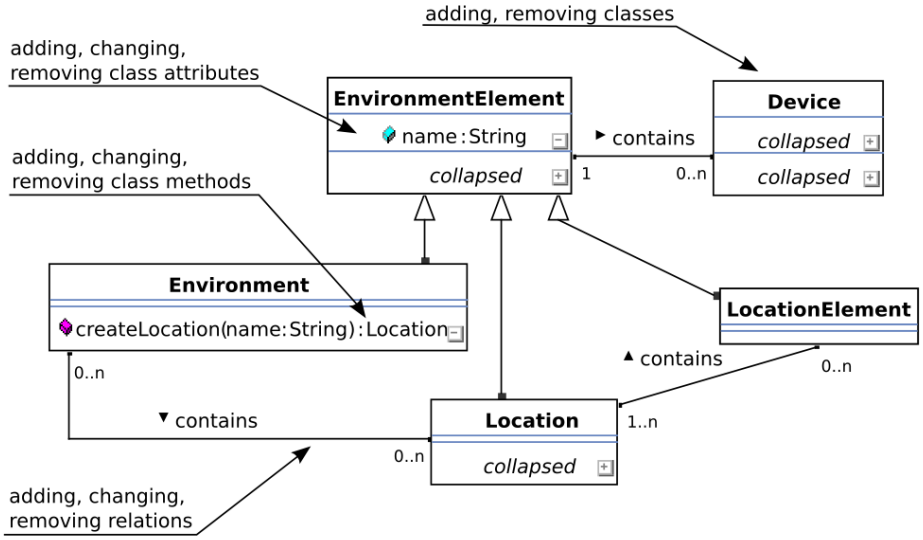

Abbildung 5.13: Mögliche Modelländerungen

behandeln, doch war der Einsatz dieses Rahmenwerks wegen ungeklärter lizenzrechtlicher Probleme und einem nicht zumutbaren Installationsaufwand, der aus der Kopplung mit PROGRES [Sch91, SWZ99] resultierte, nicht sinnvoll. Die Programmierung von Upgrade und dem DOBS [GZ02] von Fujaba [Zün99, NNZ00, KNNZ99] lieferten zwar die Idee für die hier gewählte Realisierung, dennoch wurde ein eigener Ansatz auf der Basis von Translatoren und einer Activities-Beschreibungsdatei gewählt [NSSK05].

Bei der Entwicklung der Werkzeugsuite eHomeConfigurator wurde darauf geachtet, dass sie als ein Open-Source-Projekt unter der *Gnu Lesser General Public License* (*LGPL*) [GNU99] veröffentlicht werden konnte:

1. Zum einen sollte gewährleistet werden, dass andere Forscher und Entwickler die Möglichkeit haben, sich an diesem Projekt zu beteiligen, die Änderungen und Verbesserungen an der Werkzeugsuite aber wieder dem eigentlichem Projekt zu Gute kommen lassen müssen.

2. Weiterhin sollte somit jedem, unter anderem auch möglichen Kooperationspartnern, die Möglichkeit gegeben werden, die Suite zu installieren und selber zu testen.

3. Und schließlich sollte Kooperationspartnern die Möglichkeit eingeräumt werden, die hier entwickelte Software als eine Art Bibliothek in ihren eigenen Projekten nutzen zu können, ohne ihre zusätzlich entwickelten Applikationen oder Spezifikationen selber veröffentlichen zu müssen, was im Falle der Wahl der *Gnu General Public License* (*GPL*) [GNU91] der Fall gewesen wäre.

Die programmierten Translatoren geben also an, wie bestimmte Objekte und deren Zusammenhänge in unterschiedlichen Sichten dargestellt werden sollen. Die Beschreibungs-

datei gibt an, welche Sichten angezeigt werden sollen und welche Methoden als Knöpfe beziehungsweise als Menüeinträge sichtbar sein sollen. Für die Implementierung von Translatoren siehe Abschnitt 6.5.1 auf Seite 162.

5.4 Dienstentwicklung

Da die Dienste in der Lage sein sollen, zur Laufzeit kontextabhängige Informationen zu inferieren, sollte die Kommunikation zwischen Diensten über States, also über das Modell, und möglichst selten über Methoden in der Schnittstelle erfolgen. Dies erleichtert die Abstraktion erheblich und ermöglicht es eher, Dienste in Zusammenhängen zu benutzen, die bei ihrer Erstellung nicht geplant waren, als ein direkter Methodenaufruf. Sollte ein Dienst also eine Art internen Status, wie `aktiv` oder `inaktiv` oder `angeschaltet` oder `ausgeschaltet` (wie zum Beispiel bei einer Lampensteuerung) haben, so macht es Sinn, hierfür im Dienst einen State anzulegen, auf den sich dann im Sinne des Observer-Patterns [GHJV95] andere Dienste registrieren können. Allerdings sollte der Dienst den von ihm angelegten State auch selber überwachen, um auf Änderungen reagieren zu können und so zum Beispiel im Falle der Lampensteuerung bei der Änderung des Status auf `angeschaltet`, auch die gesteuerten Beleuchtungsmedien einzuschalten. Auch muss wegen der Beziehung zu einem physikalischen Umgebungselement immer überprüft werden, ob nicht mehrere Unterdienste oder Geräte von dem gerade entwickelten Dienst gesteuert werden.

Da ein Dienst in mehreren Dienstobjekten entsprechend den Umgebungselementen, für die er zuständig ist, instanziert werden muss, werden zwei Methoden benötigt:

1. Eine globale Initialisierung (`init`), die beim Start der Softwarekomponente des Dienstes ausgeführt wird und gegebenenfalls lokationsübergreifende Initialisierungen vornimmt, also Initialisierungen, die für die ganze eHome-Umgebung gelten. Dies ist zum Beispiel beim *All On*- bzw *All Off*-Dienst interessant, da dort Methoden ausgeführt werden, die auf die gesamte Umgebung wirken.

2. Eine lokale Ausführungsmethode (`execute`), welche für jedes Umgebungselement ausgeführt wird, für die der Dienst aktiv ist. Er bekommt das Dienstobjekt selbst übergeben. So kann der entsprechende Dienst über dieses Dienstobjekt sich aus dem Kontext in der Modellinstanz die States aussuchen, auf die er reagieren muss bzw. selber States anlegen und diese States überwachen. Diese Methode wird allerdings nur für die top-level Dienstobjekte aufgerufen. Für Unterdienste sollte diese Methode von den jeweiligen steuernden Diensten aufgerufen werden.

Bei der Dienstentwicklung ist zu beachten, dass andere Dienste rekonfiguriert werden könnten, so dass fremde States evtl. neu gesucht werden müssen. Auch sollte natürlich überprüft werden, ob der eigene State schon existiert, um im Ausfallfall seinen eigenen Zustand wiederherzustellen.

Obwohl oben erwähnt wurde, dass die Anzahl der Methoden, die eine Dienstkomponente exportiert, möglichst gering sein sollte, um eine möglichst hohe Interoperabilität zu gewährleisten, können sie dennoch genutzt werden. Konzeptionell war es Ziel dieser Arbeit, dem Dienst-Programmierer möglichst viele Freiheiten einzuräumen, um ihm die

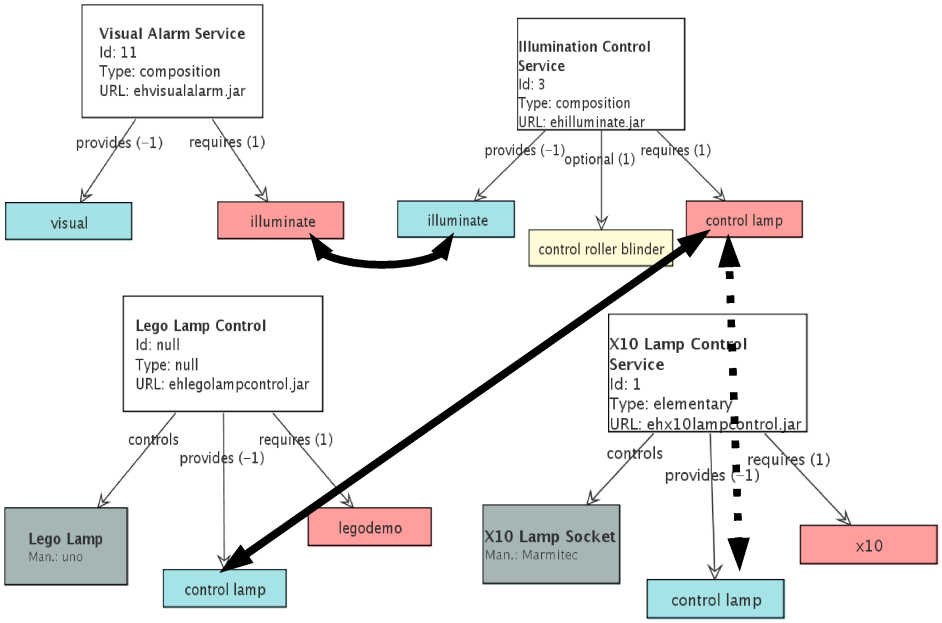

Abbildung 5.14: Ein Beispiel für Funktionalitäten-basierte Komposition

Möglichkeit zu geben, die ihm gewohnten Programmierparadigmen weiter verwenden zu können. Im Gegensatz zu den in [Kir05] vorgeschlagenen Dienstentwicklungen, die den Umgang mit einer nicht imperativen Sprache erfordern, in der das Verhalten eines Dienstes sehr schwer vorherzusehen ist, wird hier auf die imperative Realisierungsmöglichkeit Wert gelegt.

5.5 Automatische Konfigurierung

Um die manuelle Konfigurierung aus Kapitel 4 zu vereinfachen, muss sowohl die Abhängigkeitsauflösung als auch die Parametrisierung unterstützt werden. Dies ist die Hauptaufgabe des Konfigurierungswerkzeugs der eHomeConfigurator-Werkzeug-Suite. Die Idee, einen möglichst einfachen, benutzerfreundlichen und größtenteils automatisierten Konfigurierungsprozess zu erreichen, ist, den Benutzer bei der Konfigurierung Funktionalitäten-getrieben arbeiten zu lassen. Er muss also mit der Auswahl eines oder mehrerer top-level Dienste beginnen. Das Werkzeug soll dann die nötigen und möglichen Unterdienste finden und die Software entsprechend zusammenstellen. Anhand der gewählten Unterdienste können automatisch die benötigten Geräte zusammengestellt werden und auch noch durch den Nutzer festzulegende Attribute bestimmt werden. Das Auffinden der Unterdienste wird hier in der Arbeit *Funktionalitäten-basierte Komposition* genannt. Abbildung 5.14 zeigt ein Beispiel, wie diese funktioniert. Bei der Spezifikation der Dienste, also der Instanzierung der Dienstklassen (siehe Abschnitt 5.2.4), müssen die

Abhängigkeiten in Form von Funktionalitäten (siehe Abschnitt 5.2.1) angegeben werden. Hier sind es die folgenden vier Dienste:

1. `Visual Alarm Service` benötigt die Funktionalität `illuminate` und stellt `raise.alarm.visual` zur Verfügung.
2. `Illumination Control Service` benötigt die Funktionalität `control lamp`, kann auch die Funktionalität `control roller blinder` bedienen und stellt die Funktionalität `illuminate` zur Verfügung.
3. `Lego Lamp Control Service` benötigt die Funktionalität `drive.legodemo`, steuert eine Legolampe (`Lego Lamp`) und stellt die Funktionalität `control lamp` zur Verfügung
4. `X10 Lamp Control Service` benötigt die Funktionalität `drive.x10`, steuert einen X10-Lampensockel (`X10 Lamp Socket`) und stellt auch die Funktionalität `control lamp` zur Verfügung.

Für die Funktionalitäten-basierte Komposition werden nun einfach übereinstimmende benötigte und zur Verfügung gestellte Funktionalitäten entsprechend ihrer Kardinalitäten miteinander verbunden (siehe Pfeile in Abbildung 5.14) und die `uses`-Beziehungen der entsprechenden Dienstobjekte (siehe Abschnitt 5.2.5) gesetzt. Wenn eine Funktionalität von unterschiedlichen Diensten angeboten wird, so kann bisher nicht entschieden werden, welche der beiden besser zu wählen ist und der Benutzer muss gefragt werden. Auf Dauer wäre es hier sinnvoll eine Metrik zu benutzen und die Entscheidung zu bewerten. Dies könnte anhand einer Auswahlstrategie passieren, die der Benutzer vorher festgelegt hat, zum Beispiel immer die kostengünstigste Realisierung zu wählen. Die Komposition über solch einfache Labels durchzuführen, kann auch in Zukunft durch komplexere Übereinstimmungsbedingen ersetzt werden. Bisher werden hier unterschiedliche Interfaces der realisierenden Softwarekomponenten gar nicht berücksichtigt. Es bieten sich deshalb hier auch Kontrakte bzw. sogar parametrische Kontrakte [RHH05, Reu01] an, die Interfaces, deren Aufruf und auch ein gewisses Maß an internem Verhalten der Softwarekomponenten beschreiben.

5.6 Automatisches Deployment

Nach den Rahmenwerksbetrachtungen in Kapitel 4 wurde die erste Version des eHomeConfigurators für das Rahmenwerk OSGi durchgeführt. OSGi bietet zwar mit dem Configuration-Manager eine Möglichkeit zur persistenten Zustandsspeicherung und auch zur Übergabe von Daten an, dennoch unterstützt es kein Typsystem und somit auch nicht die Speicherung von Konfigurationen, die durch das hier vorgestellte eHome-Modell beschrieben werden können. Infolgedessen wird der Configuration-Manager nicht zur Speicherung der Konfigurationsdaten benutzt, sondern das Modell in einer eigenen Softwarekomponente gespeichert.

Die Mechanismen allerdings, die OSGi zum Laden und Starten und Überwachen von Komponenten bietet, eignen sich sehr gut für die Realisierung des Deployers des eHome-Configurators.

Ziel beim Deployen ist es also, die im Konfigurierungsschritt erstellte Konfiguration oder die Laufzeitkonfiguration eines abgestürzten eHome-Systems dem Deployer zu übergeben und diese dann automatisch zu starten und somit das eHome-System in Betrieb zu nehmen. Der Deployer muss dann alle benötigten Softwarekomponenten auf dem Service-Gateway in der richtigen Reihenfolge laden und starten und die beiden besprochenen Initialisierungsroutinen der Komponenten aufrufen. Somit ergibt sich folgende Vorgehensweise für ein automatisches Deployment:

1. Laden der Software-Komponenten, die in den benutzten Diensten angegeben sind.
2. Bei Erfolg, starten der Komponenten über die OSGi Methoden.
3. Ermitteln der Referenzen zur eigentlichen Softwarerealisierung in den Dienstobjekten (siehe `has runtime component` im Abschnitt 5.2.5).
4. Initialisierung der Softwarekomponenten in Bezug auf ihre Abhängigkeiten, also der Aufruf der globalen Initialisierungsmethode (`init`).
5. Aufruf der dienstbasierten Ausfürungsmethode (`execute`), der top-level Dienste.

Die genaue Implementierung ist in Abschnitt 6.7 auf Seite 197 beschrieben.

5.7 Zusammenfassung

Mit diesem Kapitel wurde die konzeptuelle Grundlage für die Implementierung einer Werkzeugsuite zur Unterstützung für die Entwicklung und Konfigurierung von eHome-Systemen geschaffen. Eine erste wichtige Voraussetzung war die Erschaffung des eHome-SCD-Prozesses. Weiterhin grundlegend ist das eHome-Modell, welches in der Lage ist innerhalb des ganzen Prozesses die Konfiguration des eHome-Systems zu beschreiben, von der Geräte-, Umgebungs- und Servicedefinition bis hin zur Laufzeitbeschreibung eines eHome-Systems. Die Umsetzung des Modells und der beschriebenen Konzepte stellt somit eine Möglichkeit der generativen Konfigurierung von eHome-Systemen dar.

Kapitel 6

Implementierung eHomeConfigurator

Dieses Kapitel beschreibt die technische Realisierung und Implementierung des eHome-Configurators.

6.1 Vorgeschichte des eHomeConfigurators

Die Forschungsarbeiten bezüglich des SCD-Prozesses für eHome-Systeme begannen im Jahr 2002. Seit dieser Zeit sind verschiedene Prototypen und Lösungsideen für die Werkzeugunterstützung entwickelt worden. Anfangs gab es eine Werkzeugkette (siehe auch Abbildung 6.1), die aus folgenden Werkzeugen besteht:

1. Protégé [NFM00], ein Open-Source-Werkzeug, welches für die Spezifikation der eHome-Ontologie (siehe Abbildung 6.2) genutzt wurde. Dieses Werkzeug wurde nicht innerhalb der eHome-Gruppe, in der auch diese Arbeit erstellt wurde, entwickelt.

2. *ComponentPreselector* [Kre04], ein Werkzeug, welches die Komponentenauswahl für eHome-Systeme unterstützt. Es handelt sich hierbei um ein Tool welches die Inferierung von Wissen aus der mit Protégé erstellten Wissensdatenbank unterstützt.

3. *DeploymentProducer* [Skr04], ein Werkzeug, welches die initiale Konfiguration importiert und dann durch Interaktion mit dem Benutzer eine Deployment-Konfiguration erstellt und in XML abspeichert.

4. *RuntimeInstancer* [Kli04], ein Werkzeug, welches die Software, die in der Deployment-Konfiguration, des DeploymentProducers beschrieben war, auf dem einem OSGi-Service-Gateway installiert und ausführt.

Dies ist eine eingeschränkte und sehr spezifische Lösung, die sehr auf die gewählten Techniken und den vereinfachten Sicherheitsdienst (siehe Abschnitt 3) zugeschnitten ist.

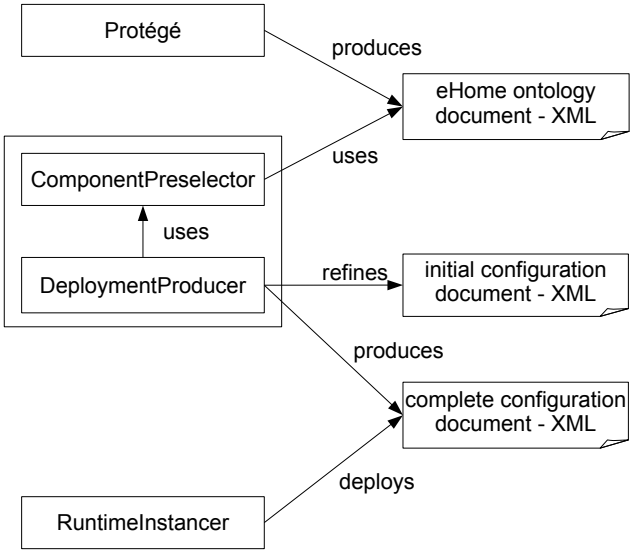

Abbildung 6.1: Die Werkzeugkette, die den SCD-Prozess vor dem eHomeConfigurator-Projekt unterstützte

Die einzelnen Werkzeuge sind zwar über die XML-Datei entkoppelt, erlauben aber nur schwer die Integration und Entwicklung anderer Werkzeuge. Das Modell, welches vom ComponentPreselector benutzt wird, ist eine Wissensdatenbank, die eine sehr spezifische Struktur aufweist, die sich nicht verallgemeinern lässt und somit auch sehr an die verwendeten Werkzeuge gebunden ist. Die Integration der Werkzeuge in der Werkzeugkette wird dadurch erreicht, dass zwischen den einzelnen Schritten dieses XML-Dokument geschrieben bzw. gelesen und transformiert wird, welches die verschieden detaillierten Konfigurationen enthält. Die Dokument-Typ-Definition (DTD) dieses XML-Dokuments ist sehr komplex und sehr spezifisch auf OSGi zugeschnitten. Die Ausgangskonfigurationsdatei muss manuell geschrieben und editiert werden.

Die Werkzeugkette aus Abbildung 6.1 funktioniert wie folgt: Die Wissensbasis (eHome-Ontologie) wird mit Protégé erzeugt und muss für jedes spezifische Heim neu angepasst werden. Die Ausgangskonfiguration, bestehend aus Geräte- und Funktionalitäten-Informationen, wird manuell ohne Werkzeugunterstützung erzeugt. Dieses initiale Ausgangskonfigurationsdokument wird während des SCD-Prozesses durch die Kooperation von DeploymentProducer und ComponentPreselector verfeinert, wobei das letztere Werkzeug die eHome-Ontologie nutzt. Die resultierende Deployment-Konfiguration wird dann auf dem Service-Gateway durch den RuntimeInstancer installiert und zur Ausführung gebracht.

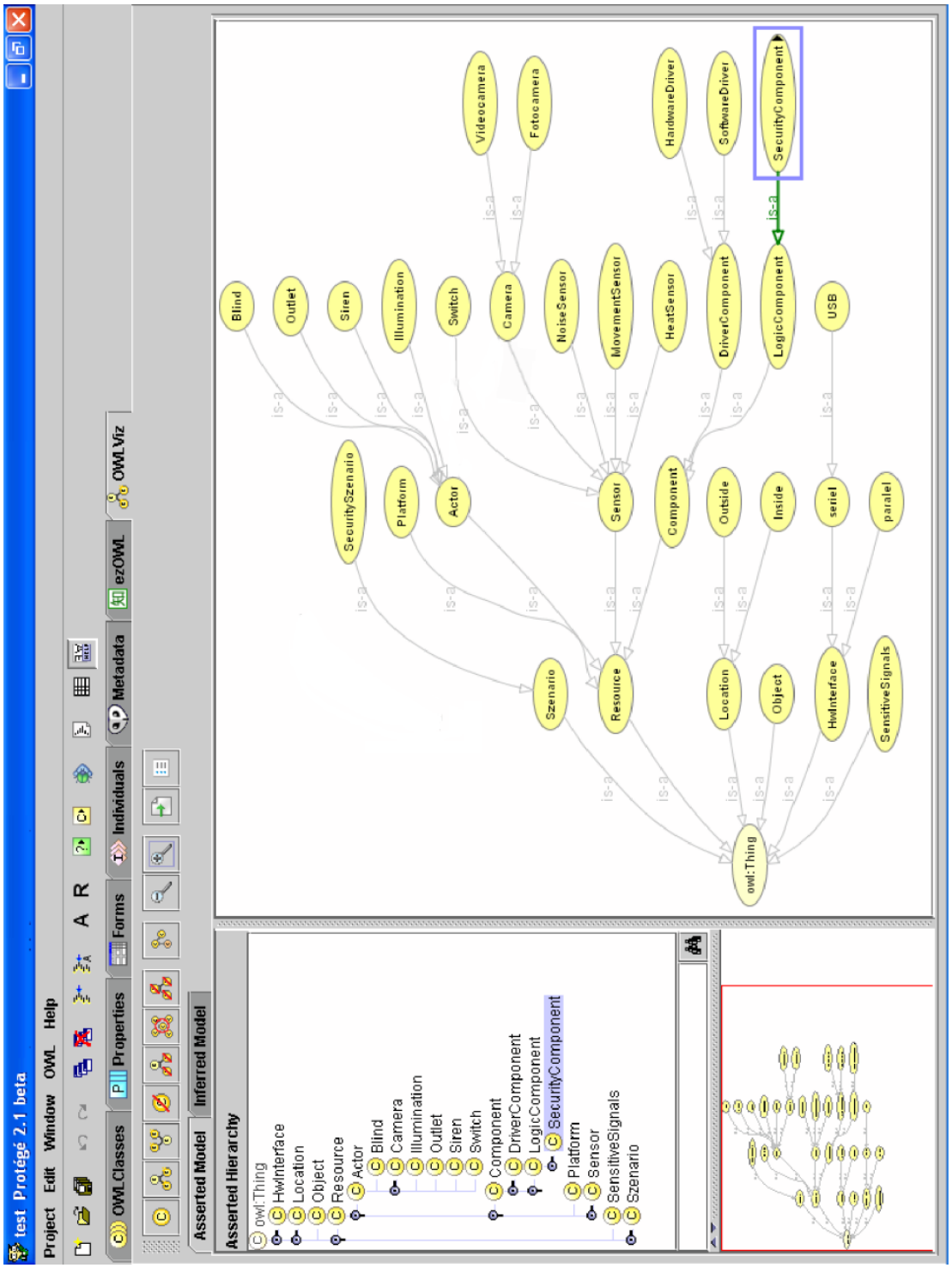

Abbildung 6.2: Alte eHome-Ontologie in Protégé

6.2 Übersicht eHomeConfigurator

Der neue Ansatz nutzt keine Wissensdatenbank-bezogenen Techniken sondern einen klassischen objektorientierten Ansatz und das eHome-Modell, um den SCD-Prozess zu unterstützen. Das eHome-Modell wird mit der *Fujaba*-Werkzeug-Suite [Zün99, NNZ00, KNNZ99] (kurz *Fujaba*) entwickelt. Fujaba ermöglicht die Entwicklung eines objektorientierten Modells und die Erzeugung von komplett ausführbarem Java Code aus diesem. In dem Modell werden sowohl statische wie auch dynamische Zusammenhänge beschrieben. Die statische Struktur des Modells wird mit UML-Klassen-Diagrammen modelliert. In diesen Diagrammen werden die Klassenstrukturen und die Relationen bzw. Assoziationen zwischen den Klassen festgelegt. Zur Laufzeit können die zu den beschriebenen Klassen korrespondierenden Objekte angelegt werden.

Wie bereits im letzten Abschnitt zu sehen, gab es verschiedene andere Versionen des eHome-Modells und dieses unterstützende Werkzeuge vor dem aktuell beschriebenen. Das erste Modell basierte auf der *Ontology-Web-Language* (*OWL*). Die Werkzeuge zur Unterstützung des SCD-Prozesses benutzten deshalb Techniken aus dem Bereich der Wissensverarbeitung (für weitere Informationen zu dieser ersten Version siehe [KNS04]). Durch den Umstieg auf objektorientiertes Modellieren mit Fujaba ergab sich eine erheblich größere Anzahl an Möglichkeiten für die Entwicklung weiterer Werkzeuge zur Unterstützung des SCD-Prozesses. Das objektorientierte Modell ist leichter im Umgang während der Werkzeugentwicklung und hat sich als richtige Wahl herausgestellt.

Die Dynamik bzw. das Laufzeitverhalten des eHome-Modells wird mit den Mitteln der Fujaba-*Story-Diagramme* [FNTZ98] modelliert. Fujaba-Story-Diagramme sind eine Kombination von UML-Aktivitäts-Diagrammen und UML-Kollaborationsdiagrammen. Story-Diagramme können als Aktivitätsdiagramme betrachtet werden, die ihren eigenen Aktivitätstyp, *Story-Aktivität* genannt, besitzen. Diese beschreiben die Interaktion von Objekten während des Programmablaufs und den Programmfluss selbst. Bezogen auf die Klassenmethoden bedeutet das, dass jede Methode in der Klasse durch ein Story-Diagramm beschrieben wird. Durch die Einbettung von Javaanweisungen (was allerdings nur selten nötig ist) und -bedingungen in Story-Diagramme, haben sie dieselbe Ausdrucksstärke wie die Programmiersprache Java selbst.

Durch die Beschreibung des eHome-Modells durch UML-Diagramme ist keine manuelle Kodierung nötig. Nach dem Design des Modells mit Fujaba kann der komplette Java-Code des Modells generiert werden. Der Code ist kompilierbar und setzt das modellierte Modell korrekt um. Die einzigen Probleme, die auftreten können, liegen an semantisch inkorrektem Modellieren der Klassen und ihrer Methoden.

Die Struktur des eHome-Modells ist komplex. Es beinhaltet sechs unterschiedliche Kontexte von der physikalischen Umgebungsstruktur bis zur Laufzeitbeschreibung der einzelnen Dienste (für die einzelnen Kontexte siehe Abschnitt 5.2). Deshalb werden die Modellinstanzen auch nicht in einem zusammenhängenden Bild visualisiert. Die Integration der einzelnen Werkzeuge der eHomeConfigurator-Werkzeugsuite erfolgt über das eHome-Modell selbst. Das bedeutet, dass das eHome-Modell als ein Kommunikationsmedium nicht nur den SCD-Prozess sondern auch das eHome-System zur Laufzeit unterstützt. XML-Parser und -Unparser, sowie komplexe XML-Reflection API-Strukturen sind nicht mehr für die Werkzeugentwicklung nötig.

Das eHome-Modell vereinheitlicht sowohl Datenaustausch als auch -abstraktion über

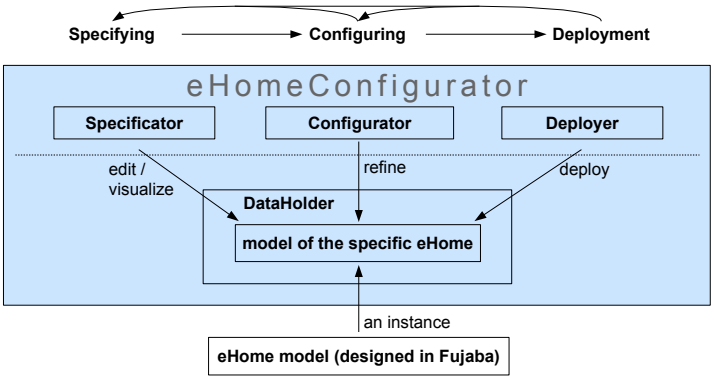

Abbildung 6.3: Übersicht über Aufbau des eHomeConfigurators

ein einfaches Kommunikationsinterface. Das objektorientierte eHome-Modell erlaubt, die unterstützenden Werkzeuge auch noch über den aktuellen Status hinaus später zu erweitern. Die gesamte eHome-Modell-Struktur, also nicht nur die Daten sondern auch die Typinformation, ist für alle Werkzeuge verfügbar, die innerhalb des eHomeConfigurators den SCD-Prozess unterstützen (siehe auch Abschnitt 6.4).

Abbildung 6.3 zeigt den Aufbau der eHomeConfigurator-Werkzeug-Suite. Der eHomeConfigurator besteht aus vier Hauptmodulen: Die drei Module Specificator, Configurator und Deployer unterstützen die jeweils entsprechende Phase im SCD-Prozess. Das vierte Modul, der DataHolder, ist zuständig für die Verkapselung der eHome-Modell-Instanz, der Konfiguration in ihrer jeweiligen Ausprägung, und wird von allen drei Modulen genutzt.

Das Ziel des eHomeConfigurator ist es, den SCD-Prozess zu unterstützen. Das heißt, den Endbenutzern wie den Kunden und den Dienstanbietern eine Schnittstelle zur eHome-Modell-Instanz bzw. der eHome-Konfiguration im hier besprochenen Sinne zur Verfügung zu stellen.

Der eHomeConfigurator bzw. seine einzelnen Werkzeuge werden benutzt, um Veränderungen hin zu einer Deployment-Konfiguration für ein spezifisches eHome durch den ganzen SCD-Prozess hindurch korrespondierend zum eHome-Modell durchzuführen. Weiterhin ist das Werkzeug auch für die Persistenz der Modellinstanz im Rahmen dieses Prozesses zuständig.

Die eHomeConfigurator-Werkzeug-Suite wurde so entwickelt, dass sie als eine eigenständige Applikation ausgeführt werden kann. Dies ermöglicht die Arbeit auf einer eHome-Konfiguration unabhängig vom eHome-Zielsystem, also der Umgebung, in der die Konfiguration deployed wird. Der eHomeConfigurator kann allerdings auch als ein Bundle innerhalb des OSGi-Rahmenwerks gestartet werden, welches die direkte Beeinflussung und Überwachung einer aktiven Konfiguration in einem eHome-System zu dessen Laufzeit erlaubt. Dies macht den eHomeConfigurator zu einem integralen Bestandteil des eHome-Systems selbst. Mit ihm lässt sich eine Visualisierung des aktuellen Zustands des eHome-Systems also der aktuellen Konfiguration des eHome-Systems vornehmen, an der Änderungen direkt durchgeführt werden können. Wurde das System bereits deploy-

ed, so sind diese Änderungen direkt sichtbar. Das heißt, dass dann Zustandsänderungen im eHomeConfigurator (wie das Ändern des Zustands einer Lampe) eine direkte Auswirkung auf das reale physikalische System haben (die Lampe wird tatsächlich ein- oder ausgeschaltet).

6.3 eHome-Modell in Fujaba

Das eHome-Modell wurde mit Fujaba spezifiziert. Die Abbildung 6.4 und zeigen die Gesamtübersicht über den statischen Teil des eHome-Modells. Im dynamischen Teil wurden hauptsächlich die Zugriffsmethoden auf eine Modellinstanz modelliert. Darunter fallen die Methoden für das Anlegen und Initialisieren einzelner Objekte entsprechend des Modells. Methoden für das Verändern und Löschen werden größtenteils automatisch generiert. Allerdings wurden hier auch die Graphsuchemethoden des Konfigurierungswerkzeugs modelliert, die erst in Abschnitt 6.6 erläutert werden.

Listing 6.1 und 6.2 zeigen eine Übersicht aller Methoden, die in Fujaba mittels Story-Diagrammen modelliert wurden. Die Erzeugungs-Diagramme sind alle recht ähnlich aufgebaut. Beispiele für solche Erzeugungsmethoden sind `Location.createLocation` und `Service.addProvidedFunction`. Abbildung 6.6 zeigt das Story-Diagramm, welches für eine bestehende Location eine Sub-Location erzeugt. Mittels der oberen `contains`-Kante und angehängtem durchgestrichenen `otherLocation`-Objekt, welches den übermittelten Namen mit dem gefunden Objektnamen vergleicht, wird überprüft, ob das aktuelle Objekt (`this`) nicht bereits schon eine Sub-Location mit dem übergebenen Namen hat. Wenn tatsächlich keine solche vorhanden ist, werden eine neue `contains`-Kante und ein neues Location-Objekt `newLocation` erzeugt. Die Erzeugung der Kante und des Objekts werden jeweils durch das `<create>` angezeigt. Im neu angelegten Objekt `newLocation` werden der übergebene Name und die Layoutposition der darüberliegenden Location etwas nach rechts unten (deshalb +5) versetzt gesetzt.

Das Story-Diagramm in Abbildung 6.7 beschreibt das Hinzufügen einer neuen angebotenen Funktionalität mit einer gegebenen Kardinalität zu einer Dienstbeschreibung. Hier werden nur das Kardinalitätsobjekt und das Funktionalitätsobjekt und deren Verbindungen angelegt.

`getValue` und `setValue` sind zwei Hilfsfunktionen, um Werte von Attributen zu setzen und auszulesen. Prinzipiell sind als Attribute alle Datentypen möglich (siehe Abschnitt 5.2.2), dennoch werden Integer und Strings am häufigsten verwendet. Deshalb werden diese Hilfsfunktionen hier zur Verfügung gestellt. Für alle anderen Datentypen kann der zugeordnete `ValueHolder` über die `has`-Assoziation adressiert werden.

Abbildung 6.8 zeigt das Story-Diagramm zum Lesen eines Integer- bzw. String-Werts, Abbildung 6.9 analog das Story-Diagramm zum Setzen eines String-Wertes. `setValue` unterstützt noch nicht das Anlegen eines Integer-Wertes. Dafür berücksichtigt es das Vorhandensein des ValueHolder. Dieser wird ggf. neu angelegt. Existiert der ValueHolder bereits, wird er nur verändert.

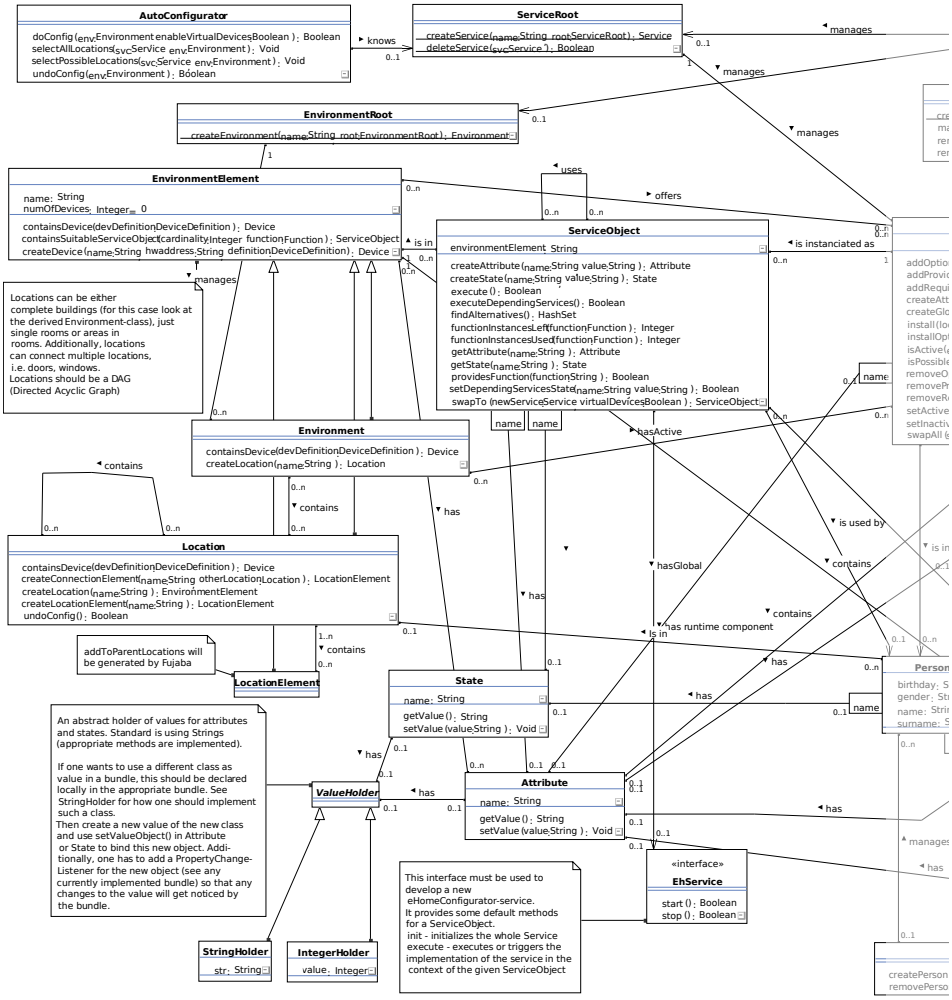

Abbildung 6.4: Das eHome-Modell, Gesamtübersicht (erste Hälfte)

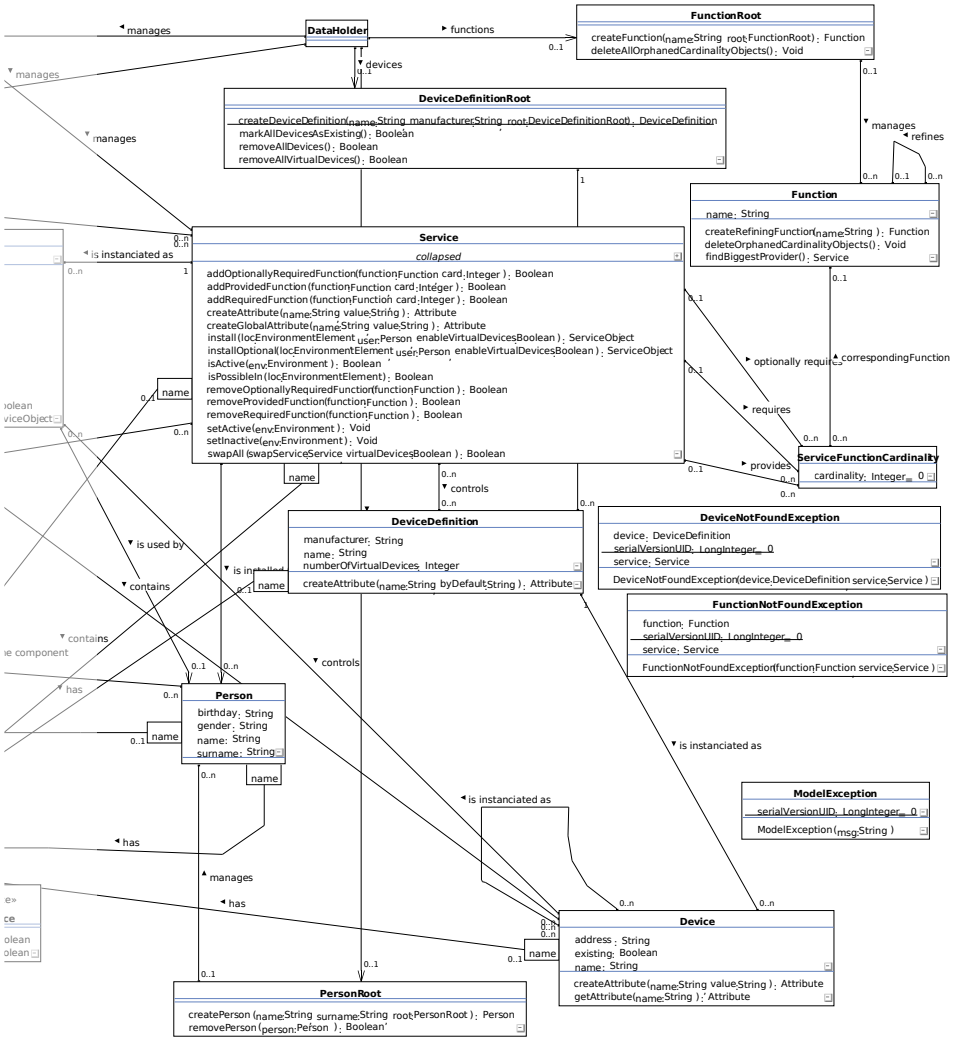

Abbildung 6.5: Das eHome-Modell, Gesamtübersicht (zweite Hälfte)

```
1  class Attribute extends PositionedObject
2     String getValue()
3     void setValue(String value)
4
5  class AutoConfigurator extends eHomeBase
6     boolean doConfig(Environment env)
7     void selectAllLocations(Service svc, Environment env)
8     void selectPossibleLocations(Service svc, Environment env)
9     boolean undoConfig()
10
11 class CategoryRoot extends eHomeBase
12   Category createCategory(String name, CategoryRoot root)
13
14 class Device extends PositionedObject
15    Attribute createAttribute(String name, String value)
16    Attribute getAttribute(String name)
17
18 class DeviceDefinition extends eHomeBase
19    Attribute createAttribute(String name, String byDefault)
20
21 class DeviceDefinitionRoot extends eHomeBase
22    static DeviceDefinition createDeviceDefinition(String name,
23      String manufacturer, DeviceDefinitionRoot root)
24    boolean markAllDevicesAsExisting()
25    boolean removeAllDevices()
26    boolean removeAllVirtualDevices()
27
28 class Environment extends EnvironmentElement
29    Device containsDevice(DeviceDefinition devDefinition)
30    Location createLocation(String name)
31
32 class EnvironmentElement extends PositionedObject
33    Device containsDevice(DeviceDefinition devDefinition)
34    ServiceObject containsSuitableServiceObject(int cardinality,
35      Function function)
36    Device createDevice(String name, String hwaddress,
37      DeviceDefinition definition)
38
39 class EnvironmentRoot extends PositionedObject
40    static Environment createEnvironment(String name,
41      EnvironmentRoot root)
42
43 class Function extends eHomeBase
44    Function createRefiningFunction(String name)
45    void deleteOrphanedCardinalityObjects()
46    Service findBiggestProvider()
47
48 class FunctionRoot extends eHomeBase
49    Function createFunction(String name, FunctionRoot root)
50    void deleteAllOrphanedCardinalityObjects()
51
52 class Location extends EnvironmentElement
53    Device containsDevice(DeviceDefinition devDefinition)
54    LocationElement createConnectionElement(String name,
55      Location otherLocation)
56    EnvironmentElement createLocation(String name)
57    LocationElement createLocationElement(String name)
58
59 class PersonRoot extends eHomeBase
60   Person createPerson(String name, String surname, PersonRoot root)
61   boolean removePerson(Person person)
```

Listing 6.1: Übersicht über alle modellierten Methoden des eHome-Modells

```
1  class Service extends eHomeBase
2     boolean addOptionallyRequiredFunction(Function function, int card)
3     boolean addProvidedFunction(Function function, int card)
4     boolean addRequiredFunction(Function function, int card)
5     Attribute createAttribute(String name, String value)
6     Attribute createGlobalAttribute(String name, String value)
7     ServiceObject install(EnvironmentElement loc)
8     ServiceObject installOptional(EnvironmentElement loc)
9     boolean isActive(Environment env)
10    boolean isPossibleIn(EnvironmentElement loc)
11    boolean removeOptionallyRequiredFunction(Function function)
12    boolean removeProvidedFunction(Function function)
13    boolean removeRequiredFunction(Function function)
14    void setActive(Environment env)
15    void setInactive(Environment env)
16    boolean swapAll(Service swapService)
17
18 class ServiceObject extends eHomeBase
19    Attribute createAttribute(String name, String value)
20    State createState(String name, String value)
21    boolean execute()
22    boolean executeDependingServices()
23    HashSet findAlternatives()
24    int functionInstancesLeft(Function function)
25    int functionInstancesUsed(Function function)
26    Attribute getAttribute(String name)
27    State getState(String name)
28    boolean providesFunction(String function)
29    boolean setDependingServicesState(String name, String value)
30    ServiceObject swapTo(Service newService)
31
32 class ServiceRoot extends eHomeBase
33    static Service createService(String name, ServiceRoot root)
34    static boolean deleteService(Service svc)
35
36 class State extends eHomeBase
37    String getValue()
38    void setValue(String value)
```

Listing 6.2: Übersicht über alle modellierten Methoden des eHome-Modells (zweiter Teil)

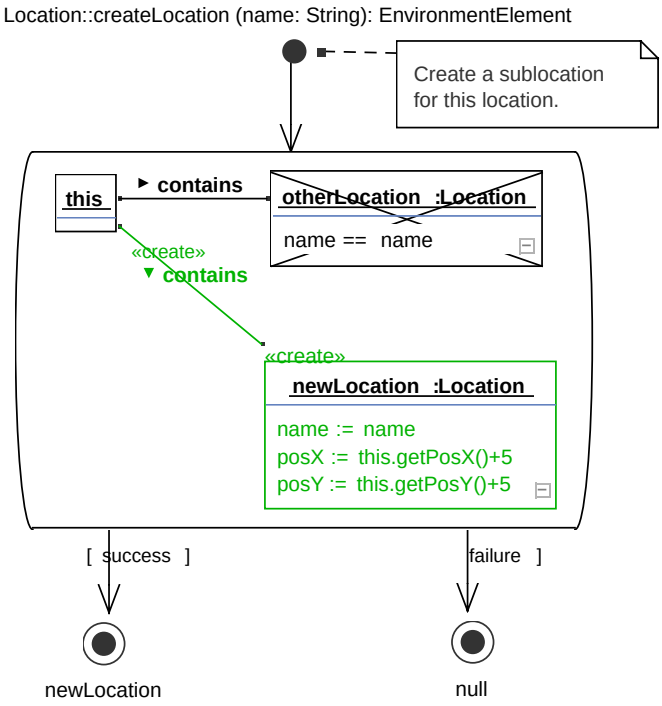

Abbildung 6.6: Story-Diagramm zu `Location.createLocation`

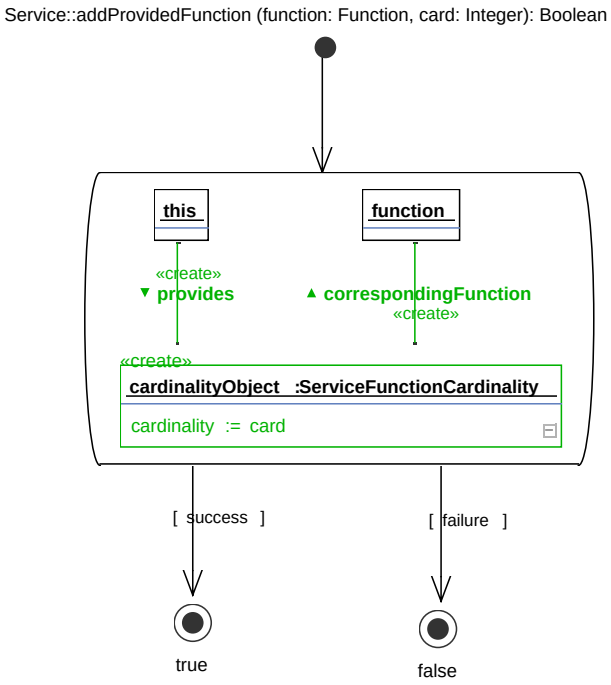

Abbildung 6.7: Story-Diagramm zu `Service.addProvidedFunction`

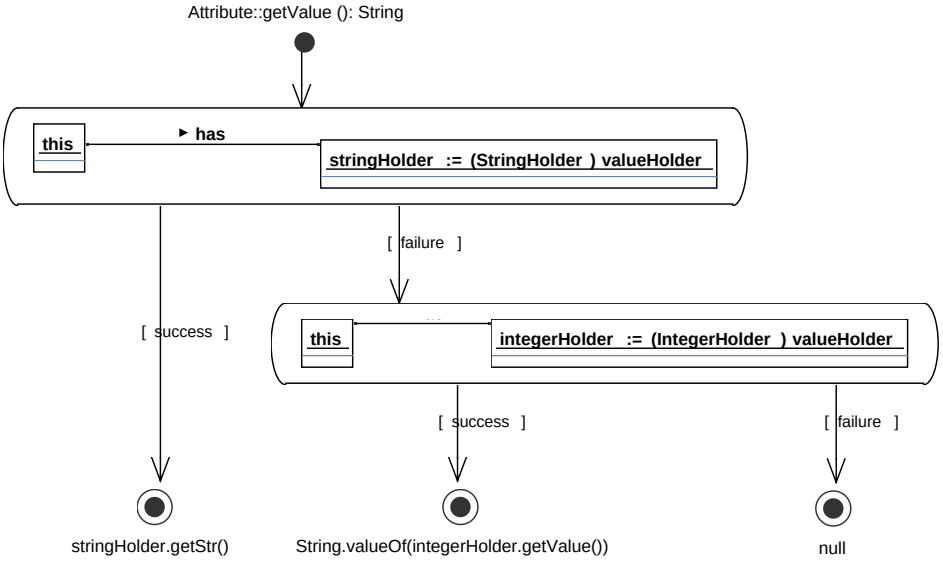

Abbildung 6.8: Story-Diagramm zu `Attribute.getValue`

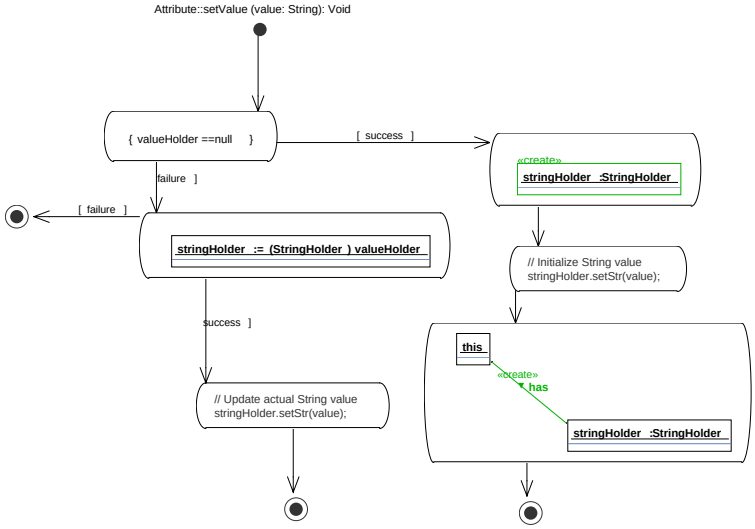

Abbildung 6.9: Story-Diagramm zu `Attribute.setValue`

6.4 DataHolder

Dieser Abschnitt beschreibt das DataHolder-Modul, seine Funktionsweise und Struktur. Der DataHolder fungiert als ein Container[1] für eine Instanz des eHome-Modells. Es speichert also die Konfiguration eines eHome-Systems in unterschiedlichen Ausprägungen. Der DataHolder bietet eine festgelegte Schnittstelle, um auf diese Modellinstanz zuzugreifen. Seine Hauptfunktionen sind

- Speichern der Konfiguration in einer Datei mit einem gewählten Namen,
- Laden der Konfiguration aus einer Datei des Dateisystems,
- Undo- und Redo-Funktionalität für Änderungen an der Konfiguration,
- und die zur Verfügungsstellung eines Einstiegspunktes für Zugriffe auf die Konfiguration.

Diese Funktionen werden implementiert, indem sie die entsprechenden Funktionen des CoObRA-Rahmenwerks [Sch03] nutzen. Im Weiteren werden im Einklang mit dem Sprachgebrauch im CoObRA-Kontext die Speicher- und Versionierungsfähigkeiten mit dem Begriff der CoObRA-Persistenz bezeichnet. Das CoObRA-Rahmenwerk liefert auch eine Überwachungsfähigkeit der Änderungen der Daten in der Modellinstanz entsprechend der JavaBeans-Specification [Sun97] bzw. des Observer-Patterns [GHJV95]. Dies erlaubt es den eHome-Diensten, gewisse Teile der Konfiguration zu überwachen, sich also bei einem gewissen Teil der Modell-Instanz anzumelden, um über deren Änderungen informiert zu werden. Das heißt in der Sprache von JavaBeans, einen Property-Change-Listener auf ein Objekt zu installieren, der ein Event im Falle der Änderung auslöst und so eine unmittelbare Reaktion auf eine Änderung ermöglicht. So kann die Konfiguration selbst als Kommunikationsmedium zwischen den Diensten benutzt werden.

Ein weiterer Grund für den Einsatz Fujabas für diese Arbeit ist die Unterstützung von CoObRA in Fujaba: CoObRA ist so in Fujaba integriert, dass Fujaba sehr einfach Code für CoObRA-persistente Klassen erzeugen kann. Dies ist von besonderer Bedeutung, da trotz der mächtigen Fähigkeiten des CoObRA-Rahmenwerks, die Ausnutzung dieser, eine gewisse Programmierdisziplin erfordert. Die Entwicklung CoObRA-persistenter Klassen ist nicht so intuitiv, wie es vielleicht für jeden Entwickler angenehm wäre, sondern recht komplex und fehleranfällig, wenn sie manuell durchgeführt wird. Die Entwicklung der Klassen muss der CoObRA-Spezifikation, insbesondere bezüglich des Property-Change-Mechanismus, genügen. Erfreulicherweise kommt der Fujaba-Entwickler nicht mit dieser Komplexität in Berührung, da Fujaba den Code der in Fujaba modellierten Klassen konform der CoObRA-Spezifikation generieren kann, so dass sich der Aufwand, eine Klasse CoObRA-persistent zu, auf einen Klick reduziert. Der von Fujaba generierte Code enthält dann automatisch Code, der den Property-Change-Mechanismus unterstützt und so die Grundlage für die Dienstekommunikation in dieser Arbeit bildet.

Der DataHolder wurde so entwickelt, dass er von unterschiedlichen Werkzeugen des eHomeConfigurators zur gleichen Zeit unterschiedlich genutzt werden kann. Wird der eHomeConfigurator als OSGi-Bundle gestartet, so gilt folgendes:

[1] Container wird hier im Sinne einer Datenstruktur verwendet, die eine größere Menge an strukturierten Daten aufnehmen kann.

1. Das Spezifizierungs-Werkzeug (siehe Abschnitt 6.5) und das Deployment-Werkzeug (siehe Abschnitt 6.7) können die eHome-Modell-Instanz gleichzeitig benutzen.

2. Die eHome-Dienste können auf die eHome-Modell-Instanz zugreifen, da auch sie OSGi-Bundles sind.

3. Verschiedene Spezifikationswerkzeuge können gleichzeitig über denselben Data-Holder auf dieselbe eHome-Modell-Instanz zugreifen. So kann ein Kunde bei der Benutzung des Werkzeugs von einem Provider im Problemfall dabei unterstützt werden (dieses Verfahren wird in [Kir05] untersucht und als Erweiterung zum eHomeConfigurator implementiert).

CoObRA ist ein Objekt-Repository, welches wie ein Concurrent Versioning System (wie CVS [Xim] oder SVN [Com00]) funktioniert. Es wird also Versionierung und das verteilte Arbeiten auf einer objektorientierten Datenbank unterstützt. Diese Funktionalität wird bisher in dieser Arbeit noch nicht verwendet, bietet aber interessante Erweiterungsmöglichkeiten im Rahmen der Provider- zu Provider- und Provider- zu Kundenkommunikation.

6.5 Spezifizierungs-Werkzeug (Specificator)

Das *Spezifizierungs-Werkzeug* (eng.: *specificator*) ist die Benutzerschnittstelle für die gesamte eHomeConfigurator-Werkzeug-Suite. Es unterstützt die Spezifizierungsphase des SCD-Prozesses. Zuerst wird hier die graphische Benutzerschnittstelle des Spezifizierungs-Werkzeugs beschreiben. Anschließend werden wichtige Struktureigenschaften vorgestellt.

Die graphische Benutzerschnittstelle (GUI) des Spezifizierungs-Werkzeugs ist mit dem Java-Swing-*Application-Programming-Interface* (API) umgesetzt. Die GUI besteht aus einem Hauptapplikationsrahmen (eng.: main application frame), einer Menüzeile (eng.: menu bar), einer Werkzeugleiste (eng.: tool bar) und einem Bedienfeld mit Reitern (eng.: tabbed panel). Siehe für diesen Aufbau Abbildung 6.10. Im Menü befindet sich eine allgemeine Auswahl für Befehle wie Redo, Undo, Speichern, Laden oder das Hinein- und Herauszoomen in der aktuellen Ansicht. Über die Werkzeugleiste sind hauptsächlich dieselben Befehle zu erreichen. Es gibt ein paar zusätzliche Befehle: dort befinden sich auch die Knöpfe, die die Aktionen des Konfigurierungs-Werkzeugs für die Konfigurierungsphase auslösen (siehe Abschnitt 6.6). Weiterhin befinden sich dort der Knopf, der das Deployment-Werkzeug startet (siehe Abschnitt 6.7), und ein Knopf welcher das *Dynamic Object Browsing System* (*DOBS*) [GZ02] startet, welches das Debuggen der eHome-Modell-Instanz ermöglicht. Für die Benutzung des DOBS siehe Abschnitt 7.7 auf Seite 238.

Die wichtigsten Komponenten der GUI sind die Reiter und die zugehörigen Bedienfelder. Dort befinden sich die notwendigen Editor- und Information-Panels für die Manipulation der Konfiguration. Es gibt zwei unterschiedliche Arten von Bedienfeldern: *Editorfelder* (eng.:*editor tabs*, wie auf Abbildung 6.10) und *Informations-Felder* (eng.: *information tabs*, vergleiche Abbildung 6.11).

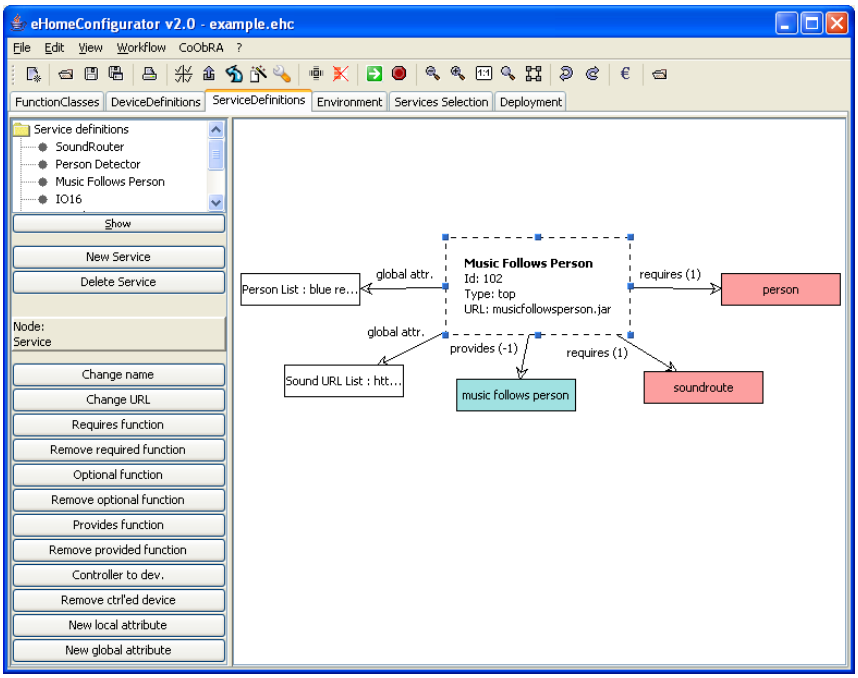

Abbildung 6.10: Bildschirmfoto des eHomeConfigurators – Spezifizierungs-Werkzeug

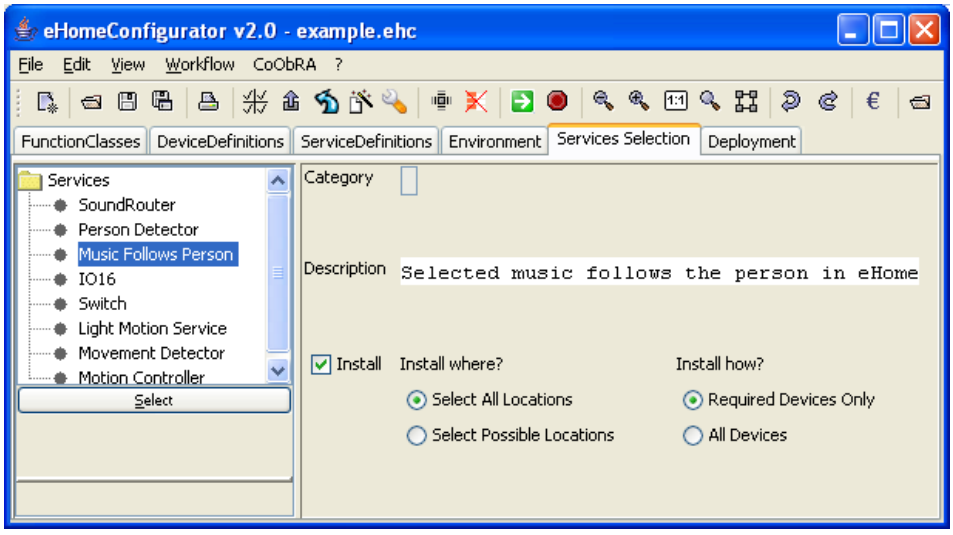

Abbildung 6.11: Bildschirmfoto des eHomeConfigurators – Informations-Feld

Für jeden eHome-Modell-Kontext (siehe Abschnitt 5.2) außer dem Personenkontext existiert ein Reiter mit einem korrespondierenden Editor, um diesen Kontext zu bearbeiten. Somit gibt es fünf verschiedene Editoren. In Abbildung 6.10 ist der MusicFollowsPerson-Dienst im Bedienfeld des Dienst-Kontext (siehe Abschnitt 5.2.4) zu sehen.

Zur Visualisierung der Kontexte in den einzelnen Editor-Feldern wird JGraph [Com] verwendet. Es wäre prinzipiell möglich, DOBS für die Visualisierung zu benutzen, doch lässt DOBS keine Einschränkung der Sichten zu und bietet nur rudimentäre graphische Ausgestaltungsmöglichkeiten. Hier entstünde schnell, wie bereits in Abbildung 7.42 zu erahnen ist, eine gewisse Unübersichtlichkeit und somit evtl. eine fehlerhafte Bedienung. Bei der Konstruktion des eHomeConfigurators war es wichtig, eine Werkzeug-Suite zu realisieren, die von unterschiedlichen Personengruppen wie Kunden, Providern, Entwicklern oder Servicetechnikern bedient werden kann. Weiterhin bietet DOBS auch einige Darstellungsformen nicht an, die in JGraph vorhanden sind. So bietet JGraph verschiedene Layoutalgorithmen und zusätzliche Möglichkeiten zum Graphlayout an. Leider ist JGraph zum Zeitpunkt der Fertigstellung dieser Arbeit keine offene Software mehr, so dass evtl. eine Umstellung auf GEF [Con06], wie es in der Entwicklungsumgebung Eclipse verwendet wird, in Zukunft sinnvoll sein könnte.

Die Entwicklung eines Editors besteht hauptsächlich aus zwei Schritten:

1. Zuerst müssen die Translatoren implementiert werden, die die eHome-Modell-Instanz entsprechend dem Modell-Kontext des Editors traversieren. Der Translator visualisiert den Modell-Kontext in einem Graph-Panel im Editor.

2. Weiterhin müssen die Methoden (im Kontext dieser Arbeit auch *Aktivitäten*, eng. *activities*, genannt), die für die Transformationen der Modell-Instanz wichtig sind, dem Benutzer zugängig gemacht werden. Dies geschieht durch die Erstellung eines Knopfes im Editor für jede benötigte Aktivität und der Erzeugung von Einträgen eines Kontextmenüs, welches beim Druck auf die rechte Maustaste auf ein Objekt der Modell-Instanz erscheint. Bei Druck auf den erzeugten Knopf im Editor oder den Kontextmenüeintrag muss, wenn Parameter für die Aktivität erforderlich sind, ein entsprechend passender Eingabedialog angezeigt werden, welcher Standardknöpfe zur Bestätigung und zum Abbrechen der Aktivität beinhaltet.

Das Layout der unterschiedlichen Editoren ist größtenteils dasselbe. Die Baumansicht und die Knopfleiste befinden sich auf der linken Seite des Bedienfeldes. Erscheint ein Eingabedialog, so wird dieser oben direkt unter der Werkzeugleiste dargestellt. Innerhalb des Eingabedialogs können Beziehungen zu Listen aus andern Kontexten hergestellt werden, die in einem eigenen Fenster angezeigt werden können (vergleiche Abbildung 6.12).

Das heißt, dass die Entwicklung eines neuen Editors bedeutet, eine Klasse von der Klasse `EditorPanel` abzuleiten und in diesem abgeleitete Klassen von `GraphPanel` und für JTrees von `JPanel` abzuleiten. Die Translatoren für das JGraph-Panel und gegebenenfalls für das JTree-Panel werden gewöhnlich von der Klasse `Translator` abgeleitet. Diese Klassen werden durch den Graph-Traversierungscode vervollständigt. Die Aktivitäten-Knöpfe und korrespondierenden Eingabedialoge werden dynamisch durch den Generic-Activity-Invocation-Mechanismus erzeugt. Die Erstellung der Translatoren

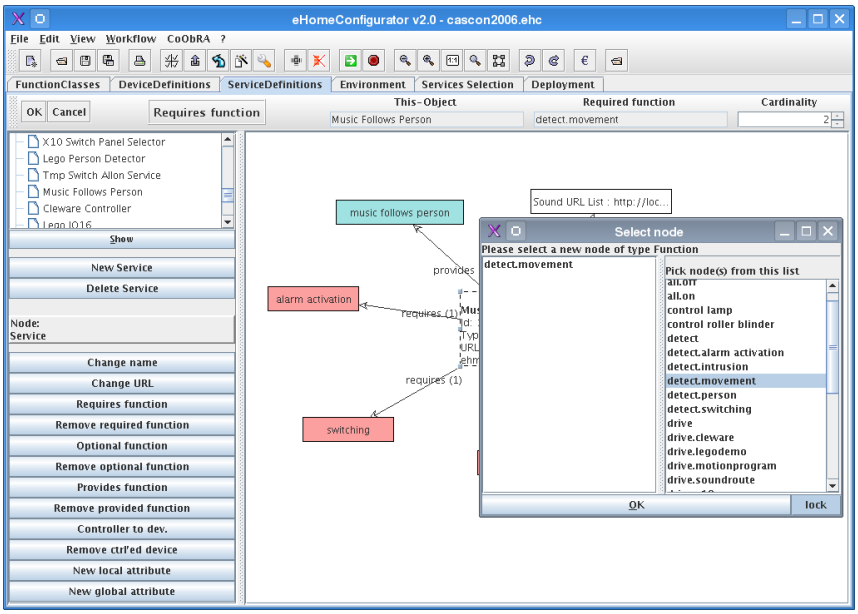

Abbildung 6.12: Bildschirmfoto des eHomeConfigurators – Eingabedialog und Auswahl für Feld des Eingabedialogs

und der Generic-Activity-Invocation-Mechanismus werden in den nächsten Abschnitten (6.5.1 und 6.5.2) behandelt.

Die Entwicklung eines Editors kann als eine Anwendung des Rahmenwerks angesehen werden, welches entsprechend der *framework design principles* aus [GHJV95] gestaltet wurde. Das in allen Editoren übereinstimmende Verhalten ist in abstrakten Klassen generalisiert. Diese lassen sich durch Ableitung zu einem speziellen Editor zusammensetzen.

Somit erfordert die Entwicklung eines bestimmten Editors für den eHomeConfigurator das Erstellen von Unter-Klassen der Panels und Translatoren und die Implementierung des speziellen Verhaltens in Unterklassen.

6.5.1 Translatorprogrammierung

Die Translatoren sind die Brücken zwischen dem eHome-Modell und anderen Java-Technologien. Sie werden genutzt, um JGraph, JTree oder OWL-Strukturen entsprechend den Strukturen der eHome-Modell-Instanz aufzubauen (siehe Abbildung 6.13). Die Translatoren werden hauptsächlich zur Visualisierung der eHome-Modell-Instanz benutzt. Das heißt, sie übersetzen die eHome-Modell-Instanz-Strukturen in JGraph- oder JTree-Strukturen. Im Moment werden also meist zwei Translatoren für einen Editor benötigt: JGraph-Translator und JTree-Translator. Diese müssen manuell kodiert werden. Möglichkeiten, wie diese manuelle Entwicklung vermieden werden kann, werden in [Sal05] diskutiert. Neben dem Einsatz der Translatoren für Editoren, können sie also

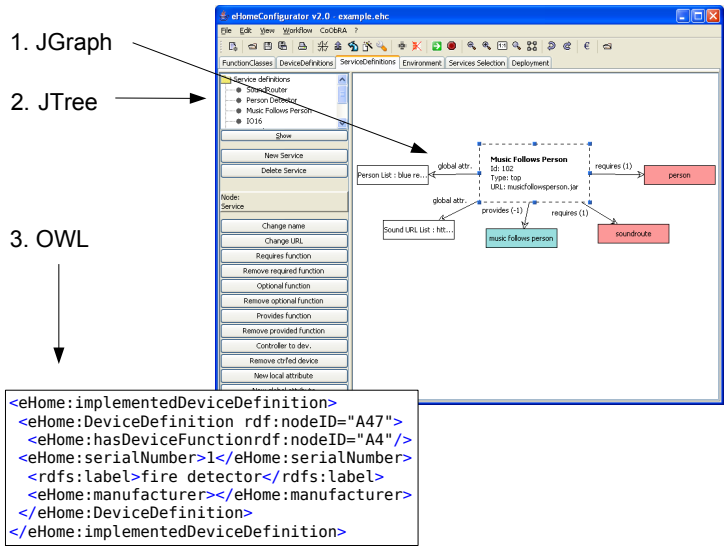

Abbildung 6.13: Verschiedene Translatoren im eHomeConfigurators

auch für die Erzeugung von OWL-Strukturen eingesetzt werden. Dazu wurde das Jena-Version-2-Semantic-Web-API [McB02, McB04] verwendet. In einem frühen Stadium des eHomeConfigurators wurden diese OWL-Translatoren dazu verwendet, weiterhin die alten Werkzeuge ComponentPreselector [Kre04] und DeploymentProducer [Skr04] einsetzen zu können, bevor das XML-Dokument vollständig durch das jetzige eHome-Modell und den DataHolder ersetzt wurde.

Die gesamte Translatorenentwicklung folgt dem Ansatz des *template design pattern* [GHJV95]. Das bedeutet, dass die allgemeinen Eigenschaften und das allgemeine Verhalten der Translatoren ohne spezifisches Verhalten der einzelnen Translatoren in einer `Translator`-Super-Klasse zusammengefasst wurden. Deshalb wird jeder neue Translator von der Klasse `Translator` abgeleitet und implementiert eine `protected abstract void construct()`-Methode, um die entsprechenden relevanten Teile der eHome-Modell-Instanz zu traversieren. Zum Beispiel gibt es im Environment Editor eine Klasse `EnvironmentTranslator`, welche die spezifischen Traversierungsroutinen in der `construct()`-Methode implementiert, um den Umgebungs-Kontext der eHome-Modell-Instanz zu bearbeiten.

Außerdem stellt die Klasse `Translator` auch einige nützliche Methoden für die Entwicklung spezifischer Translatoren zur Verfügung und kapselt einen Teil der Komplexität des JGraph API. Der Aufwand für die Entwicklung neuer Translatoren ist somit gering. Nach Messungen der Entwicklungszeit ist ein solcher von einem Programmierer mit einem Aufwand von weniger als einem Manntag zu erledigen.

Ein neuer Editor verlangt in der Regel also die Entwicklung von drei oder fünf Klassen. Es sind fünf Klassen, wenn ein Graph-Panel und ein Tree-Panel genutzt werden sol-

```
1  package ehome.specificator.ui.editor.service;
2
3          ...
4
5  /**
6   * The Panel displaying the service editor. The underlying panel for
7   * other swing components.
8   *
9   * @author Priit
10  */
11 public class ServiceEditorPanel extends EditorPanel {
12
13     private ServiceJTreePanel serviceJTreePanel;
14
15     /**
16      * Constructor for ServiceEditorPanel - the underlying panel of
17      * ServiceComponentEditor
18      * @param config - the configurator to work for
19      */
20     public ServiceEditorPanel(ConfiguratorGUI config) {
21         super(config);
22         ServiceGraphPanel graph= new ServiceGraphPanel(kernel,this);
23         setGraphPanel(graph);
24         setTreeComponent(serviceJTreePanel =
25             new ServiceJTreePanel(kernel, graph));
26     }
27
28     /**
29      * @return Returns the softwareComponentJTreePanel.
30      */
31     ServiceJTreePanel getServiceJTreePanel() {
32         return serviceJTreePanel;
33     }
34 }
```

Listing 6.3: Klasse `ServiceEditorPanel`

len, und drei, wenn nur ein Graph-Panel genutzt werden soll. Der Service-Editor ist zum Beispiel mit fünf Klassen realisiert worden. Siehe hierfür die Listings 6.3, 6.4, 6.5, 6.6 und 6.7.

6.5.2 Der Generic-Activity-Invocation-Mechanismus

Die Idee des *Generic-Activity-Invocation-Mechanismus* (*GenAIM*) wurde durch das Upgrade-Framework [Jäg00, BJSW02] inspiriert, welches einen ähnlichen Method-Invocation-Mechanismus implementiert. Der Mechanismus wurde in [NSSK05] besprochen. Der GenAIM im Spezifizierungswerkzeug hat das Ziel die Aktivitäten des eHome-Modells, das heißt die Methoden der Klassen des Modells, in der eHomeConfigurator-GUI verfügbar zu machen.

Dies geschieht durch die dynamische Erzeugung der entsprechenden Knöpfe und Eingabefelder in der GUI. Die Angabe der aufrufbaren Aktivtäten geschieht innerhalb einer XML-Datei. Diese Datei wird beim Start des eHomeConfigurators ausgelesen und geparst. Die Knöpfe werden dann zur Laufzeit abhängig vom gerade ausgewählten Editor und Objekt entsprechend der Beschreibung in der Datei erzeugt. Der eigentliche Aufruf der Aktivität, also die Method-Invocation, wird mittels der Java-Reflection-API [FF04]

```
1  package ehome.specificator.ui.editor.service;
2    ...
3  public class ServiceGraphPanel extends GraphPanel {
4
5    public ServiceGraphPanel(Kernel k, ServiceEditorPanel editor) {
6      super(k, editor);
7    }
8
9    /** Method assinging the service translator for translator object
10    *  used in the super classes template constructor */
11   protected void initTranslator() {
12     Service serv =
13       getKernel().getDataHolder().getFirstService();
14     setTranslator(new ServiceTranslator(serv));
15   }
16
17   public ActivityContext getEditorContext() {
18     return ActivityDescriptor.CONTEXT_SERVICE;
19   }
20
21   public void resetModel(Service serv) {
22     ((ServiceTranslator)getTranslator()).setService(serv);
23         refreshModel(false);
24         getEditorPanel().layoutGraphPanel();
25   }
26
27     public void refreshModel(boolean complete) {
28         super.refreshModel(complete);
29         ServiceJTreePanel treepanel= ((ServiceEditorPanel)
30           getEditorPanel()).getServiceJTreePanel();
31         if (treepanel != null) {
32             treepanel.resetTree();
33         }
34     }
35
36     /** Determines a complete graph view for this editor by
37      *  constructing new transl. and query foreach devicedefinition.
38      *  @return a set of nodes that might all be displ. in editor */
39     public Set getAllFujabaNodes() {
40         Set nodes= new HashSet();
41         ServiceTranslator trans= new ServiceTranslator(null);
42
43         for (Iterator devIt= getKernel().getDataHolder().
44           getServices().iterator(); devIt.hasNext();) {
45             trans.setService((Service)devIt.next());
46             trans.completeUdateGraph();
47             nodes.addAll(trans.getVertices().keySet());
48         }
49         return nodes;
50     }
51 }
```

Listing 6.4: Klasse `ServiceGraphPanel`

```
1  package ehome.specificator.ui.editor.service;
2    ...
3  /** This panel contains the JTree on the JScrollPane. The tree
4   *  presents all service definitions and their structure. */
5  public class ServiceJTreePanel extends JPanel {
6    private final Kernel kernel;
7    private final ServiceJTreeTranslator translator;
8    private final ServiceGraphPanel graphParent;
9    private final JTree tree;
10
11   public ServiceJTreePanel(Kernel k, ServiceGraphPanel graphParent)
12   {
13     this.kernel = k;
14     this.graphParent = graphParent;
15     this.translator = new ServiceJTreeTranslator(); // Get transl.
16     this.tree = new JTree(); // Prepare the JTree
17     // Set single selection on the tree
18     tree.getSelectionModel().setSelectionMode(
19       TreeSelectionModel.SINGLE_TREE_SELECTION);
20     // Add MouseListener to tree
21     MouseListener ml = new MouseAdapter() {
22       public void mousePressed(MouseEvent e) {
23         int selRow = tree.getRowForLocation(e.getX(), e.getY());
24         TreePath selPath =
25           tree.getPathForLocation(e.getX(), e.getY());
26         if (selRow != -1) {
27           if (e.getClickCount() == 2) {
28             userDoubleClicked(selPath);
29     } } } };
30
31     tree.addMouseListener(ml);
32         tree.setVisibleRowCount(5);
33     JButton showButton= new JButton("Show");
34     showButton.addActionListener(new ActionListener() {
35       public void actionPerformed(ActionEvent e) {
36         userDoubleClicked(tree.getSelectionPath());
37     } });
38     showButton.setFont(showButton.getFont().
39       deriveFont(showButton.getFont().getSize()-2F));
40     showButton.setMnemonic('s');
41     resetTree(); // Set the tree model
42     // Assemble the GUI parts
43     this.setLayout(new BorderLayout());
44     this.add(new JScrollPane(tree), BorderLayout.CENTER);
45     this.add(showButton, BorderLayout.SOUTH); }
46
47   public void resetTree() {
48     // Get the services listing
49     Vector services = kernel.getDataHolder().getServices();
50     // Construct the tree model
51     tree.setModel(
52       new DefaultTreeModel(translator.construct(services), false));}
53
54   private void userDoubleClicked(TreePath p) {
55     if (((DefaultMutableTreeNode)p.getLastPathComponent())
56       .getUserObject() instanceof ModelWrapper) {
57       ModelWrapper userObject =
58         (ModelWrapper) ((DefaultMutableTreeNode)
59           p.getLastPathComponent()).getUserObject();
60       if (userObject.getElement() instanceof Service) {
61         graphParent.resetModel((Service)userObject.getElement());
62   } } }
63
64   public ServiceJTreeTranslator getTranslator() {return translator;}
65
66   public Hashtable getFujabaToJTreeMapping() {
67     return this.getTranslator().getNodes();  }
68 }
```

Listing 6.5: Klasse `ServiceJTreePanel`

```
1  package ehome.specificator.ui.editor.service;
2    ...
3  /** The translator class for service definition editor */
4  public class ServiceTranslator extends Translator {
5    private static String REQUIRES = "requires";
6    private static String OPTIONAL = "optional";
7    private static String PROVIDES = "provides";
8    private Service service;
9
10   public ServiceTranslator(Service service) { setService(service); }
11
12   public void setService(Service serviceDef) {
13     this.service = serviceDef; }
14
15   protected void construct() {
16     if (service != null) { addService(); } }
17
18   private void addService() {
19     DefaultGraphCell v =
20       this.addRootVertexToGraph(new ModelWrapper(service), 3,
21       Color.WHITE);
22     // Handling functions
23     // required
24     for (Iterator required=service.
25         iteratorOfCardinallyRequiredFunction(); required.hasNext();)
26     {
27       ServiceFunctionCardinality srv =
28         (ServiceFunctionCardinality) required.next();
29       addFunction(v, srv.getFunction(), REQUIRES + " (" +
30         srv.getCardinality() + ")"); }
31     // optionally required
32     for (Iterator optional =
33         service.iteratorOfCardinallyOptionallyRequiredFunction();
34         optional.hasNext();) {
35       ServiceFunctionCardinality srv = (ServiceFunctionCardinality)
36         optional.next();
37       addFunction(v, srv.getFunction(), OPTIONAL + " (" +
38         srv.getCardinality() + ")"); }
39     // provided
40     for (Iterator provided=service.
41         iteratorOfCardinallyProvidedFunction(); provided.hasNext();)
42     {
43       ServiceFunctionCardinality srv =
44         (ServiceFunctionCardinality) provided.next();
45       addFunction(v, srv.getFunction(), PROVIDES + " (" +
46         srv.getCardinality() + ")"); }
47     // adding device definitions
48     for (Iterator deviceDefs =
49         service.iteratorOfControlledDeviceDefinitions();
50       deviceDefs.hasNext();) {
51       DeviceDefinition devDef = (DeviceDefinition)deviceDefs.next();
52       addVertexToGraph(v, new ModelWrapper(devDef),
53         "controls", 2, COLORDEVICEDEFINITION); }
54     // adding attributes
55     for (Iterator attributes = service.iteratorOfAttributes();
56         attributes.hasNext();) {
57       Attribute attribute = (Attribute) attributes.next();
58       addVertexToGraph(v, new ModelWrapper(attribute), "attr."); }
59     for (Iterator globals = service.iteratorOfGlobalAttributes();
60         globals.hasNext();) {
61       Attribute global = (Attribute) globals.next();
62       addVertexToGraph(v, new ModelWrapper(global), "global attr.");
63   } }
64
65   private void addFunction(
66     DefaultGraphCell parent,
67     Function f, String type) {
68     String colorString = type.substring(0, type.indexOf("(") - 1);
69     Color color = colorString.equals(REQUIRES) ?
70        COLOR_REQUIRED_FUNCTION : colorString.equals(OPTIONAL) ?
71          COLOR_OPTIONAL_FUNCTION : COLOR_PROVIDES_FUNCTION;
72     DefaultGraphCell v = this.addVertexToGraph(parent,
73       new ModelWrapper(f), type, 1, color);
74 } }
```

Listing 6.6: Klasse `ServiceTranslator`

```
1  package ehome.specificator.ui.editor.service;
2    ...
3  /** Translator which translates the ServiceDefintion part of Fujaba
4   *  model to a JTree structure. */
5  public class ServiceJTreeTranslator {
6    private DefaultMutableTreeNode root;
7    private Hashtable nodes = new Hashtable();
8
9    public ServiceJTreeTranslator() {}
10
11   public DefaultMutableTreeNode construct(Vector services) {
12     nodes.clear();
13     root = new DefaultMutableTreeNode("Service definitions");
14
15     for (int i = 0; i < services.size(); i++) {
16       Service service = (Service)services.elementAt(i);
17       DefaultMutableTreeNode v = addNodeToTree(root,
18         new ModelWrapper(service, false));
19       Iterator j = service.iteratorOfRequiredFunctions();  }
20     return root; }
21
22   private void addFunction( DefaultMutableTreeNode parent,
23       Function f) {
24     // Add requirement list
25     DefaultMutableTreeNode v = addNodeToTree(parent,
26       new ModelWrapper(f, false)); }
27
28   /** Adds a node to tree */
29   public DefaultMutableTreeNode addNodeToTree(
30     DefaultMutableTreeNode parent, ModelWrapper child) {
31     DefaultMutableTreeNode v = new DefaultMutableTreeNode(child);
32     nodes.put(child.getElement(), v);
33     parent.add(v);
34     return v; }
35
36   public Hashtable getNodes() { return nodes; }
37 }
```

Listing 6.7: Klasse `ServiceJTreeTranslator`

```
<CLASS name="Environment">
    <ACTIVITY name="newEnvironment" label="New Environment">
        <TOOLTIP>Create new environment</TOOLTIP>
        <CONTEXTS>
            <ENVIRONMENT_EDITOR/>
        </CONTEXTS>
        <PARAM label="Name">
            <TOOLTIP>The name of the environment.</TOOLTIP>
        </PARAM>
        <PARAM label="Environment root">
            <VALUE type="fixed">
                <REF key="ENVIRONMENTROOT"/>
            </VALUE>
        </PARAM>
    </ACTIVITY>
    ...
</CLASS>
```

Listing 6.8: Ein Beispiel für die XML-Einträge in activities.xml für eine Aktivität

durchgeführt. Im Folgenden wird zuerst der Aufbau dieser XML-Datei beschrieben, anschließend die Knopf- und Eingabefeldgenerierung und schließlich der Aktivitätsaufruf selbst.

Ein Beispiel für die Beschreibung eines Aktivitätsaufrufs ist in Listing 6.8 zu sehen. Die Einstellungen in der `activities.xml`-Datei bestehen aus Klassen-Beschreibungen korrespondierend zu ihrer DTD (siehe 6.9). Diese Aktivitätsbeschreibungs-Datei beinhaltet die Beschreibung der Methoden der Klassen aus dem eHome-Modell, die in der GUI des eHomeConfigurators erscheinen sollen. Im Listing 6.8 ist ein Beispiel der XML-Einstellungen für die Methode `Environment newEnvironment(String name, EnvironmentRoot root)` der Klasse `Environment` zu sehen. Diese Methode erstellt ein neues `Environment`-Objekt und verbindet es mit dem `EnvironmentRoot`-Objekt.

Die Einstellungen bedeuten, dass ein Knopf für diese Methode mit der Beschriftung (label) `Button Name` im Editor `ENVIRONMENT_EDITOR` angezeigt wird. Wenn der Benutzer diesen Knopf drückt, erscheint ein Eingabedialog unterhalb der Werkzeugleiste (ähnlich wie in Abbildung 6.14), um die Werte für die Argumente `name` und `root` abzufragen. Mittels `TOOLTIP` kann wie beim Feld `name` eine beschreibende Kurzdokumentation angezeigt werden. Das zweite Argument soll vom Typ `fixed` sein. Das heißt, dass es fixiert, also unveränderlich und nicht auswählbar, ist. Der zuzuordnende Wert wird über den `key` festgelegt. Hier handelt es sich um ein Label, welches vom DataHolder erkannt wird und durch das entsprechende Objekt ersetzt wird.

Das Listing 6.8 und die DTD in Listing 6.9 geben einen Überblick über die Tags, die in dieser XML-Beschreibungsdatei benutzt werden können:

`CLASS` beschreibt die Klasse, die gerade behandelt wird.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT ACTIVITIES (CLASS*)>
<!ELEMENT CLASS (ACTIVITY*)>
<!ATTLIST CLASS name CDATA #REQUIRED>
<!ELEMENT ACTIVITY (TOOLTIP|POSITIVERB|NEGATIVERB|CUSTOMRB|
                    CONTEXTS|PARAM)*>
<!ATTLIST ACTIVITY
          name    CDATA #REQUIRED
          label   CDATA #IMPLIED
          icon    CDATA #IMPLIED >
<!-- Note: Restriction does only apply to string parameter
(both collection and normal ones) -->
<!ELEMENT PARAM (VALUE|TOOLTIP)*>
<!ATTLIST PARAM
          label           CDATA #IMPLIED
          collectiontype  CDATA #IMPLIED
          forceModelElement (true|false) "false"
          restriction       (none|id)   "none" >
<!ELEMENT VALUE (REF|CONST|(CHOICE+))>
<!ATTLIST VALUE type (fixed|preset) "preset" >
<!-- Note: REF values may only be used for nodes -->
<!ELEMENT REF EMPTY>
<!ATTLIST REF key (DEPLOYMENTROOT|DEVICEDEFINITIONROOT
                |ENVIRONMENTROOT|FUNCTIONROOT|SERVICEROOT)
    #REQUIRED>
<!-- Note: CONST values may only be used for non-node types -->
<!ELEMENT CONST (#PCDATA)>
<!-- Note: CHOICE values may only be used for non-fixed
  single-valued non-node types -->
<!ELEMENT CHOICE EMPTY>
<!ATTLIST CHOICE value  CDATA #REQUIRED
          label CDATA #IMPLIED>
<!ELEMENT POSITIVERB EMPTY>
<!ATTLIST POSITIVERB
          type     (value|success|failure|nothing) "nothing">
<!ELEMENT NEGATIVERB EMPTY>
<!ATTLIST NEGATIVERB
          type     (value|success|failure|nothing) "failure">
<!ELEMENT CUSTOMRB (#PCDATA)>
<!ATTLIST CUSTOMRB  type CDATA              "default"
                    style (info|warn|error) "info">
<!ELEMENT TOOLTIP (#PCDATA)>
<!ELEMENT CONTEXTS ((GENERAL|DEPLOYMENT|DEVICEDEFINITION|
        ENVIRONMENT|FUNCTION|SERVICE)*)>
<!ELEMENT GENERAL EMPTY>
<!ELEMENT DEPLOYMENT EMPTY>
<!ELEMENT DEVICEDEFINITION EMPTY>
<!ELEMENT ENVIRONMENT EMPTY>
<!ELEMENT FUNCTION EMPTY>
<!ELEMENT SERVICE EMPTY>
```

Listing 6.9: Die DTD für activities.xml

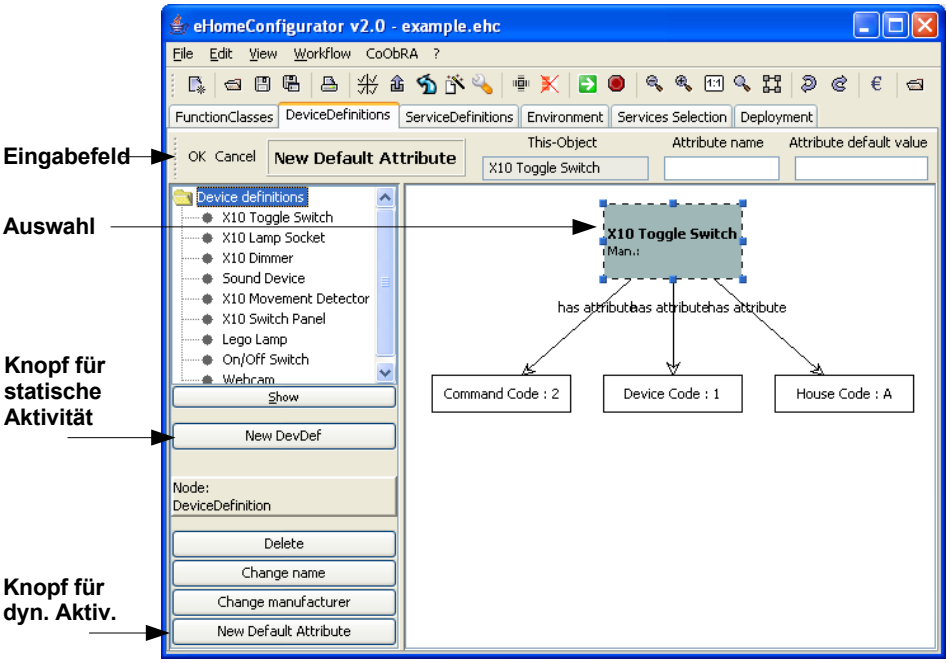

Abbildung 6.14: GUI-Elemente der `createAttribute`-Methode in der Klasse `DeviceDefinition`

`ACTIVITY` beschreibt die auszuführende Methode der betrachteten Klasse.

`CONTEXTS` beinhaltet eine vordefinierte Liste von Editor-Kontexten, in welchen die Methode als Aktivität zur Verfügung steht.

`PARAM` beschreibt ein Argument der Methode. Die Reihenfolge der Argumente wird durch die Methodensignatur festgelegt.

`TOOLTIP` gibt die Möglichkeit eine Kurzdokumentation in Form eines Tool-Tips zu einem Argument anzeigen zu lassen.

Somit stellt diese XML-Beschreibungsdatei hauptsächlich Beziehungen zwischen den Bezeichnungen sowie Kurzdokumentationen und den Methoden her. Weiterhin wird in ihr beschrieben, wie die Eingabefelder für die Aktivität auszusehen haben. Die Typinformation der Argumente wird in der XML-Datei nicht behandelt. Die Verifikation der Eingabefelder und die Typüberprüfung wird mittels der Java Reflection API zur Laufzeit der entsprechenden Aktivität im GenAIM durchgeführt.

Trotzdem ist es auch möglich mittels des `VALUE`-Tag Objekte zu fixieren, so dass sie nicht editiert werden können und mit einem festen Objekt aus der Modellinstanz belegt werden. Das heißt, dass das korrespondierende Eingabefeld grau dargestellt wird wie das "This-object"-Feld in Abbildung 6.14 , nur dass sich sein Wert nicht ändern lässt.

Weiterhin ist es möglich, eine Auswahlliste vordefinierter Werte in den VALUE-Tags anzugeben. Das Eingabefeld ist dann entweder ein Drop-Down-Menü oder ein editierbares Drop-Down-Menu, welches neben der Auswahl aus einer Liste auch die Eingabe neuer Werte erlaubt.

Die Rückgabewerte der Aktivitäten werden nicht ausgewertet und somit ignoriert. Bisher war in diesem Kontext des eHomeConfigurators die Rückgabe eines Objektes an den Benutzer nicht sinnvoll. Die meisten Aktivitäten haben wegen ihres interaktiven Charakters direkte Auswirkung auf die Struktur der eHome-Modell-Instanz, so dass ihr Ergebnis direkt im Graphen im entsprechenden Editor sichtbar ist. Der Fehlerfall wird über Ausnahmebehandlungen abgefangen (Exceptions) und diese und ihre Informationen werden in einem Dialogfenster dem Benutzer präsentiert.

Beim Start des eHomeConfigurators lädt das Spezifizierungswerkzeug die XML-Einstellungsdatei `activities.xml` und erstellt die Knöpfe für die statischen Methoden auf dem linken Seitenpanel des entsprechenden Editors. In Abbildung 6.14 ist dies die statische Methode `DeviceDefinition createDeviceDefinition(String name, String manufacturer, DeviceDefinitionRoot root)` der Klasse `DeviceDefinitionRoot`, welche ein neues Objekt der Klasse `DeviceDefinition` anlegt. Sie wird durch den Knopf mit der Aufschrift `New DevDef` repräsentiert. Die nicht-statischen Aktivitäten werden auch durch Knöpfe repräsentiert, aber getrennt von den Knöpfen der statischen Aktivitäten. Die Knöpfe verändern sich je nach selektiertem Objekt im Graph-Panel. Wenn ein anderes Objekt markiert wird, werden die Knöpfe für dynamische Methoden entsprechend dieser Selektion neu generiert. In Abbildung 6.14 ist ein `DeviceDefinition`-Objekt im Graph-Panel selektiert. Wenn ein Selection-Event auftritt, erscheinen die Knöpfe auf dem linken Seiten-Panel entsprechend der Beschreibung in der XML-Datei. Diese Knöpfe repräsentieren dann die von dem im Editor selektierten Objekt zur Verfügung gestellten Aktivitäten. In Abbildung 6.14 werden vier Knöpfe für das selektierte `DeviceDefinition`-Objekt angezeigt, obwohl die Klasse `DeviceDefinition` ungefähr dreißig öffentliche Methoden implementiert (siehe Listing 6.10), die von Fujaba größtenteils automatisch erzeugt wurden. Sie alle könnten über die XML-Datei an dieser Stelle im Editor als Aktivitäten zur Verfügung gestellt werden.

Weiterhin zeigt Abbildung 6.14 den `New Default Attribute`-Knopf. Dieser Knopf korrespondiert zu der Methode `Attribute createAttribute (String name, String byDefault)` der Klasse `DeviceDefinition`. Bei Druck auf diesen Knopf erscheint der Eingabedialog wie in Abbildung 6.14 dargestellt. Der Dialog beinhaltet die Eingabefelder für Name und Standardwert, das selektierte Objekt (durch `This-Object` bezeichnet) und Knöpfe für die Bestätigung und den Abbruch der Aktivität. Der Eingabedialog verhält sich für statische und dynamische Aktivitäten gleich. Die Knopf-Beschriftungen, die Anzahl der Eingabefelder und die Tool-Tips werden analog wie für die Methode `newEnvironment` Listing 6.8 in der Datei `activities.xml` spezifiziert.

Nachdem der Eingabedialog mit Werten gefüllt wurde und der `OK`-Knopf zur Bestätigung ausgelöst wurde, wird die spezifizierte Methode der Aktivität des im Graph-Panel selektierten Objektes aufgerufen. Die Anzahl und Typen der Eingabewerte werden vom *Activity-Controller* mittels des Java Reflection API überprüft. Der Activity-Controller überprüft auch, ob die spezifizierte Methode im selektierten Objekt existiert und die At-

```
1  class DeviceDefinition
2     String getManufacturer()
3     void setManufacturer(String value)
4     String getName()
5     void setName(String value)
6     int getNumberOfVirtualDevices()
7     void setNumberOfVirtualDevices(int value)
8     boolean addToAttributes(Attribute value)
9     Iterator entriesOfAttributes()
10    Attribute getFromAttributes(String key)
11    String getKeyForAttributes(Attribute value)
12    boolean hasInAttributes(Attribute value)
13    boolean hasKeyInAttributes(String key)
14    Iterator iteratorOfAttributes()
15    void keyChangedInAttributes(String oldKey,
16                                Attribute value)
17    Iterator keysOfAttributes()
18    void removeAllFromAttributes()
19    boolean removeFromAttributes(Attribute value)
20    boolean removeKeyFromAttributes(String key)
21    int sizeOfAttributes()
22    boolean addToControllingServices(Service value)
23    boolean hasInControllingServices(Service value)
24    void removeAllFromControllingServices()
25    boolean removeFromControllingServices(Service value)
26    int sizeOfControllingServices()
27    DeviceDefinitionRoot getDeviceDefinitionRoot()
28    boolean setDeviceDefinitionRoot
29                    (DeviceDefinitionRoot value)
30    boolean addToDevices(Device value)
31    boolean hasInDevices(Device value)
32    Iterator iteratorOfDevices()
33    void removeAllFromDevices()
34    boolean removeFromDevices(Device value)
35    int sizeOfDevices()
36    Attribute createAttribute
37                    (String name, String byDefault)
38    void removeYou()
```

Listing 6.10: Alle öffentlichen Methoden (einschließlich der automatisch von Fujaba erzeugten) der Klasse `DeviceDefinition`

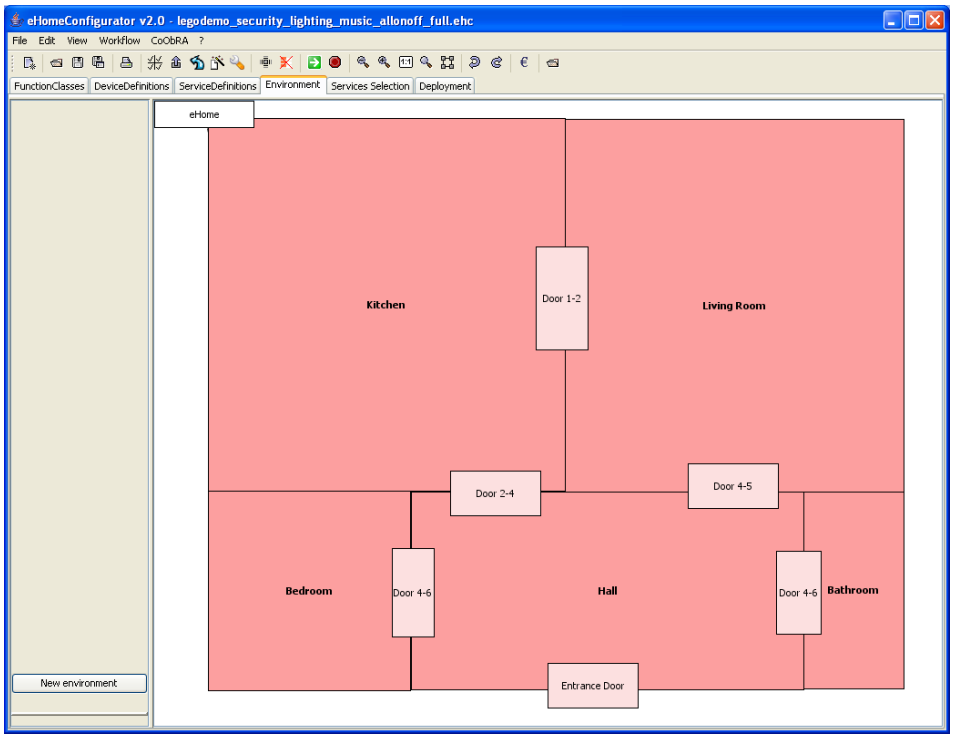

Abbildung 6.15: Grundriss-Ansicht im Environment-Editor

tributwerte zur Signatur der Methode passen. Wenn diese Validierung positiv ist, wird die Methode ausgeführt. Ansonsten wird eine Fehlermeldung angezeigt. Kehrt die Methode ohne das Auftreten einer Ausnahme (exception) zurück, wird der Eingabedialog geschlossen, aus dem Editor-Panel entfernt, und das Graph-Panel wird neu gezeichnet, um die durchgeführte Änderung sichtbar zu machen. Tritt eine Ausnahme auf, wird die Exception-Information in einem Fehlerdialog angezeigt.

6.5.3 Umgebungsdarstellung

Eines der eingangs formulierten Ziele ist das Senken der Kosten für die Einrichtung bzw. den Bau eines automatisierten Heims. Zu diesem Zweck ist es effektiv, bestimmte Aufgabenbereiche, die vorher Experteneinsatz benötigten, auch Laien zugänglich zu machen. Einer dieser Aufgabenbereiche ist das Modellieren der physikalischen Umgebung im Environment-Editor, also aller vorhandenen Räume, der in ihnen befindlichen Geräte sowie die zwischen den Räumen bestehenden Verbindungen wie Türen und Fenster.

Eine Darstellung in Form eines Grundrisses ist auch für unerfahrene Benutzer intuitiv verständlich. Aus diesem Grunde wurde die Umgebungsdarstellung so realisiert, dass sie einen schematischen zweidimensionalen Grundriss repräsentieren kann. Vergleiche dafür auch Abbildung 6.15.

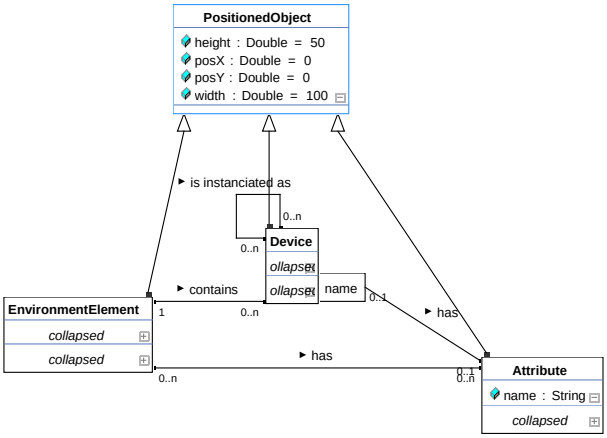

Abbildung 6.16: Die Klasse `PositionedObject`

Zur Realisierung einer solchen Darstellung wurden alle in der Umgebungsdarstellung vorkommenden Klassen des Modells um Klassen-Attribute zur Speicherung von Position und Größe erweitert. Dies wird durch eine Oberklasse namens `PositionedObject` erreicht, von der die entsprechenden Klassen, wie in Abbildung 6.16 dargestellt, erben.

Während der Arbeit mit dem Werkzeug werden bei jeder Betätigung einer Maustaste das darunterliegende Objekt des Datenmodells ermittelt und Positions- sowie Größendaten aktualisiert. Beim Wiederherstellen einer Modellierung werden diese Daten geladen, die korrekten Dimensionen wiederhergestellt und die Objekte dann an die entsprechenden Positionen verschoben. Hierbei werden solche, die Räume repräsentieren, in der Darstellung nach hinten sortiert, gefolgt von Elementen wie Türen und Fenstern. Über diesen befinden sich die modellierten Geräte, und auf oberster Ebene werden schließlich die Attribute dargestellt.

Durch einfaches Mitzählen aller Objekte der `Device`-Klasse in einer Lokation ist während der Konfigurierungsphase eine automatisch erfolgende Anordnung der erzeugten Geräte möglich. Diese werden relativ zur Position des Raumes, in dem sie sich befinden, um einige Pixel nach rechts und nach unten verschoben und dort untereinander dargestellt. Analog erfolgt auch die Anordnung der Attribute.

Auf diese Weise wird eine übersichtliche und intuitive Benutzeroberfläche gewährleistet, die auch für unerfahrene Anwender ohne Informatikkenntnisse verständlich ist. Eine Modellierung verschiedener Etagen ist ebenfalls möglich, auch wenn eine dreidimensionale Darstellung bisher nicht unterstützt wird.

6.6 Konfigurierungs-Werkzeug (Configurator)

Das Konfigurierungs-Werkzeug hat die Aufgabe, den Dienst-Objekt-Kontext entsprechend den in der Spezifizierungsphase ausgewählten eHome-Diensten zu erstellen. Das Werkzeug vervollständigt die Konfiguration so, dass sie im eHome deployed werden

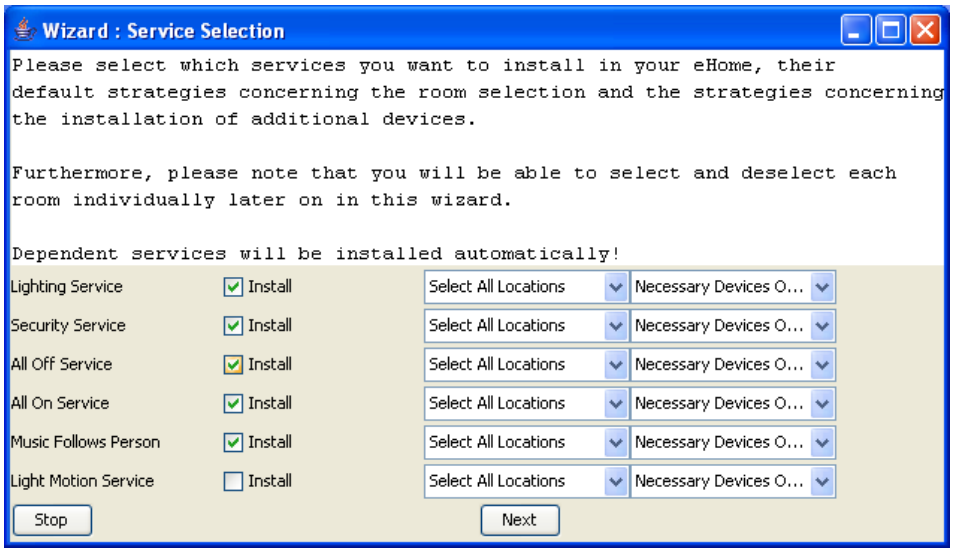

Abbildung 6.17: Der Dialog des Konfigurierungsassistenten zur Auswahl der Dienste und der entsprechenden Raum- und Geräteauswahlstrategie

kann. Es sind verschiedene Assistenten (eng. Wizards) vorhanden, die den Benutzer durch diese Konfigurierungsphase leiten. Die Summe der Assistenten wird unter dem Stichwort Konfigurierungsassistent zusammengefasst. Die entsprechenden Startknöpfe befinden sich in der Werkzeugleiste des Spezifikations-Werkzeuges. Dieses Werkzeug beinhaltet alle notwendigen Aktionen der Konfigurierungsphase. Es automatisiert nahezu vollständig den gesamten Konfigurierungsprozess, durch spezielle Graphsuche- und Graphtraversierungsmechanismen. Die verbleibenden manuellen Operationen sind Auswahl von Realisierungsalternativen und die Eingabe der zur Parametrisierung erforderlichen Attribute.

Das Ergebnis der Konfigurierungsphase ist der instanzierte Dienst-Objekt-Kontext innerhalb der eHome-Modell-Instanz. Die generierten Strukturen repräsentieren die Grundlage für die Laufzeit-Konfiguration der eHome-Dienste. Diese unterscheidet sich nur noch von dieser Deploymentkonfiguration durch die State-Objekte. Diese Ergebnisstruktur ist die Eingabe für das Deployment-Werkzeug.

Dieses Kapitel beschreibt die Schritte, die in der Konfigurierungsphase mit dem Konfigurierungs-Werkzeug durchgeführt werden und die entsprechende Realisierung und Implementierung. Ein Teil der Vorarbeiten zu diesem Kapitel wurde in [Mal05, Sch05b] durchgeführt.

6.6.1 Dienstauswahl, Raum- und Geräteauswahlstrategie

Beim Start des Konfigurierungsassistenten, wird dem Benutzer als erstes der Dialog aus Abbildung 6.17 präsentiert. Dieser ermöglicht die Auswahl aus den verfügbaren

(also in der Dienstspezifikation angelegten) top-level Diensten und der zugehörigen Raumauswahl- und Geräteauswahlstrategie.

Bei der Raumauswahlstrategie kann zwischen "Select All Locations" und "Select Possible Locations" und bei der Geräteauswahlstrategie zwischen "Necessary Devices Only" oder "All Devices" gewählt werden. Die Raumauswahlstrategien werden im nächsten Abschnitt erklärt. Die Geräteauswahlstrategie spielt bei der automatischen Dienstkonfigurierung im übernächsten Abschnitt eine Rolle. So werden bei der Auswahl des Punktes "Necessary Devices Only" nur die benötigten Dienste bei der Unterdienstsuche einbezogen und deren benutzte Geräte in die Konfiguration als "zu installieren" eingebaut. Insgesamt werden also nur die für die Minimalanforderung eines Dienstes wirklich benötigten Geräte gesucht. Bei der Auswahl von "All Devices" werden bei der Unterdienstsuche auch die optionalen Dienste mit einbezogen und somit alle für diese Dienstrealisierung möglichen Geräte angefordert.

Raumauswahlstrategie

Die Wahl der Raumauswahlstrategie trifft eine Vorentscheidung darüber, in welchen Räumen ein Dienst installiert werden soll. Die mit dieser Strategie bestimmte Selektion ist jedoch noch nicht final und kann im weiteren Verlauf des Assistenten noch an die individuellen Wünsche des Benutzers angepasst werden. Eine sinnvolle Auswahl der hier anzuwendenden Strategie erleichtert dem Anwender jedoch diese Aufgabe.

Zur Verfügung stehen zwei verschiedene Möglichkeiten. Zum einen können standardmäßig alle vorhandenen Räume für die Installation selektiert werden ("Select All Locations"), was im *Bauszenario* oder *Ausrüstszenario*, wie in Kapitel 2.3 beschrieben, sinnvoll ist. Zum anderen kann die automatische Raumauswahl auf diejenigen Räume beschränkt werden, in denen alle notwendigen Geräte bereits vorhanden sind ("Select Possible Locations"). Ein solches Verfahren ist im Rahmen des *Steuerungsszenarios* zu präferieren, da der Besitzer mit großer Wahrscheinlichkeit nicht am Erwerb weiterer Technik interessiert ist. Für jede dieser beiden Strategien existiert in der `AutoConfigurator`-Klasse eine separate Methode, nach deren Terminierung die entsprechenden Räume für einen Dienst ausgewählt sind:

Wahl aller Räume: Wie in Abbildung 6.18 dargestellt, erfolgt der Aufruf der `selectAllLocations`-Methode mit zwei Parametern. Bei diesen handelt es sich um den auszuwählenden Dienst, `svc` vom Typ `Service` sowie um die Umgebung, `env` vom Typ `Environment`, in der die Auswahl aller Räume stattfinden soll. Die Auswahl aller Räume erfolgt in zwei Schritten, welche jeweils durch ein eigenes Story-Pattern modelliert werden. Im ersten Schritt wird die bestehende Raumauswahl gelöscht.

In diesem Teildiagramm wird nach einer Objektinstanz vom Typ `Location` gesucht, welche bestimmte Eigenschaften erfüllt. Zum einen muss dieser Raum in der beim Aufruf übergebenen Umgebung liegen, was durch die Relationskante `contains` zwischen `loc` und `env` dargestellt ist. Zum anderen muss der vorgegebene Dienst dort zur Installation ausgewählt sein. Diese Beziehung wird durch die `offers`-Kante zum Ausdruck gebracht. Falls es im betrachteten Datensatz eine solche Konstellation gibt, dann liefert dieses Story-Pattern die entsprechende

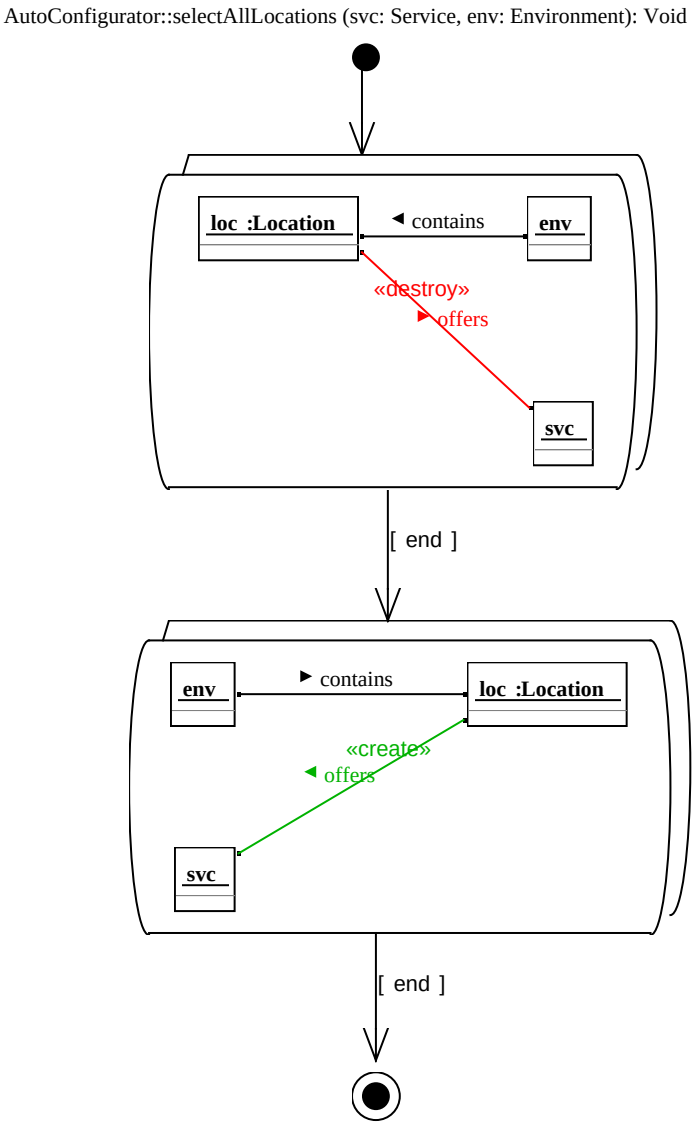

Abbildung 6.18: Story-Diagramm: Auswahl aller Räume

`Location`-Instanz zurück. Sinn des Teilschrittes ist es, die Auswahl des Dienstes zu löschen, daher ist die Relation mit dem Schlüsselwort `destroy` versehen. Wenn ein solcher Raum existiert, dann wird die Kante, die die Dienstauswahl symbolisiert, folglich gelöscht.

Die Dienstauswahlkante soll jedoch nicht nur in einem, sondern in allen Räumen

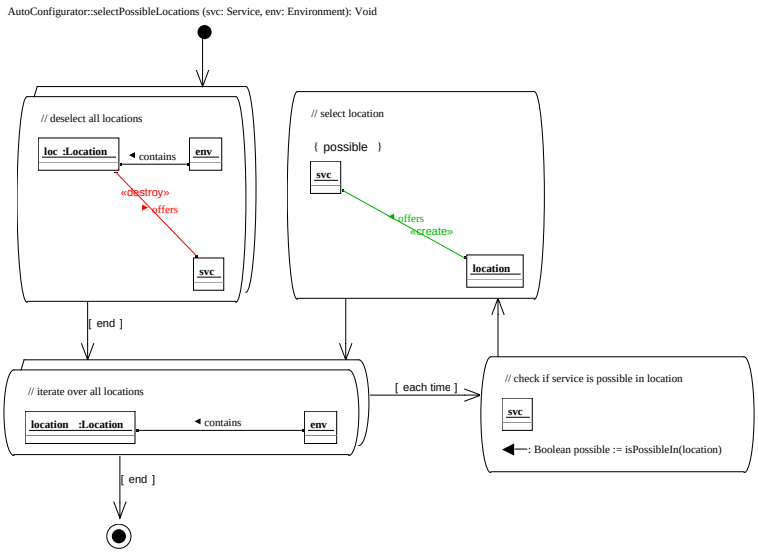

Abbildung 6.19: Activity-Diagramm: Auswahl eines Teils der Räume

gelöscht werden. Aus diesem Grund ist das Story-Pattern mit einem Doppelrahmen dargestellt. Hierdurch wird eine Schleife modelliert, die nicht nur eine einzelne, die abgebildeten Eigenschaften erfüllende `Location`-Instanz betrachtet, sondern über alle solche Konstellationen iteriert. Folglich werden alle Räume, die für die Installation des betreffenden Dienstes ausgewählt sind, gefunden und ihre `offers`-Kante gelöscht. Nach Beendigung der Schleife erfolgt eine Transition zu einem weiteren Teildiagramm. Diese Tatsache wird durch die Beschriftung der Transitionskante mit dem Schlüsselwort `end` zum Ausdruck gebracht. Wäre eine Ausführung des zweiten Story-Patterns bei jeder Schleifeniteration erwünscht, so wäre dieser Übergang mit `each time` beschriftet und es gäbe eine unbeschriftete Transition zurück zum Schleifendiagramm.

Im zweiten Schritt erfolgt ein weiterer Schleifendurchlauf, welcher über alle in der Umgebung vorhandenen Räume iteriert und in jedem Schritt eine Auswahlkante zum Dienstobjekt erzeugt, was durch das Schlüsselwort `create` ausgedrückt wird. Hierbei ist zu beachten, dass die gesuchte Instanzkonstellation hier im Gegensatz zur oben erläuterten Situation lediglich aus dem Tupel `loc` und `env` besteht. Nach Beendigung dieses Vorgangs, wiederum angezeigt durch das Schlüsselwort `end`, wird in den Endzustand der Methode übergegangen.

Wahl anhand der vorhandenen Infrastruktur: Die Auswahl der Räume anhand der vorhandenen Geräte gestaltet sich schwieriger als die oben beschriebene Methode, wie Abbildung 6.19 zu entnehmen ist. Der erste Schritt ist gleich. Zunächst wird die vorher getroffene Auswahl aufgehoben und dann über alle `Location`-Objekte der `Environment`-Instanz iteriert. Hier enden dann die Parallelen. Mit

einer `for each`-Transition sind zwei Story-Patterns an die Schleife gebunden, welche für jeden gefundenen Raum ausgeführt werden. Im ersten Teildiagramm erfolgt der Aufruf einer Methode der `Service`-Instanz, welche den Namen `isPossibleIn` trägt. Diese Methode liefert einen booleschen Wert zurück, welcher wahr ist, falls alle für den Dienst notwendigen Geräte in dem als Parameter übergebenen Raum vorhanden sind und diese noch nicht von einem anderen Service beansprucht werden. Gespeichert wird dieser Rückgabewert in einer Variable namens `possible`.

Im darauf folgenden Diagramm wird dann die `offers`-Kante zwischen `Service` und `Location` erstellt. Die Ausführung dieses Teildiagramms erfolgt jedoch *ausschließlich* dann, wenn der in geschweiften Klammern angegebene boolesche Ausdruck wahr ist, in unserem Fall also `possible`. Durch den unbeschrifteten Transitionspfeil zurück zum Schleifendiagramm wird modelliert, dass im Anschluss zur nächsten Schleifeniteration übergegangen werden kann.

Suche nach einem geeigneten Dienst

Nach der Raumauswahl müssen die geeigneten Dienste und Unterdienste gesucht werden, die eine bestimmte Funktionalität in der richtigen Kardinalität anbieten.

Mit der `findBiggestProvider`-Methode kann nach dem Dienst gesucht werden, der die größte Kardinalität zu einer gegebenen Funktionalität anbietet. Die entsprechende Instanz der `Service`-Klasse wird nach erfolgreicher Beendigung der Methode zurückgegeben. Sollte die gesuchte Funktionalität von keinem der spezifizierten Dienste zur Verfügung gestellt werden, so wird ein Nullzeiger zurückgegeben. Im Idealfall bietet der gefundene Dienst unendlich viele Funktionalitätssinstanzen an.

Wie in Abbildung 6.20 dargestellt, besteht das Aktivitätsdiagramm dieser Methode aus drei Story-Patterns. Im ersten erfolgt lediglich die Deklarierung zweier Variablen. Hier wird eine ganzzahlige Variable namens `bestProvision` angelegt, in welcher, wie der Name andeutet, die momentan beste gefundene Kardinalität zwischengespeichert wird. Diese Zahl dient jedoch nur dem Vergleich, denn von Interesse ist die dazugehörige Instanz der `Service`-Klasse, welche diese Anzahl an Funktionalität zur Verfügung stellt. Diese wird in der zweiten Variable, `bestService`, gespeichert. Bei der Erzeugung des Quellcodes sorgt Fujaba dafür, dass diese zweite Variable mit `NULL` initialisiert wird.

Im zweiten Schritt wird in einer Schleife über alle Dienste iteriert, welche die Funktionalität zur Verfügung stellen, deren `findBiggestProvider`-Methode aufgerufen wurde. Das zwischen `Function`- und `Service`-Klasse liegende `ServiceFunctionCardinality`-Objekt dient lediglich der Speicherung der Kardinalität. Dieser Ansatz ist notwendig, weil Fujaba eine Beschriftung der Relationskanten nicht unterstützt. Nicht jeder der auf diese Weise gefundenen Dienste ist von Interesse, sondern lediglich solche, deren angebotene Kardinalität höher ist als die des bisher besten Kandidaten. Aus diesem Grund wird in der Schleife eine weitere Einschränkung definiert. Diese befindet sich im aktuell betrachteten Story-Pattern in geschweiften Klammern und beinhaltet zwei durch ein logisches *oder* verknüpfte Ausdrücke. Der erste ist dann wahr, wenn die Kardinalität des Dienstes -1 beträgt. Dies symbolisiert, dass die gesuchte Funktionalität unendlich oft zur Verfügung gestellt wird. Der zweite stellt sicher, dass der Dienst eine größere Anzahl der Funktionalität zur Verfügung stellt, als der mo-

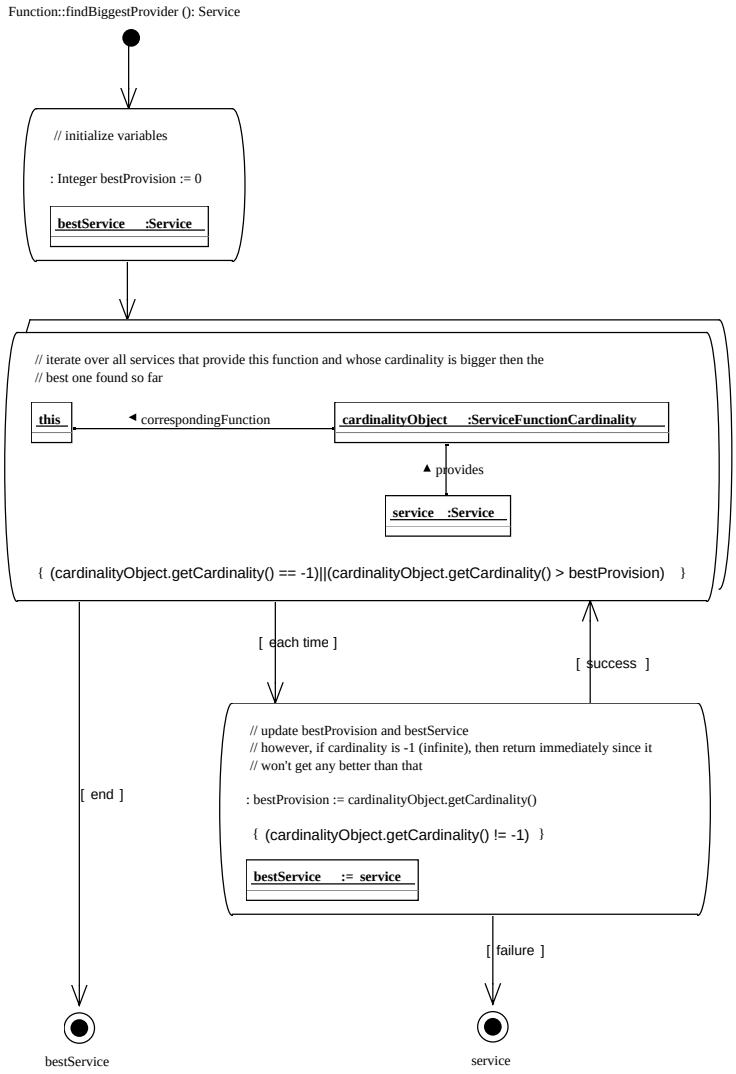

Abbildung 6.20: Activity-Diagramm: Suche nach einem geeigneten Dienst

mentan beste. Sobald eine dieser beiden Bedingungen eintritt, erfolgt die Abarbeitung des dritten Story-Patterns.

In diesem erfolgt die Aktualisierung der beiden Variablen `bestService` und `bestProvision`. Zudem erfolgt auch hier die Überprüfung einer Bedingung. Falls der gefundene Dienst eine unendliche Menge an Funktionalität zur Verfügung stellt, so wird dieser direkt zurückgegeben. Die Methode `findBiggestProvider` sucht nach einem der größten Anbieter, und da die Suche nach einem Dienst, der eine Funktionalität mehr als unendlich oft zur Verfügung stellt, aussichtslos ist, wird die Suche an dieser Stelle abgebrochen.

Für den Fall, dass mehrere Dienste die gleiche, größte Anzahl an Funktionalität anbieten, liefert diese Methode folglich immer den zuerst gefundenen Dienst zurück. An dieser Stelle sind sicherlich noch andere Algorithmen denkbar. Durch ein einfaches Ersetzen der Bedingung `(cardinalityObject.getCardinality() > bestProvision)` durch `(cardinalityObject.getCardinality() >= bestProvision)` ändert sich das Verhalten insofern, als dass nun immer der letzte gefundene Dienst zurückgeliefert wird. Es ist jedoch fraglich, ob sich die Qualität des Ergebnisses hierdurch verbessert.

Während der Installation eines Dienstes in einem Raum werden zu jeder benötigten Funktionalität passende Unterdienste gesucht, welche diese anbieten. Dies geschieht mit Hilfe der gerade beschriebenen `findBiggestProvider`-Methode, welche diejenige Dienstbeschreibung zurück liefert, deren Kardinalität für diese betrachtete Funktionalität am größten ist. Auf diese Weise wird ein Abhängigkeitsbaum erzeugt, welcher voneinander abhängige Dienstinstanzen beinhaltet.

Eine gesonderte Betrachtung der Erstellung dieses Baumes ist sinnvoll, da durch ihn auch die vorgeschlagenen Geräte bestimmt werden, welche im Rahmen eines Bau- oder Ausrüstszenarios Kosten verursachen. Hier sind weitere Strategien denkbar:

- Minimiere die Anschaffungskosten der Geräte!
- Nutze möglichst Geräte mit der gleichen Schnittstelle!
- Nutze möglichst Geräte desselben Herstellers!
- Nutze möglichst Geräte eines speziellen Herstellers!
- ...

Diese können durch separate Funktionen realisiert und anhand von zuvor festgelegten Präferenzen ausgewählt werden. Hierdurch ist eine noch stärkere Anpassung an die individuellen Wünsche des eHome-Besitzers möglich.

Suche nach vorhandener Dienstinstanz

Die Klasse `EnvironmentElement`, welche als gemeinsame Oberklasse für alle Arten von Räumen und sonstigen Lokalitäten in einer eHome-Umgebung dient, stellt unter anderem auch eine Methode namens `containsSuitableServiceObject` zur Verfügung, mit deren Hilfe festgestellt werden kann, ob in der betreffenden Lokalität ein Dienst installiert ist, der eine bestimmte Funktionalität in ausreichendem Maße anbietet. Für den Fall, dass mehrere solcher Dienstinstanzen existieren, verfolgt

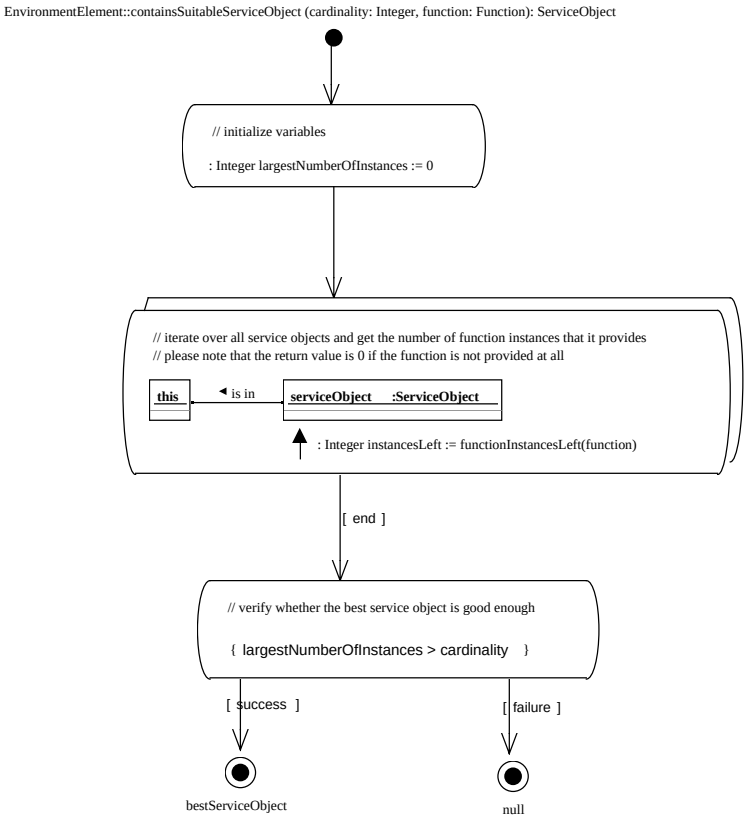

Abbildung 6.21: Activity-Diagramm: Suche nach einem geeigneten Dienst-Objekt

`containsSuitableServiceObject` das Prinzip, diejenige Dienstinstanz als Ergebnis zurückzuliefern, welche die größere Kardinalität anbietet. Dieser Ansatz verhält sich analog zum *Worst-Fit*-Ansatz im Bereich der Speicherfragmentierung, bei dem eine Allozierung immer im größten zusammenhängenden freien Fragment des Speichers stattfindet, um den Grad der Fragmentierung zu senken [SGG02]. Es gibt jedoch eine Ausnahme: Sollte ein Dienst gefunden werden, der exakt die benötigte Menge anbietet, dann wird diesem der Vorzug gegeben.

Das Gerüst der Methode ist in Abbildung 6.21 dargestellt. Diese wird mit zwei Parametern aufgerufen, bei denen es sich um die gesuchte Funktionalität und deren benötige Kardinalität handelt. Die zu durchsuchende Lokalität ist durch die Objektinstanz bestimmt, deren Methode aufgerufen wird.

Zunächst erfolgt, ähnlich wie bei der Suche nach einer Dienstbeschreibung, die Deklaration einer Integer-Variable, in der das bisher beste Suchergebnis zwischengespeichert wird. Diese wird mit dem Wert 0 initialisiert. Im Anschluss erfolgt die Iteration über alle in der Lokalität installierten Dienste. Für jeden dieser Dienste erfolgt der Aufruf einer Methode namens `functionInstancesLeft`, welche von der `ServiceObject`-

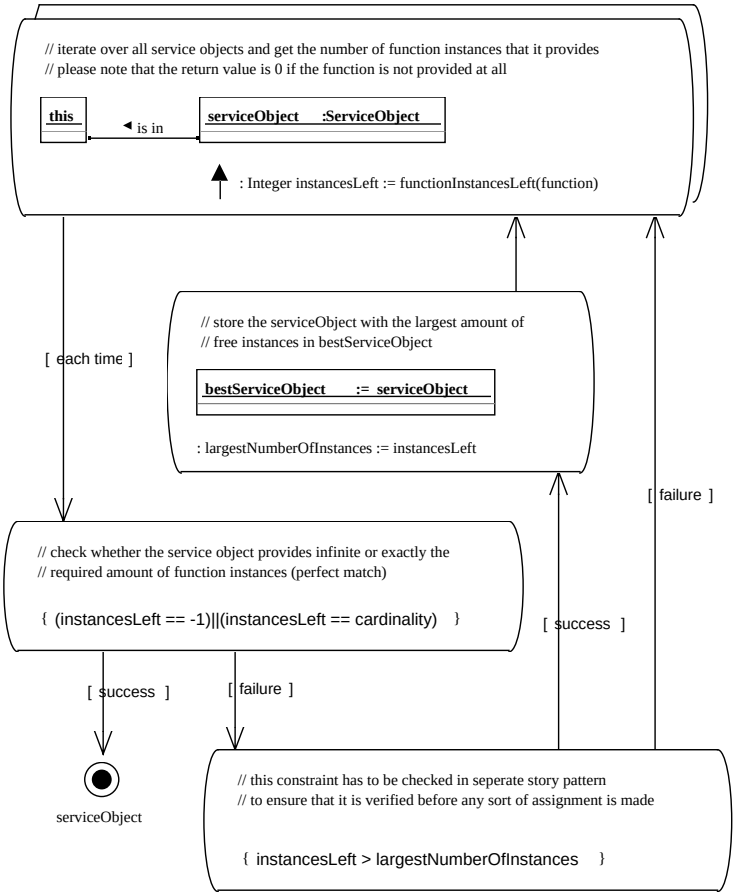

Abbildung 6.22: Activity-Diagramm: Suche nach einem geeigneten Dienst-Objekt (2)

Klasse zur Verfügung gestellt wird. Diese Methode erhält als Parameter eine Funktionalität und liefert als Ergebnis zurück, welche Kardinalität die betreffende Dienstinstanz von dieser Funktionalität noch zur Verfügung stellen kann. Der Rückgabewert hängt hierbei von den übergeordneten Diensten ab, welche dieses `ServiceObject` bereits benutzen und somit eine bestimmte Kardinalität für sich beanspruchen. Falls die übergebene Funktionalität gar nicht angeboten wird, erfolgt die Rückgabe des Wertes Null, so dass in jedem Fall ein sinnvolles Ergebnis geliefert wird.

Dieser besagte Rückgabewert wird in einer Variable namens `instancesLeft` zur weiteren Auswertung zwischengespeichert. Der Graph der nun folgenden Schritte ist in Abbildung 6.22 zu sehen.

Zunächst wird überprüft, ob die momentan betrachtete Dienstinstanz die gesuchte Funktionalität mit exakt der benötigten oder mit unendlicher Kardinalität zur Verfügung stellt. In beiden Fällen erfolgt die sofortige Terminierung der Methode und die Rückga-

be dieses `ServiceObjects`. Da bei der beschriebenen automatischen Dienstauswahl immer solchen mit unendlicher Kardinalität Vorrang gegeben wird, ist davon auszugehen, dass sich diese beiden Möglichkeiten gegenseitig ausschließen, solange der Benutzer nicht nachträglich die Auswahl von Hand geändert hat, um die Konfiguration an seine individuellen Wünsche anzupassen. Dies macht auch Sinn, denn alle Dienste, die das `ServiceObject` mit eingeschränkter Kardinalität benutzen, könnten genauso dasjenige mit unendlicher benutzen. Dadurch wäre die Dienstinstanz mit Restriktionen dann überflüssig.

Für denn Fall, dass weder ein perfektes noch ein unendliches Ergebnis vorliegt, wird im folgenden Story-Pattern überprüft, ob der Wert besser ist, als das bisher optimale Ergebnis. Falls dies der Fall ist, wird `largestNumberOfInstances` auf den neuen Wert gesetzt und das `ServiceObject` in einer Variable namens `bestServiceObject` zwischengespeichert. Dieser Vorgang wird für alle in der Lokalität vorhandenen Dienstinstanzen wiederholt.

Nach erfolgreichem Abarbeiten aller installierten Dienste befindet sich nun dasjenige `ServiceObject` in der Variable `bestServiceObject`, welches die größte Kardinalität der gesuchten Funktionalität in der betrachteten Lokalität zur Verfügung stellt. Diese ist weder unendlich, noch entspricht sie genau der gesuchten Anzahl, da die Methode ansonsten bereits vorzeitig beendet worden wäre.

Im letzten Schritt, welcher in der ersten Teilabbildung 6.21 dargestellt ist, erfolgt noch eine Überprüfung, ob die gefundene Kardinalität größer ist als die gesuchte. Abhängig von diesem Ergebnis erfolgt entweder die Rückgabe der entsprechenden Dienstinstanz oder eines Nullzeigers.

Verbesserungsmöglichkeiten

Im Rahmen der Dienstinstallation erfolgt eine Überprüfung, ob im betrachteten Raum bereits ein Dienst installiert ist, welcher die benötigte Funktionalität in ausreichendem Maße zur Verfügung stellt. Hierbei ergibt sich die Einschränkung, dass Funktionalität in vollem Umfang von einem einzigen Dienst zur Verfügung gestellt werden muss. Werden beispielsweise fünf Instanzen benötigt, und existieren zwei Dienstobjekte im Raum, welche jeweils drei anbieten, dann erfolgt *keine* Verteilung des Bedarfes auf diese beiden Dienste. Dies kann je nach Situation sinnvoll oder unsinnig sein. Handelt es sich um fünf benötigte Leitungen auf einer parallelen Schnittstelle, so ist die Verteilung der Installation eines weiteren Ports vorzuziehen. Handelt es sich jedoch um Schalter auf einer Fernbedienung, so wünscht sich der Benutzer mit Sicherheit keine Steuerung des Dienstes mit Hilfe von zwei verschiedenen Geräten.

6.6.2 Dienstinstallation

Ausgelöst wird der automatische Installationsvorgang durch den Aufruf der Methode `doConfig` der `AutoConfigurator`-Klasse, welche in Abbildung 6.23 dargestellt ist. Die eigentlich interessanten Vorgänge passieren jedoch in den `install`- und `installOptional`-Methoden der `Service`-Klasse, welche hier aufgerufen werden. Die grafische Modellierung von `doConfig` ist folglich kurz und schnell erläutert.

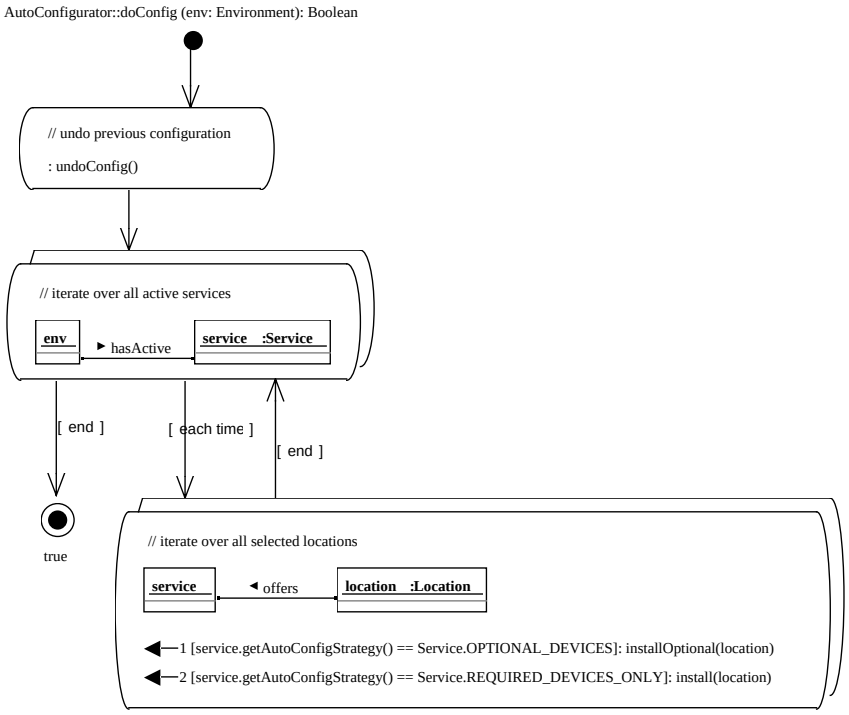

Abbildung 6.23: Activity-Diagramm: Start der automatischen Konfigurierung

Das erste Teildiagramm besteht lediglich aus einem Methodenaufruf. Mit `undoConfig`, ebenfalls eine Methode der `AutoConfigurator`-Klasse, werden alle zuvor im Rahmen einer vorhergegangenen Konfiguration erstellten Objektinstanzen gelöscht.

Bei den beiden folgenden Story-Patterns handelt es sich um zwei ineinander verschachtelte Schleifen. In der ersten wird über alle zur Installation ausgewählten Dienste iteriert. Im darauf folgenden Diagramm werden nun zu jedem dieser Dienste die jeweils ausgewählten Räume gefunden und als Parameter an eine der Installationsmethoden übergeben. Hierzu werden die beiden Kollaborationsausdrücke der Reihe nach abgearbeitet. Es handelt sich um einfache `if-then` Ausdrücke, bei denen das `autoConfigStrategy`-Attribut der Serviceinstanz abgefragt wird, in welchem die anzuwendende Installationsstrategie gespeichert ist.

Installationsmethode ohne optionale Geräte

Der Graph der Methode `install` ist sehr umfangreich und komplex (siehe Abbildung 6.24). Im Folgenden wird er daher in kleineren Ausschnitten besprochen. Die `install`-Methode der `Service`-Klasse installiert den Dienst in exakt einem Raum, welcher als Parameter in Form einer `Location`-Instanz übergeben wird. Zurückgegeben wird eine `ServiceObject`-Instanz, welche den Dienst in einem Teil der Umgebung

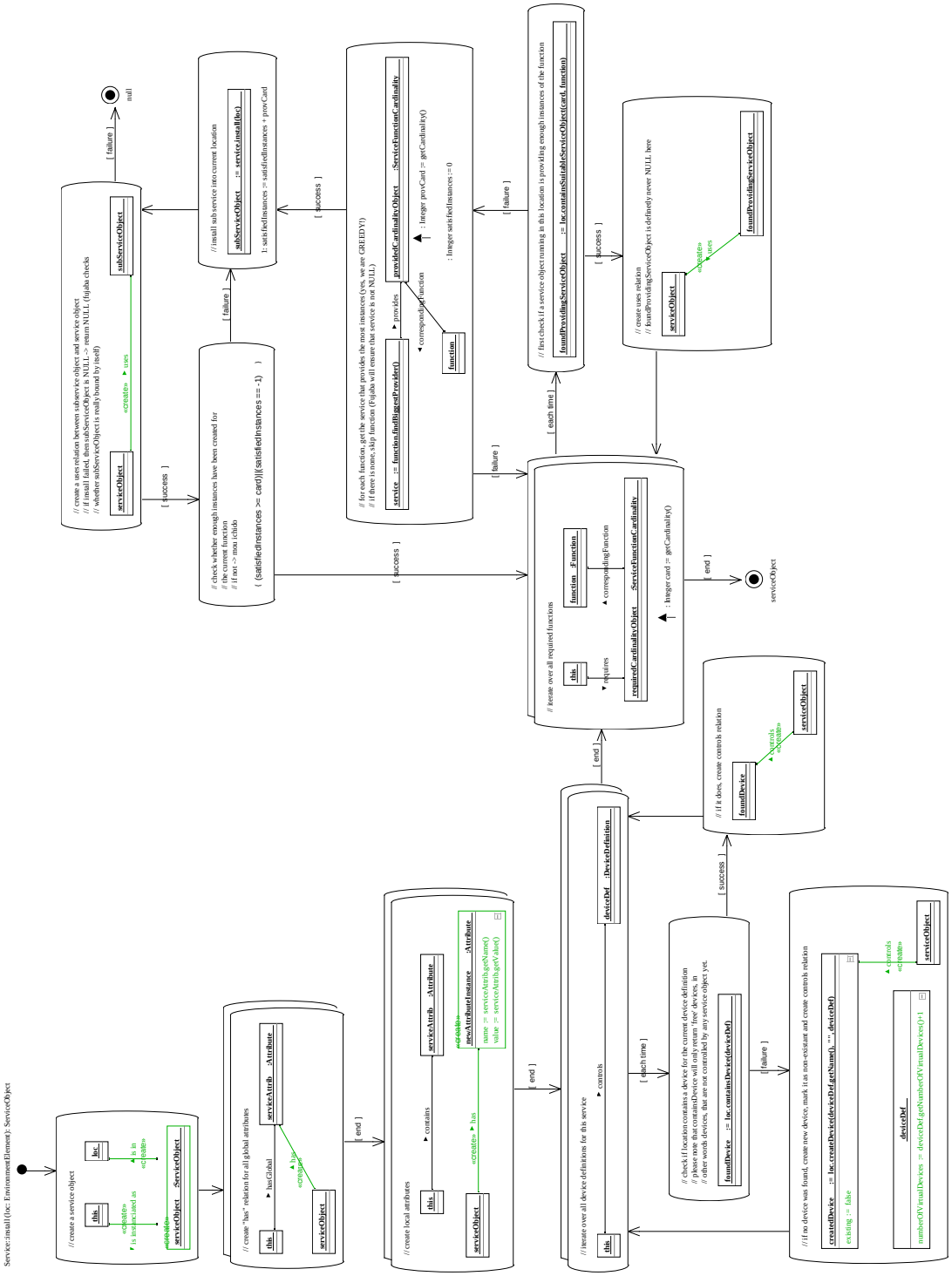

Abbildung 6.24: Activity-Diagramm: `install`-Methode

zur Laufzeit repräsentiert. Zur Erinnerung sei hier noch einmal erwähnt, dass das Verhältnis von `Service` und `ServiceObject` demjenigen von `DeviceDefinition` und `Device` entspricht.

Generierung einer Dienstinstanz Die ersten Schritte sind in Abbildung 6.25 dargestellt.

Im ersten Schritt wird eine neue Instanz der `ServiceObject`-Klasse, eine `is in`-Relation zum übergebenen Raum und eine `is instantiated as`-Relation zu dem `Service` erzeugt, dessen `install`-Methode aufgerufen wurde.

Im Anschluss wird eine Schleife durchlaufen, welche über alle globalen Attribute des Dienstes iteriert und eine Verknüpfung zum neu erstellten `ServiceObject` erstellt. Folglich betrifft die Änderung eines solchen zur Dienstbeschreibung gehörigen Attributes automatisch auch alle Dienstinstanzen, was aber, wie durch das Schlüsselwort *global* zum Ausdruck gebracht wurde, ein erwünschter Effekt ist.

Nach Beendigung dieser Schleife erfolgt die Behandlung aller lokalen Attribute, welche mit der Dienstbeschreibung durch eine `contains`-Kante verbunden sind. Von diesen erhält jede Dienstinstanz eine eigene Kopie, was durch das Schlüsselwort `create` und die grünfarbige Umrandung des Objekts namens `newAttributeInstance` zum Ausdruck gebracht wird, was in der gedruckten Fassung dieser Ausarbeitung durch gepunktete Linien dargestellt ist. Ebenso wird eine entsprechende `has`-Relationskante angelegt, welche die neue Attributinstanz der Dienstinstanz zuordnet. Die folgenden Schritte sind in Abbildung 6.26 zu sehen.

Anschließend wird über alle Geräte iteriert, welche der Dienst direkt kontrollieren soll. Da es sich, wie bereits mehrfach erwähnt, bei einem `Service` um eine abstrakte Beschreibung handelt, besteht eine Verknüpfung zu `DeviceDefinition` und nicht zu `Device`.

Für jede gefundene Gerätebeschreibung wird zunächst überprüft, ob der betrachtete Raum ein entsprechendes Gerät enthält. Da sich ein Raum in beliebig viele sogenannte `Sublocations` unterteilen kann, ist diese Überprüfung nicht durch triviale Suche nach einer Kante zwischen einem Gerät dieser Gerätebeschreibung und dem aktuellen Raum möglich, denn dieses könnte sich beispielsweise auch an einer Tür befinden, die zum Raum gehört. Vielmehr existiert zu diesem Zweck in der `Location`-Klasse eine eigene Methode namens `containsDevice`, welche feststellt, ob in dem Raum, den sie symbolisiert, ein zu einer als Parameter übergegebenen Gerätedefinition passendes Gerät existiert, welches von keinem anderen Dienst benutzt wird. Ist dies der Fall, dann wird die gefundene `Device`-Instanz zurückgegeben. Im Falle eines Fehlschlages erfolgt die Rückgabe eines `NULL`-Zeigers.

Nach Ausführung dieser Methode wird dem Ergebnis entsprechend verzweigt. Im Erfolgsfall erfolgt die Erzeugung einer `controls`-Relationskante, welche die Kontrollbeziehung ausdrückt. Diese Kante hat zur Folge, dass ein weiterer Aufruf der `containsDevice`-Methode nun nicht mehr das Gerät als Ergebnis zurückliefert.

Falls ein solches Gerät jedoch nicht existiert, wird es im Raum neu angelegt und mit Hilfe einer booleschen Variable namens `existing` als *virtuell*, also nicht existent markiert. Im Gegensatz zu real existierenden Geräten, welche grau dargestellt sind, tauchen solche Geräte in der Umgebungsdarstellung des Werkzeugs als orangefarbene Rechtecke auf, um auszudrücken, dass diese Geräte noch zu kaufen und zu installieren sind. Bei dem

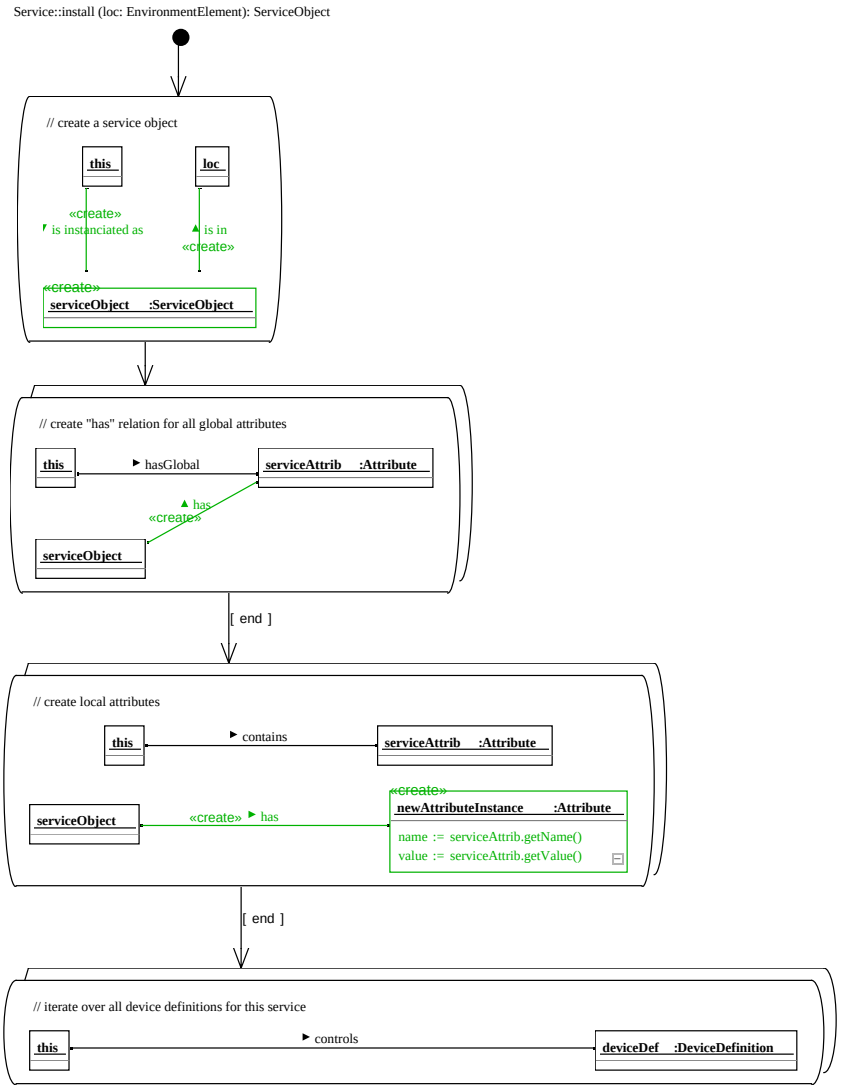

Abbildung 6.25: Activity-Diagramm: Dienstinstallation (1)

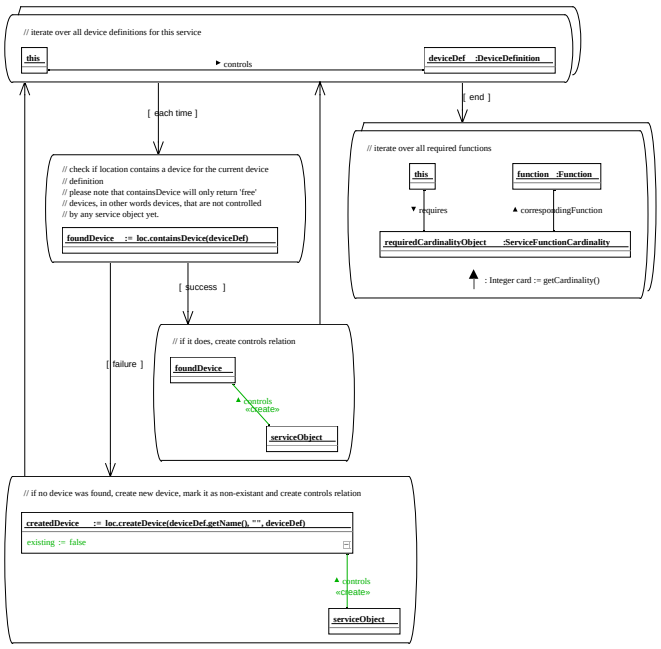

Abbildung 6.26: Activity-Diagramm: Dienstinstallation (2)

beschriebenen Beispiel handelt es sich bei allen Geräten um solche virtuellen Geräte, da hier ein Ausrüstszenario (vgl. Kapitel 2.3) betrachtet wird. In dem dort beschriebenen Haus sind keine steuerbaren Geräte vorhanden, daher müssen diese erst gekauft und installiert werden.

Das Anlegen der Geräteinstanz erfolgt wieder mit Hilfe einer (ebenfalls grafisch spezifizierten) Methode der `Location`-Klasse, welche zusätzlich eine Reihe von Variablen setzt und notwendige Relationen kreiert. Auf eine detaillierte Erläuterung wird hier verzichtet.

Die Tatsache, dass eine solche Verzweigung in `success` und `failure` auf diese Weise funktioniert, ist nicht intuitiv verständlich und liegt vielmehr in der Art und Weise der Codeerzeugung seitens Fujabas begründet. Der erzeugte Quellcode des betreffenden Story-Patterns ist in Listing 6.11 zu finden. Die relevante Stelle ist durch den ebenfalls durch Fujaba erzeugten Kommentar *"ensure correct type and really bound"* gekennzeichnet. Nach dem Aufruf von `containsDevice` in der darüber liegenden Zeile wird durch den Aufruf von `JavaSDM.ensure` sichergestellt, dass es sich bei der Rückgabeinstanz tatsächlich um ein `Device`-Objekt handelt. Ist dies nicht der Fall, handelt es sich also um die Instanz einer anderen Klasse oder, wie im hier betrachteten Fall möglich, um einen Nullzeiger, dann wird eine Ausnahme erzeugt, so dass direkt in den durch das Schlüsselwort `catch` markierten Teil des Programmcodes gesprungen wird. Hier wird eine boolesche Variable namens `fujaba__success` auf den Wert `falsch` gesetzt. Anhand dieser Variable wird im weiteren Verlauf die Verzweigung zu den beiden durch `success` und `failure`-Kanten angebundenen Story-Patterns entschieden.

```
1  // for each flow
2  // check if location contains a device for the current device definition
3  // please note that containsDevice will only return 'free' devices, in
4  // other words devices, that are not controlled by any service object yet.
5  try
6  {
7    fujabaSuccess = false;
8    fujabatmpTypeCastObject = loc.containsDevice(deviceDef) ;
9    // ensure correct type and really bound
10   JavaSDM.ensure ( fujabatmpTypeCastObject instanceof Device ) ;
11   // explicit type cast foundDevice = (Device) fujabatmpTypeCastObject ;
12   fujabaSuccess = true ;
13 }
14 catch ( JavaSDMException fujabaInternalException )
15 {
16   fujabaSuccess = false ;
17 }
```

Listing 6.11: Behandlung von Nullzeigern durch Fujaba

Aus diesem Grund ist eine explizite Überprüfung des Ergebnisses von `containsDevice` nicht notwendig, was die grafische Modellierung vereinfacht. Wäre dies nicht der Fall, so wäre es erforderlich, in einem separaten Story-Pattern zu verifizieren, ob es sich beim Rückgabewert von `containsDevice` um eine Instanz der `Device`-Klasse oder um einen Nullzeiger handelt. Dies erledigt Fujaba jedoch ohne zusätzlichen Modellierungsaufwand. Von dieser praktischen Eigenheit wird im Folgenden noch häufiger Gebrauch gemacht.

Nach Durchführung dieser Schritte für alle gefundenen Gerätebeschreibungen erfolgt die Betrachtung der benötigten Funktionen. Zur Erinnerung sei noch einmal erwähnt, dass in dieser Variante der Installationsmethode lediglich notwendige Funktionen betrachtet werden. Eine vollständige Installation erfolgt durch den Aufruf von `installOptional`. Doch auch ohne die Betrachtung optionaler Funktionalitäten handelt es sich im Vergleich zur vorhergegangenen Geräteinstallation um den komplexeren Teil der Methode. Der entsprechende Teilgraph ist in Abbildung 6.27 dargestellt.

Das erste Story-Pattern modelliert eine Schleife, in der über alle Instanzen der `Function`-Klasse iteriert wird. Da zum Ausdruck von Kardinalitäten eine Hilfskonstruktion notwendig war, befindet sich eine Instanz der Kardinalitätenklasse `ServiceFunctionCardinality` zwischen der `Service`-Instanz, deren `install`-Methode aufgerufen wird, und der eigentlich benötigten Funktionalität. Über die Zugriffsmethode `getCardinality` wird die benötigte Kardinalität in der Variable `card` gespeichert.

Alle darüber hinaus in Abbildung 6.27 dargestellten Story-Patterns werden für jede Iteration dieser Schleife, also für jede gefundene Funktionalität durchgeführt.

Zunächst wird überprüft, ob im betrachteten Raum bereits eine Serviceinstanz installiert ist, welche die notwendige Funktionalität in ausreichendem Maße zur Verfügung stellt. Hierzu bietet die `EnvironmentElement`-Klasse eine Methode namens `containsSuitableServiceObject` an. Diese durchsucht den entsprechenden Raum inklusive aller `Sublocations`. Falls dort eine Dienstinstanz existiert, die noch eine ausreichende Menge der benötigten Funktionalität zur Verfügung stellt, so wird diese

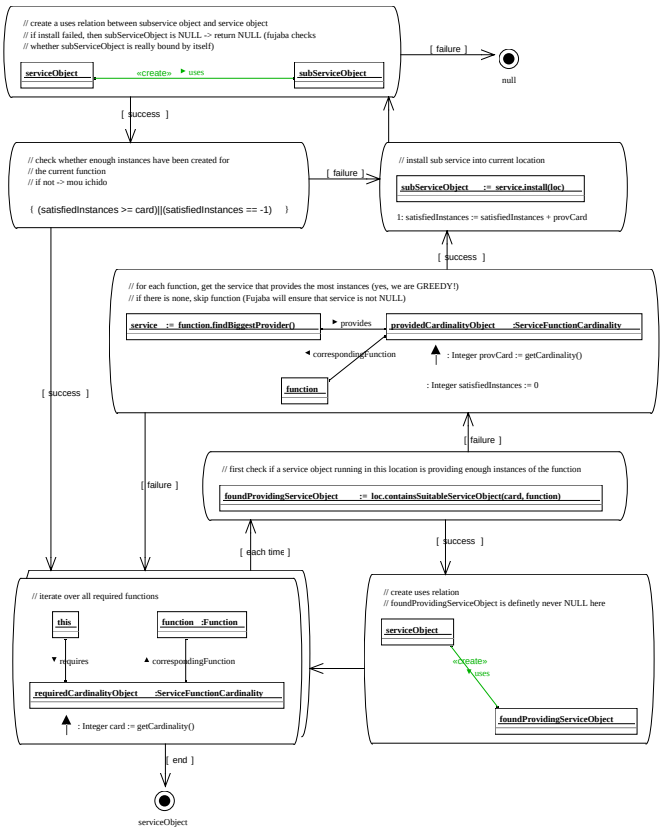

Abbildung 6.27: Activity-Diagramm: Dienstinstallation (3)

zurückgegeben. Existiert sie jedoch nicht, so erfolgt die Rückgabe eines Nullzeigers. Wie auch schon beim Auffinden von im Raum vorhandenen, noch unbenutzten Geräten, wird hier die von Fujaba durchgeführte Überprüfung des Rückgabedatentyps dazu benutzt, abhängig von Erfolg oder Misserfolg in verschiedene Story-Patterns zu verzweigen (vgl. Listing 6.11).

Wurde eine Dienstinstanz gefunden, welche eine ausreichende Anzahl der gesuchten Funktionalität anbietet, so wird eine `uses`-Kante zu diesem erzeugt, um auszudrücken, dass das Dienstobjekt, dessen Installationsprozess gerade durchgeführt wird, diese als Unterdienst benutzt.

Falls eine solche Dienstinstanz jedoch nicht existiert, so wird diejenige Dienstbeschreibung gesucht, welche die größte Anzahl der benötigten Funktionalität zur Verfügung stellt. Dies erfolgt mit Hilfe einer weiteren separaten Methode, welche den Namen `findBiggestProvider` trägt. Die entsprechende Kardinalität wird in einer Variable namens `provCard` gespeichert. Darüber hinaus wird eine weitere Variable deklariert und mit Null initialisiert, welche den Namen `satisfiedInstances` trägt. Diese wird für den Fall benötigt, dass selbst der größtmögliche Dienst mit einer einzelnen Instanz nicht genügend Funktionalität zur Verfügung stellt, um den Anforderungen gerecht zu werden.

Es erfolgt wieder eine Verzweigung abhängig davon, ob `findBiggestProvider` eine Dienstbeschreibung zurückliefert. Im Erfolgsfall wird eine Dienstinstanz des Unterdienstes im entsprechenden Raum angelegt. Hierzu erfolgt wiederum der Aufruf der `install`-Methode. Es handelt sich in diesem Fall *nicht* um eine Rekursion im herkömmlichen Sinne, da aufrufende und aufgerufene Methode zu zwei verschiedenen Instanzen der Dienstbeschreibungsklasse gehören. Als Rückgabewert erhält man die neu angelegte `ServiceObject`-Instanz. Im selben Schritt wird `satisfiedInstances` um den in `provCard` gespeicherten Wert erhöht. Im darauf folgenden Story-Pattern wird eine `uses`-Kante zwischen dieser Unterdienstinstanz und der aktuell betrachteten Dienstinstanz angelegt. Mit Hilfe der zuvor angelegten Variablen erfolgt eine Überprüfung, ob die benötigte Kardinalität vollständig erfüllt ist. Solange dies nicht der Fall ist, werden in einer Schleife weitere Instanzen des Unterdienstes erzeugt. Nach der Terminierung dieser Schleife wird zum ursprünglichen Story-Pattern zurückgekehrt, welches über alle benötigten Funktionen iteriert. Nach Beendigung dieser Iteration endet die `install`-Methode und liefert als Rückgabewert die neu generierte Dienstinstanz zurück.

6.6.3 Suche nach Dienstalternativen

Die `ServiceObject`-Klasse stellt eine Methode namens `findAlternatives` zur Verfügung, welche nach alternativen Diensten sucht, die dieselbe Funktionalität anbieten (siehe Abbildung 6.28). Diese werden in einem `HashSet`, einem von Java zur Verfügung gestellten Datentypen zur Verkapselung einer Menge von Objekten, gespeichert und zurückgegeben. Diese Methode unterliegt im Moment jedoch der Einschränkung, dass nur solche Dienstinstanzen ihre eigenen Alternativen suchen können, die lediglich eine einzige Funktionalität zur Verfügung stellen. Da `findAlternatives` in der Praxis aber zum Auffinden von alternativen Geräte-Controllern genutzt wird, welche diese Bedingung in der Regel erfüllen, stellt die Einschränkung kein Hindernis dar.

Wie bereits angedeutet liegt der größte Nutzen dieser Methode beim Auffinden von

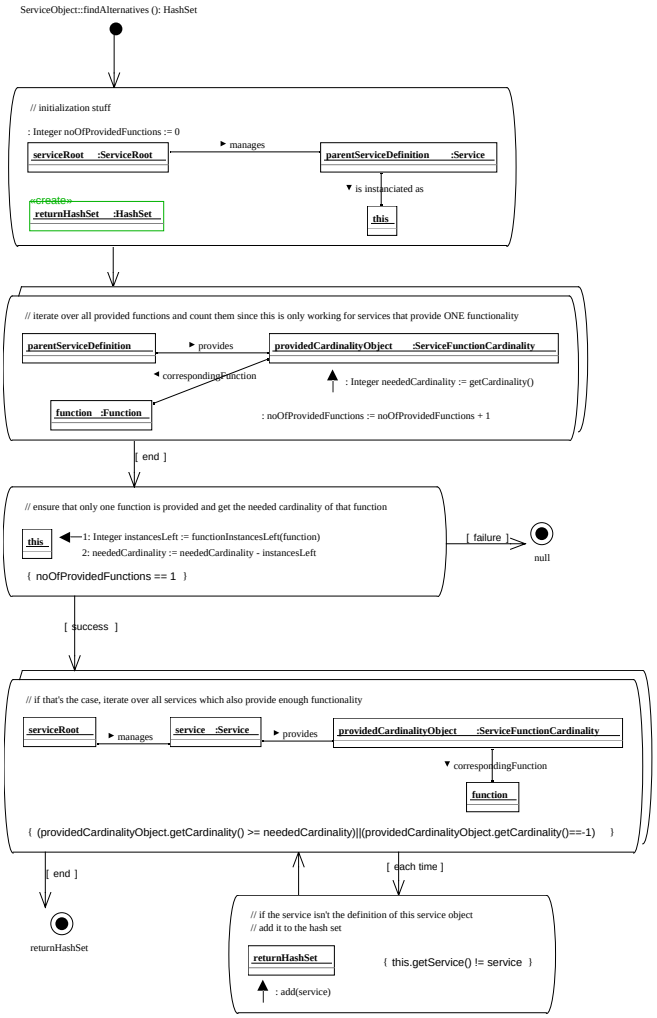

Abbildung 6.28: Activity-Diagramm: Suche nach Dienstalternativen

Geräten, welche alternativ in der automatisch erstellten Konfiguration genutzt werden können. Sie kommt daher im entsprechenden Teil des Konfigurationsassistenten zum Einsatz.

Im Vergleich zu den bisher beschriebenen grafischen Spezifikationen nutzt `findAlternatives` die von Fujaba gebotenen Möglichkeiten geschickt aus, so dass sie trotz ihrer relativen Kompaktheit einige interessante Aspekte veranschaulicht. Daher soll auf eine detaillierte Erläuterung nicht verzichtet werden.

Im ersten Schritt erfolgt die Deklaration von vier benötigten Variablen. Leicht erkennbar ist das Anlegen eines Integers namens `noOfProvidedFunctions`, in welchem festgehalten wird, wie viele Funktionen die betrachtete Dienstinstanz zur Verfügung stellt. Sie wird benutzt, um die oben erwähnte Einschränkung zu gewährleisten. Durch einen grünen Rahmen, welcher in der gedruckten Fassung dieser Arbeit als gepunktete Linie dargestellt ist, ist die Erzeugung eines Objektes vom Typ `HashSet` dargestellt, in welchem die gefundenen Alternativdienste hinterlegt und bei Beendigung der Methode zurückgegeben werden. Mit Hilfe der verbleibenden Kette aus drei Klassen werden simultan Referenzen auf zwei weitere Objekte erzeugt. Zum einen liefert sie eine Dienstbeschreibung namens `parentServiceDefinition`, bei der es sich um den zu diesem `ServiceObject` zugehörigen `Service` handelt. Zum anderen wird eine Referenz auf `ServiceRoot` erzeugt, mit deren Hilfe sich über alle spezifizierten Dienstbeschreibungen iterieren lässt.

Es folgt eine Schleife, in der über alle zur Verfügung gestellten Funktionen iteriert wird. Diese werden mit Hilfe der zuvor zu diesem Zweck deklarierten Variable, `noOfProvidedFunctions`, gezählt. Darüber hinaus wird eine weitere Variable namens `neededCardinality` angelegt, in der die angebotene Kardinalität gespeichert wird.

Im nächsten Story-Pattern erfolgt die Sicherstellung der Einschränkung, dass das betrachtete `ServiceObject` nur eine Funktionalität zur Verfügung stellen darf. Zu diesem Zweck ist in geschweiften Klammern ein entsprechender boolescher Ausdruck dargestellt. Sollte dieser nicht erfüllt sein, so wird in die mit `Failure` markierte Kante verzweigt und ein Nullzeiger zurückgegeben.

Im selben Schritt erfolgt der Aufruf der Methode `functionInstancesLeft`. Diese liefert als Ergebnis die verbleibende Anzahl an angebotener Funktionalität zurück. Um die von übergeordneten Diensten verwendete Kardinalität zu berechnen, wird dieses Ergebnis vom ursprünglich angebotenen Wert abgezogen und die Differenz in der Variable `neededCardinality` abgelegt. Dies erfordert die Abarbeitung der Schritte in einer bestimmten Reihenfolge, da erst das Ergebnis von `functionInstancesLeft` vorliegen muss, bevor man die Subtraktion durchführen kann. In der grafischen Spezifikation ist dies durch eine Nummerierung der beiden Schritte dargestellt.

Im darauf folgenden Schritt wird in einer Schleife über alle Dienste iteriert, welche die gleiche Funktionalität in ausreichendem Maße zur Verfügung stellen. Die Suche nach solchen spezifischen Diensten wird durch zwei Ausdrücke innerhalb des Story-Patterns erreicht. Mit Hilfe einer Kette aus vier Objekten werden hierzu zunächst alle Dienste identifiziert, welche die gesuchte Funktionalität mit einer beliebigen Kardinalität zur Verfügung stellen. Diese Kardinalität wird dann mit Hilfe einer Bedingung, die in geschweiften Klammern festgelegt ist, einer weiteren Prüfung unterzogen. Nur wenn diese zusätzliche Einschränkung bezüglich der zur Verfügung gestellten Menge ebenfalls erfüllt ist, erfolgt

die Verzweigung in das durch eine `each time`-Kante verknüpfte *Story Diagramm*. In diesem wird der gefundene Dienst dem Rückgabe-HashSet hinzugefügt, jedoch nur unter der Bedingung, dass es sich tatsächlich um einen unterschiedlichen Dienst handelt. Diese zusätzliche Abfrage ist nötig, da die vorhergehende Schleife auch den bereits installierten Dienst als vermeintliche Alternative ausfindig machen wird.

Nach der Terminierung dieser Schleife ist `returnHashSet` mit allen alternativ installierbaren Diensten gefüllt. Es erfolgt die Beendigung der Methode und die Rückgabe dieser Menge. An dieser Stelle wird deutlich, dass komplizierte Sachverhalte, wie die Suche nach Diensten mit ganz bestimmten Eigenschaften, in einem einzigen Story-Pattern zum Ausdruck gebracht werden können. Der Aufwand für eine explizite Programmierung wäre deutlich höher, da viele Zeilen Code notwendig sind. Darüber hinaus besteht eine hohe Wahrscheinlichkeit, dass bei einer Implementierung von Hand Fehler gemacht werden, deren Beseitigung viel Zeit erfordert.

Das Auffinden von alternativen Diensten bietet dem Benutzer beim Anpassen der Konfiguration an seine individuellen Wünsche eine wertvolle Unterstützung. So bietet diese Methode ihm im *Bau-* und *Ausrüstszenario* die Möglichkeit, automatisch vorgeschlagene Geräte durch die von ihm bevorzugten Marken zu ersetzen, ohne dadurch eine aufgrund der Kardinalitäten nicht realisierbare Konfiguration zu erstellen.

Verbesserungsmöglichkeiten

Bei der Suche nach Alternativdiensten wurde bereits auf die Einschränkung hingewiesen, dass die betrachteten Dienste lediglich eine Funktionalität zur Verfügung stellen dürfen. Dies kann bei bestimmten Dienstmodellierungen zu einer Reihe von Auffälligkeiten führen.

Gegeben seien beispielsweise zwei Beleuchtungsdienste, ein einfacher und ein erweiterter. Der einfache Dienst steuert lediglich Lampen, der erweiterte kann zusätzlich die Jalousien steuern. So wird der erweiterte Dienst als Alternative für den einfachen erkannt, da beide dieselbe Funktionalität anbieten. Da der erweiterte Beleuchtungsdienst aber auch die Jalousien steuert, ist der einfache Dienst keine Alternative für ihn, so dass ein Tauschvorgang nicht mehr rückgängig gemacht werden kann.

Die Suche nach Alternativdiensten bei der Betrachtung von mehreren Funktionalitäten gestaltet sich jedoch schwierig. Hier ist die Suche nach geeigneten Algorithmen notwendig, da Situationen auftreten können, in denen ein Dienst mit zwei Funktionen durch zwei Dienste mit je einer ersetzbar ist. So ist beispielsweise der erweiterte Beleuchtungsdienst, der Lampen und Jalousien steuern kann, durch den simplen Beleuchtungsdienst und einen weiteren Dienst zur Verdunkelung der Jalousien austauschbar. Umgekehrt sollte er aber auch als Alternative für die Kombination aus diesen beiden Einzeldiensten erkannt und angeboten werden. Hier ergibt sich auch das Problem einer geeigneten Aufbereitung für den Anwender. Wie können solche Situationen sinnvoll präsentiert werden? Unter Umständen hat der Benutzer zwar Interesse an einem Tausch des simplen Beleuchtungsdienstes, möchte den Verdunklungsdienst aber unbedingt behalten. Das Aufspüren und Präsentieren von Alternativdiensten stellt folglich eine Herausforderung dar, welche sich mit zunehmender Zahl von gleichzeitig angebotenen Funktionen verschärft.

6.6.4 Zusammenfassung

In diesem Abschnitt wurden die Implementierungsdetails des Konfigurierungs-Werkzeugs erläutert. Zu diesem Zweck wurden mehrere Methoden Schritt für Schritt anhand ihrer grafischen Spezifikation erläutert. Es wurde insbesondere der Algorithmus zur Dienstinstallation erläutert, welcher direkt kontrollierte Geräte findet oder bei Bedarf zur Umgebung hinzufügt und Funktionsabhängigkeiten auflöst. Bei dieser Auflösung wird auf bereits installierte Dienste zurückgegriffen und, falls notwendig, neue Unterdienste installiert. Hierbei werden Dienste mit großer Kardinalität bevorzugt, um möglichst wenige Unterdienste zu erzeugen. Obwohl diese Strategie nicht notwendigerweise zu einer kostenminimalen Installation führt, wird die Anzahl der notwendigen Geräte auf diese Weise gesenkt.

Bei der Behandlung von optionalen Funktionen ist es momentan nur möglich, alle oder keine solcher Funktionalitäten aufzulösen. Gerade im Hinblick auf Dienste mit mehreren optional benötigten Funktionen ist es jedoch wünschenswert, nur einen Teil davon zu betrachten. In diesem Fall wäre es auch möglich, Dienste zu formulieren, die ausschließlich optionale Funktionen besitzen. Ein Beispiel hierfür wäre ein kombinierter Beleuchtungs- und Verdunklungsdienst, der entweder Lampen oder Jalousien oder beides kontrollieren kann. Die Behandlung solcher Fälle ist jedoch ein Problem, welches nicht auf einfache Art und Weise lösbar ist. Hierzu reicht die simple Funktionsklasse des Datenmodells nicht mehr aus, da man zusätzliche Informationen darüber benötigt, welche der optionalen Funktionen betrachtet werden sollen und welche nicht. Um dies zu realisieren gibt es verschiedene Möglichkeiten, beispielsweise den Einsatz einer Kostenfunktion.

Problematisch ist die Verteilung der verschiedenen Teilprobleme der automatischen Konfiguration auf die verschiedenen Klassen des Modells. So wird diese erst dadurch möglich, dass unterschiedliche Klassen diverse Hilfsmethoden zur Verfügung stellen, welche aus der Installations-Methode heraus aufgerufen werden. Beispiele hierfür sind die Suche nach einem im Raum vorhandenen Gerät durch die Methode `containsDevice` der `EnvironmentElement`-Klasse und die Suche nach einer vorhandenen Dienstinstanz, welche durch `containsSuitableServiceObject` zur Verfügung gestellt wird. Dies sind nur zwei Beispiele für eine Vielzahl solcher Hilfsmethoden, auf welche im Verlauf des Konfigurierungsvorgangs zurückgegriffen wird.

6.7 Deployment-Werkzeug (Deployer)

Das Deployment-Werkzeug installiert die eHome-System-Konfiguration, also eine eHome-Modell-Instanz auf dem Service-Gateway im eHome und nimmt somit das eHome-System in Betrieb. Die Konfiguration ist das Ergebnis des gerade besprochenen Konfigurierungs-Werkzeugs. Der Start des Deployer-Werkzeuges ist die der Konfigurierungsphase folgende Aktion. Somit ist dies der Übergang zur Deployment-Phase.

Der Start des Deployment-Werkzeugs wird auch über die GUI des Spezifikationswerkzeugs durchgeführt. In dessen Werkzeugleiste befindet sich ein Knopf, mit dem das Deployment-Werkzeug gestartet werden kann. Das Deployment-Werkzeug analysiert die Dienst-Objekt-Kontext-Struktur (siehe Abschnitt 5.2.5) und führt die eHome-System-Installation mit den folgenden Schritten durch (vergleiche Listing 6.12):

```
1  ServiceList servicelist; // List of services to deploy
2
3  EhData ehdata = (EhData)   // Get the dataholder from ehdata
4    bc.getService(bc.getServiceReference(EhData.class.getName()));
5
6  DataHolder dataHolder = ehdata.getDataHolder(); // get DataHolder
7  ServiceRoot sRoot = dataHolder.getServiceRoot();
8
9  servicelist = getServicesToDeploy(sRoot);
10 loadBundles(servicelist);
11 startBundles(servicelist);
12 setServiceReferences(servicelist);
13 initializeServices(servicelist);
14 executeServiceObjects(servicelist);
```

Listing 6.12: Code der Hauptroutine des Deployers

1. `getServicesToDeploy`: Zuerst muss die Liste der zu installierenden Dienste ermittelt werden. Dies erfordert die Traversierung des Objekt-Graphen der aktuellen Dienst-Konfigurierung im Dienstobjekt-Kontext des eHome-Modells. Diese Traversierung wird über die `uses`-Beziehung zwischen den Objekten der Klasse `ServiceObject` (siehe Abbildung 5.10) durchgeführt. Die Traversion erfolgt rekursiv und folgt dieser `uses`-Beziehung. Im Beispiel aus Abbildung 5.12, würde die erstellte Liste die Dienste `Switching`, `Motion Controller`, `Person Detector`, `Sound Router` und `Music Follows Person` enthalten.

2. `loadBundles(servicelist)`: In diesem Schritt, wird jeder der gerade ermittelten Dienste auf das Service-Gateway geladen. Das Deployment-Werkzeug benutzt für diese Installation die Fähigkeiten, die über den Bundle-Context des OSGi-Gateway angeboten werden. In diesem Schritt, werden auch die Referenzen auf die Laufzeitinstanzen der einzelnen nun geladenen OSGi-Komponenten gespeichert.

3. `startBundles(servicelist)`: Nach dem Laden der Dienst-Komponenten, wird jeder Dienst im Rahmenwerk mit Hilfe der von OSGi zur Verfügung gestellten Startmethode gestartet. Dabei werden alle Dienste in sequentieller Reihenfolge gestartet. Wenn Fehler beim Start auftreten, wird der Zyklus für das Starten der Dienste bis zum Ende durchgeführt und es wird anschließend versucht, Dienste, die Probleme beim Start hatten, noch einmal zu starten. Dadurch hat das Rahmenwerk mehr Zeit, ungelöste Abhängigkeiten zwischen den Komponenten zu erkennen und gegebenenfalls aufzulösen. Diese Vorgehensweise ist zwar nicht besonders elegant, da angenommen wird, dass die erstellte Konfiguration korrekt ist, dennoch werden einige Probleme, die bei einem parallelen Start auftreten würden vermieden. Auch ist dieser Schritt wegen des sequentiellen Starts, sehr leicht zu debuggen, so dass Fehler in der Dienstinitialisierung leicht zu finden sind.

   Die Installation und das Starten der Dienste kann, wie gerade angedeutet, auch parallel in mehreren Threads passieren und mittels Bundle-Events und zugehöriger Listener (`BundleListener` interface), die das OSGi framework API zur Verfügung stellt, synchronisiert werden. Jedoch waren bei dieser Methode erhebliche

Mängel in Bezug auf den Class-Loader der getesteten OSGi-Frameworks festzustellen, weshalb die sequentielle Methode bevorzugt wird.

4. `setServiceReferences(servicelist)`: Die gestarteten Dienste werden in der eHome-Modell-Instanz registriert. Das Deployment-Werkzeug implementiert das Service-Listener-Interface (`ServiceListener`) der OSGi-API. Dadurch ist es möglich, Dienst-Ereignisse (service events) des OSGi-Rahmenwerks zu empfangen, welche über Änderungen des Lebenszyklus von Komponenten informieren. Nachdem ein Dienst gestartet wurde, sendet das Rahmenwerk ein Service-Registration-Event. Wird dieses empfangen, wird die mit diesem Event empfangene Referenz in die zu dem gerade gestarteten Dienst korrespondierenden Dienst-Objekten als Implementierungsreferenz eingetragen. Das heißt, dass die Beziehung `has runtime component` zwischen dem Dienstobjekt und dem gerade gestarteten Dienst, welcher vom Typ `EhService` sein muss, gesetzt wird. Zum Beispiel hat das `Music Follows Person` Objekt in Abbildung 5.12 eine Referenz auf die Implementation des `MusicFollowsPerson`-Dienstes, welches auf dem Service-Gateway installiert ist.

5. `initializeServices(servicelist)`: Wenn alle Dienste registriert sind, werden alle Dienste über die `init()`-Methode ihres `EhService`-Interface (siehe Abbildung 5.10) initialisiert. Diese zweite Initialisierung wurde erforderlich, um sicherzustellen, dass beim Aufruf dieser bereits alle Abhängigkeitsprobleme gelöst sind. Auf Dauer sollte diese mit der `start` Methode der eHome-Dienst-`Activator`-Klasse des Rahmenwerks zusammengefasst werden.

6. `executeServiceObjects(servicelist)`: Schließlich werden alle Dienstobjekte der Top-Level-Dienste in ihrem Dienstkontext mittels des Aufrufs der Methode `execute(ServiceObject so)` des `EhService`-Interface ausgeführt. Dies übergibt dem Dienst die Informationen der Kontexte seiner Dienstobjekte. Durch die Übergabe des `ServiceObject so` werden Informationen über andere benötigte Dienste, die physische Umgebung (environment), Zustände und Attribute, des aktuellen und benachbarter Dienstobjekte und gesteuerter Geräte und die abstrakte Beschreibung des Dienstes selbst zur Verfügung gestellt. Diese kann jetzt zur Laufzeit sehr leicht aus dem Objektgraphen inferiert werden (siehe Abbildungen 5.9 und 5.10). Während dieses Methodenaufrufs kann der Dienst die Zustandsinformationen (also Objekte vom Typ der Klasse `State`) anlegen und sie mit den entsprechenden Objekten des Typs `ServiceObject` verbinden (siehe auch die Struktur des Dienstobjekt-Kontext in Abschnitt 5.2.5). Zum Beispiel erstellt der Switching-Dienst für das Objekt `Switch` in Abbildung 5.12 einen Zustand `switch` mit dem Wert (value) `1`.

   Da nur die `execute`-Methode der top-level Dienste aufgerufen werden, rufen diese in der Regel selbst die `execute`-Methode ihrer Unterdienste auf. Dies hängt allerdings von der Art der Unterdienste und somit von der Dienstimplementierung ab. Es wird von den Dienstentwicklern dann benutzt, wenn Dienste, die in der Dienstebenutzunghierarchie an höherer Stelle stehen, Zustandsinformationen darunterliegender Dienste benötigen, um deren Initialisierung zu triggern. Es kann

aber auch dazu benutzt werden, ein synchrones Verhalten beim Ansprechen eines Unterdienstes zu realisieren, wie zum Beispiel beim e-Mail-Sende-Dienst des Sicherheitsdienstes. Dieser sendet bei jedem Aufruf der `execute`-Methode eine Nachricht mit den eingestellten Attributen.

Das Deployment-Werkzeug arbeitet mit und wurde auf zwei unterschiedlichen OSGi Implementationen getestet: auf dem proprietären mBedded Server 5.x [Proa] und auf der Open-Source-Implementation Knopflerfish [Kno04].

6.8 Vereinfachter Dienstaktivator

Um die Dienstentwicklung in OSGi zu vereinfachen, wurde dem DataHolder noch eine `EhActivator`-Klasse hinzugefügt, die die durchschnittliche Anzahl der Zeilen für einen OSGi-Aktivator (siehe für die Beschreibung des Aktivators Abschnitt 4.3) von etwa vierzig Zeilen auf zehn reduziert.

Ein typischer Aktivator vor dieser Vereinfachung ist in Listing 6.13 abgebildet. Er implementiert eine start-, eine stop- und eine register-Methode. Seine Funktion besteht allerdings nur darin, den eigenen OSGi-Kontext festzustellen und die Bundle-Schnittstelle und -Implementation beim OSGi-Rahmenwerk zu registrieren, so dass diese von anderen Bundles gefunden werden können. Natürlich kann in der Start-Methode noch anderes initialisiert werden, da in dieser Arbeit aber die Kommunikation hauptsächlich über die getypte eHome-Modell-Instanz läuft und eine Init-Methode zur Verfügung steht, ist der Zugriff auf den OSGi-Kontext nur selten nötig. Dienste, die diesen nicht benötigen, können also die hier besprochene Vereinfachung benutzen.

Ein vereinfachter Aktivator ist in Listing 6.14 zu sehen. Hier werden nur die nötigen Daten für die Registrierung übergeben. Listing 6.15 zeigt die Klasse `EhActivator`, die die Vereinfachung implementiert. Diese übernimmt die Implementation der `Start`- und `Stop`-Methoden (Zeile 18-23) und die Anmeldung des Bundles im OSGi-Rahmenwerk (Zeile 25-38). Auch ermöglicht sie die Erstellung einer init-Methode, die den BundleContext und die Registrierungsinformationen übergeben bekommt. Notfalls kann also auch hier eine direkte Interaktion mit dem OSGi-Rahmenwerk stattfinden oder der BundleContext zwecks späterer Verwendung zwischengespeichert werden.

Für die Vorgehensweise, einen kompletten Dienst für den eHomeConfigurator zu implementieren, siehe das folgende Kapitel 7.

6.9 Ergebnisse

Tabelle 6.1 stellt die entwickelten und erzeugten Codezeilen der eHomeConfigurator-Werkzeug-Suite zusammen. Da das Projekt unter der LGPL-Lizenz steht, hängt an jeder nicht automatisch erzeugten Java-Datei der Verweis auf diese Lizenzbestimmungen. Dies sind jeweils 28 Zeilen, die jeweils als Kommentar an Dateien angehängt und nicht selbst entwickelt oder erzeugt wurden. Deshalb werden hier auch die Codezeilen in einer um die Lizenzen bereinigten Zahl gezeigt. In den Veröffentlichungen [NM06b, NM06a] wurde diese Bereinigung nicht vorgenommen, so dass das dort gezeichnete Bild für die durch den eHomeConfigurator erreichte Unterstützung noch etwas zu negativ ausfällt.

```
1  package de.rwth.i3.ehome.parportchecker.impl;
2
3  import java.util.Hashtable;
4
5  import org.osgi.framework.BundleActivator;
6  import org.osgi.framework.BundleContext;
7  import org.osgi.framework.BundleException;
8  import org.osgi.framework.ServiceRegistration;
9
10 import com.rweplus.ehome.powerutil.Log;
11
12 import de.rwth.i3.ehome.parportchecker.WatchData;
13 import de.rwth.i3.ehome.parportchecker.impl.ParportCheckerImpl;
14
15 public class Activator implements BundleActivator {
16
17     ServiceRegistration servRegistration;
18     ParportCheckerImpl servImpl;
19
20
21     public void start(BundleContext bc) throws BundleException {
22         register (bc);
23     }
24
25     public void stop(BundleContext bc) throws BundleException {
26         servImpl.clearWatches();
27     }
28
29     private void register(BundleContext bc) throws BundleException {
30         try {
31             servImpl = new ParportCheckerImpl();
32             Hashtable properties = new Hashtable();
33             properties.put("name", "ParportChecker");
34             properties.put("description",
35     "service for checking the data-bits of the parport for events");
36             servRegistration =
37                 bc.registerService(
38                   "de.rwth.i3.ehome.parportchecker.ParportChecker",
39                   servImpl, properties);
40         } catch (Exception exc) {
41             throw new BundleException(exc.getMessage(), exc);
42         }
43     }
44
45 }
```

Listing 6.13: Ursprünglicher Aktivator

```
1  package ehome.ehsecurity;
2  import org.osgi.framework.BundleContext;
3  import ehome.ehdata.tools.EhActivator;
4
5  public class Activator extends EhActivator
6  {
7      public void init(BundleContext bc) {
8          this.set(new EhSecurityImpl(), "EhSecurity",
9                  "Security control service");
10     }
11 }
```

Listing 6.14: `Activator.java` des top-level Sicherheitsdienstes `Security` (Kopie von Listing 7.1)

```
1  package ehome.ehdata.tools;
2
3  import java.util.Hashtable;
4
5  import org.osgi.framework.BundleActivator;
6  import org.osgi.framework.BundleContext;
7  import org.osgi.framework.BundleException;
8  import org.osgi.framework.ServiceRegistration;
9
10 public abstract class EhActivator implements BundleActivator
11 {
12     private String serviceName, description;
13     Object servImpl;
14     ServiceRegistration servRegistration;
15
16     public abstract void init(BundleContext bc);
17
18     public void start(BundleContext bc) throws BundleException {
19         init(bc);
20         register (bc);
21     }
22
23     public void stop(BundleContext bc) throws BundleException {}
24
25     private void register(BundleContext bc) throws BundleException {
26         try {
27             Hashtable properties = new Hashtable();
28             properties.put("name", serviceName);
29             properties.put("description", description );
30             // bad way to find Interface-Name
31             String implname = servImpl.getClass().getName();
32             String nameonly = implname.substring(
33                         0, implname.lastIndexOf("Impl") );
34             servRegistration =
35                 bc.registerService(nameonly, servImpl, properties);
36         } catch (Exception exc)
37         { throw new BundleException(exc.getMessage(), exc); }
38     }
39
40     /* set all values at once */
41     public void set( Object servImpl, String serviceName,
42                                         String description) {
43         this.servImpl = servImpl;
44         this.description = description;
45         this.serviceName = serviceName; }
46
47     // Getter-, Setter-Methods
48     public Object getServImpl() { return servImpl; }
49     public void setServImpl(Object servImpl) {
50         this.servImpl = servImpl; }
51     public String getDescription() { return description; }
52     public void setDescription(String description) {
53         this.description = description; }
54     public String getServiceName() { return serviceName; }
55     public void setServiceName(String serviceName) {
56         this.serviceName = serviceName; }
57 }
```

Listing 6.15: Aktivatorvereinfachungs-Klasse `EhActivator` für eHomeConfigurator-Dienste

Betrachteter Teil, Komponente oder Datei	Javacode, auch generierter Code	Anzahl Dateien	Javacode, aus Modell generierter	Anzahl Dateien	Javacode, ohne Lizenzen und generierten Code	Anzahl Dateien	Zeilen der Manifestdateien	Codezeilen, Summe ohne Lizenzen
Gesamtes eHomeConfigurator-Projekt incl. Diensten	40391	238	12895	26	21560	212	612	22172
DataHolder (ehdata)	14026	31	12895	26	991	5	54	1045
Spezifikations-Werkzeug (specificator)	17024	99	0	0	14252	99	101	14353
Deployment-Werkzeug (deployer)	563	3	0	0	479	3	15	494
Alle eHome-Dienste	8778	105	0	0	5838	105	442	6280

Sonstiger Code	Codezeilen
`clewareshell.cpp`	141
clewareshell-`Makefile`	13
`build.xml`	231
script-code	296
`activities.xml`	505
Summe	1186

Gesamtsummen	Codezeilen
eHomeConfigurator-Projekt	41577
Projekt ohne gen. Code und Lizenzen	23358
Alle eHome-Dienste (ohne ehdata, spec. und depl.)	6280
Projekt ohne eHome-Dienste (nur ehdata, specificator und deployer ohne gen. code u. Lizenzen)	17078

Tabelle 6.1: Kodierungsaufwand eHomeConfigurator und eHome-Dienste

Der Vergleich der bereinigten Zahlen für den erzeugten Code (12895 Zeilen) mit dem gesamten manuell programmierten Code des eHomeConfigurators ohne die eHome-Dienste (17078 Zeilen, also insgesamt bereinigt 29973 Zeilen) zeigt, dass mehr als ein Drittel der unterstützenden Werkzeugsuite automatisch aus dem Modell generiert werden konnte. Das bestätigt den Eindruck, dass durch den Einsatz von Fujaba ein erheblicher Entwicklungsaufwand eingespart werden konnte.

An dieser Stelle wird auch klar, dass wesentlich mehr Entwicklungsaufwand in die eHomeConfigurator-Werkzeugsuite als in die Entwicklung der Beispieldienste geflossen ist (6280 Zeilen Code für die Dienstentwicklung und 17078 Zeilen für die Werkzeugsuite).

Tabelle 6.2 zeigt den Kodierungsaufwand des vereinfachten Sicherheitsdienstes im Rahmen der eHomeConfigurator-Werkzeugsuite, aufgespalten in seine Unterdienste. Die Entwicklung einfacher Dienste nimmt im Durchschnitt etwa 150 Zeilen in Anspruch. Bei komplexeren Treibern sind es mehr als 300. Dennoch ist diese Zahl überschaubar. Die Erfahrungen aus diesem Projekt zeigen, dass ein einfacher Erweiterungsdienst ohne Treiberfunktionalität innerhalb eines Personentages entwickelt und spezifiziert werden kann. Entwicklung von Treibern bzw. Basisdiensten brauchen in der Regel zwei bis drei Manntage.

Zur Bewertung und dem Vergleich dieser Ergebnisse aus Kapitel 4 vergleiche Abschnitt 7.8.

Bundle	**Javacode, gesamtes Bundle**	**Anzahl Dateien**	**Javacode, ohne Lizenz**	**Javacode, Aktivator**	**Aktivatorcode, ohne Lizenz**	**Codezeilen, Manifest**	**Codezeilensumme, ohne Lizenz**
ehsecurity	248	3	164	49	21	13	177
ehilluminate	236	3	152	50	22	15	167
ehintrusiondetector	205	3	121	50	22	13	134
ehemail	205	3	121	86	58	15	136
clewarecontrol	400	4	288	71	43	13	301
ehlegomovementdetector	223	3	139	47	19	14	153
ehlegomotioncontrol	434	5	294	72	44	13	307
ehlegolampcontrol	228	3	144	48	20	16	160
Summe	2179	27	1423	473	249	112	1535

Tabelle 6.2: Kodierungsaufwand vereinfachter Sicherheitsdienst im Rahmen der eHomeConfigurator-Werkzeugsuite

Kapitel 7

Nutzung & Bewertung eHomeConfigurator

Die im Rahmen dieser Arbeit entwickelte Werkzeugsuite eHomeConfigurator unterstützt den SCD-Prozess. Dieses Kapitel beschreibt am Beispiel des in Abschnitt 3 auf Seite 27 beschriebenen Sicherheitsdienstes und seiner Unterdienste, wie die Werkzeugsuite in der Praxis eingesetzt werden kann. Die Konfigurierung wird für alle vorgestellten Dienste des Feinszenarios (siehe Abschnitt 2.2.1) durchgeführt. Anschließend wird der in der Praxis entstehenden Entwicklungsaufwand und die tatsächlichen Auswirkungen auf den Entwicklungsprozess von eHome-Systemen bewertet und mit den Ergebnissen der manuellen Konfigurierung aus Kapitel 4 verglichen.

Bevor der Kunde die Werkzeugsuite zur Auswahl eines Dienstes nutzen kann, müssen der Dienstanbieter oder der Softwarekomponenten-Entwickler eine Menge von Softwarekomponenten (in OSGi Bundles genannt) und ihre Dienstbeschreibungen in einem semantischen Kontext (das heißt, in Bezug auf die angebotenen und benötigten Funktionalitäten) zur Verfügung stellen. Ferner müssen Gerätebeschreibungen vorhanden sein. Um einen Dienst zu spezifizieren, müssen alle seine Funktionalitäten-Anforderungen (Importe) und Exporte gegeben sein. Eine URL, die angibt, wo sich der Binärcode der realisierenden Softwarekomponente befindet und das Merkmal, ob es sich hier um einen top-level Dienst handelt, müssen festgelegt werden. Bisher werden alle Editoren zur Spezifikation angezeigt. Es ist aber durchaus denkbar, gewisse Editor-Panels für bestimmte Benutzergruppen wie den Endkunden, den Dienstentwickler oder den Dienstanbieter zu deaktivieren.

Dieses Kapitel spielt – am Beispiel der Konfigurierung des Sicherheitsdienstes und seiner vier Unterdienste, die für die Realisierung benötigt werden – die Benutzung des eHomeConfigurators durch. Weiterhin wird am Beispiel des Sicherheitsdienstes und seiner Unterdienste gezeigt, wie der Quelltext für Dienste aussieht und wie hoch deren Entwicklungsaufwand ist.

Für die im Folgenden verwendeten Funktionalitäten vergleiche das Funktionalitätenbild aus Abbildung 5.3 auf Seite 129.

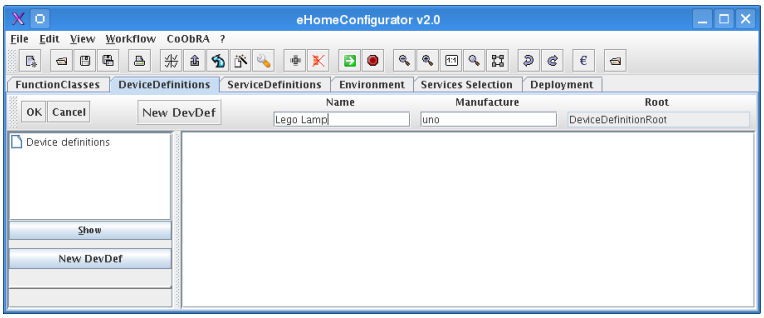

Abbildung 7.1: Spezifikation neues Gerät, Beispiel Lego-Lampe. Schritt 1

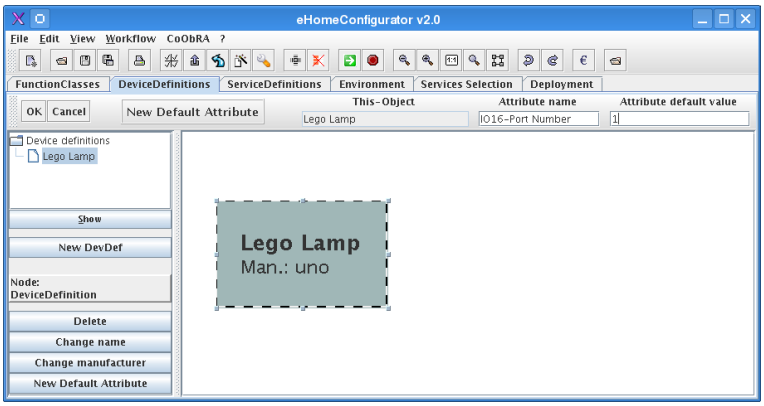

Abbildung 7.2: Spezifikation neues Gerät, Beispiel Lego-Lampe. Schritt 2

7.1 Gerätespezifikation

Die Geräte werden vom Dienstanbieter spezifiziert. Im Sicherheitsdienst werden die Geräte Lego-Lamp, Webcam und SoundSocket verwendet. Um ein neues Gerät wie zum Beispiel die Lego-Lampe zu spezifizieren, muss im eHomeConfigurator das Editor-Panel `DeviceDefinitions` ausgewählt werden. Dort muss dann der Knopf mit der Aufschrift `New DevDef` angeklickt werden. Daraufhin müssen in Name der Name des neuen Gerätes, hier "Lego Lamp", und in Manufacturer der Hersteller, hier "uno" für den Autor dieser Arbeit, eingetragen werden (siehe Abbildung 7.1). Dies ist dann via Druck auf den Knopf `OK` zu bestätigen. Daraufhin sollte das neu erzeugte Gerät ausgewählt werden und der Knopf `New Default Attribute` betätigt werden. Dort ist der Attribut-Name, hier "IO16-Port Number", und der default-Wert, hier "1", des Attributs anzugeben (siehe Abbildung 7.2). Das Ergebnis nach Bestätigung über den `OK`-Knopf ist in Abbildung 7.3 zu sehen. Dies ist die Spezifikation der Lego-Lampe. Die IO16-Port-Nummer gibt die Adressleitung einer parallelen Schnittstelle an, mit der die Lampe verbunden ist. Dies kann eine Zahl von 1 bis 5 sein.

Weiterhin müssen für den Sicherheitsdienst noch die Webcam (Abbildung 7.4) und

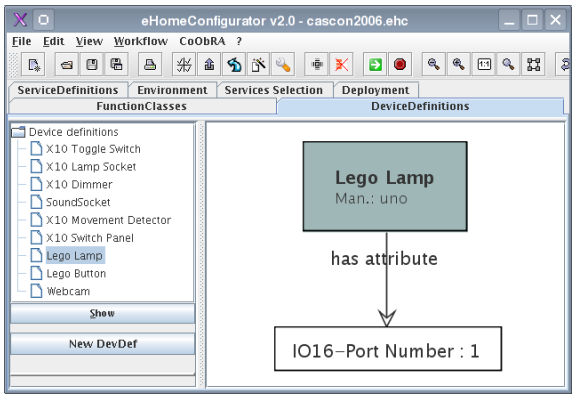

Abbildung 7.3: Spezifikation Lego-Lampe

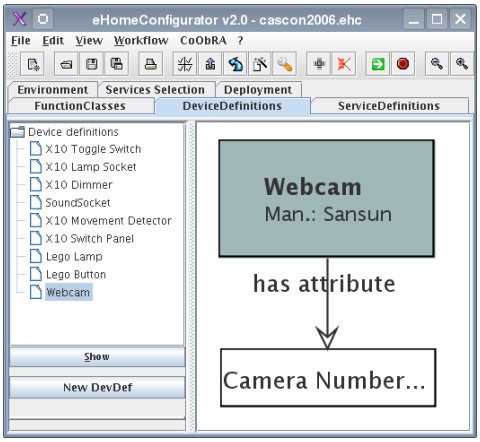

Abbildung 7.4: Spezifikation Webcam

der SoundSocket (Abbildung 7.5) spezifiziert werden. Die Webcam besitzt als Attribut die Kameranummer (`Camera Number`), welche die Nummer der angeschlossenen Kamera angibt. Beim Lego-eHome-Demonstrator gibt es drei-Kameras, so das hier später eine Zahl von 1 bis 3 eingetragen werden kann. Der SoundSocket besitzt zwei Attribute. Das eine ist `Destination` und gibt den Zielrechner an, auf dem der Klang des benutzenden Dienstes ausgegeben werden soll. Beim Lego-eHome-Demonstrator wird dies immer "localhost" sein. Im Attribut `Card` wird die Nummer der Soundkarte des Zielrechners angegeben. Da in dem Residential-Gateway des Lego-eHome-Demonstrators drei Soundkarten installiert sind, werden hier bei der Konfigurierung Nummern von 0 bis 2 eingetragen.

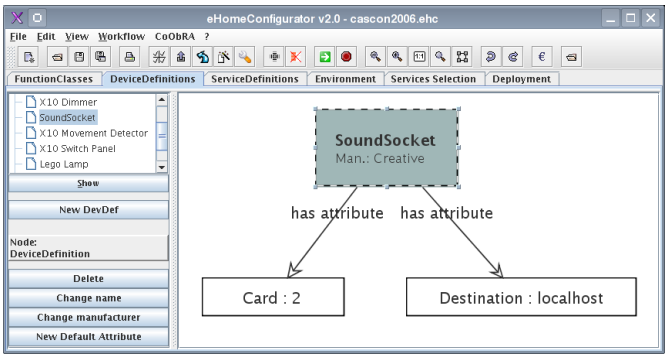

Abbildung 7.5: Spezifikation SoundSocket

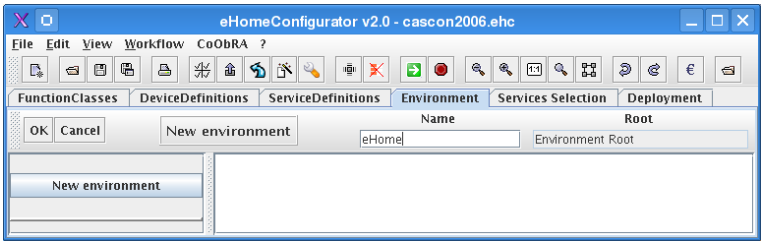

Abbildung 7.6: Neue eHome-Umgebung anlegen

7.2 Umgebungsspezifikation

Die Umgebung soll idealerweise in Zukunft automatisch analysiert oder aus einem digitalen Architekturplan extrahiert werden können. Bis dahin werden allerdings der Bewohner bzw. Kunde oder ein Techniker diese spezifizieren.

Um eine neue eHome-Umgebung zu spezifizieren, muss das `Environment`-Panel ausgewählt werden. Dort kann über den Knopf mit der Beschriftung `New environment` eine neue eHome-Umgebung angelegt und benannt werden (siehe Abbildung 7.6).

Nach Auswahl dieser, kann mittels des Knopfes mit Beschriftung `New Location` ein neue Umgebungslokation erzeugt werden. Hier im Beispiel wird das Wohnzimmer (living room) erzeugt (siehe Abbildung 7.7).

Es ist möglich Unterlokationen wie eine Küchenecke von Umgebungslokationen zu erzeugen. Dies geschieht durch Anwahl einer Umgebungslokation und anschließend dem Knopf mit der Beschriftung `New Sublocation` (siehe Abbildung 7.8).

Um zwei Lokationen zum Beispiel mit einer Tür zu verbinden, muss zuerst eine der Lokationen und dann der Knopf mit der Beschriftung `Connect to Loc` ausgewählt werden. Anschließend kann der Name dieser Verbindung und die zu konnektierende Ziellokation angegeben werden (siehe Abbildung 7.10).

Um installierte Geräte zur Umgebung hinzuzufügen, müssen zuerst, wie im vorherigen Abschnitt, Geräte spezifiziert worden sein. Anschließend muss im Graphen ein Lo-

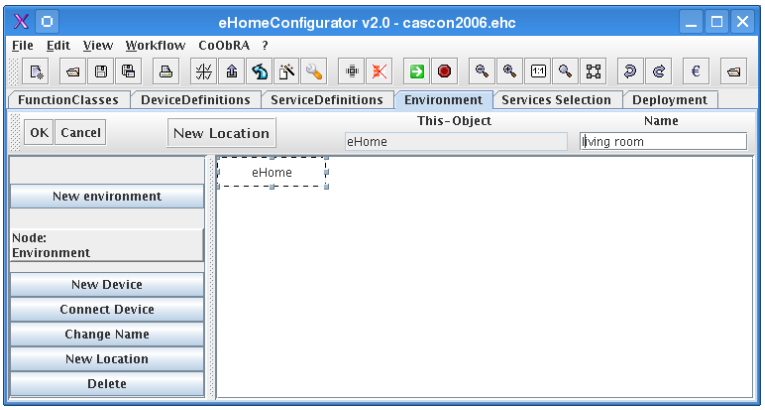

Abbildung 7.7: Neue Umgebungslokation anlegen

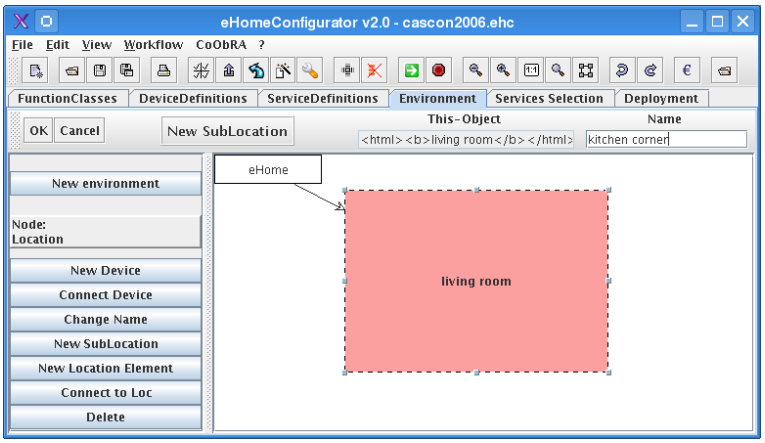

Abbildung 7.8: Neue Unterlokation anlegen

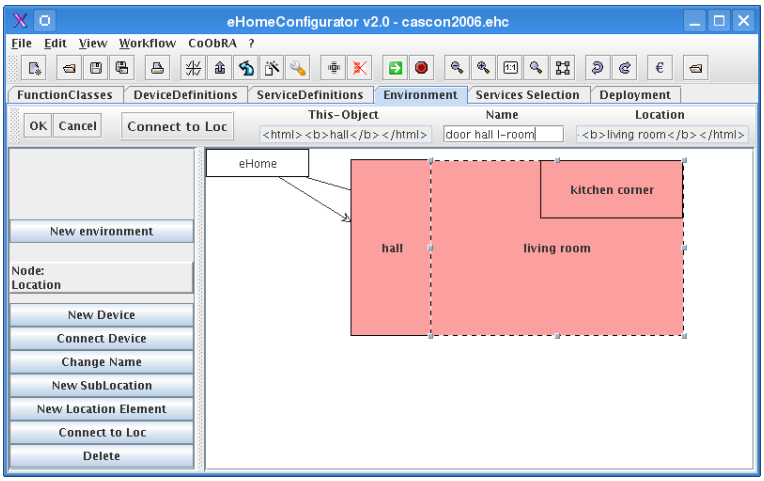

Abbildung 7.9: Verbindung von Lokationen

kationsobjekt oder Verbindungsobjekt selektiert werden und darauffolgend der Knopf mit der Beschriftung `New Device` benutzt werden. Im erscheinenden Dialog lässt sich dann ein Name für das Gerät und seine Definition auswählen.[1]

Eine minimale Umgebung mit einem vorinstalliertem Gerät sieht dann wie in Abbildung 7.11 aus.

7.3 Dienstespezifikation

Die Dienstespezifikation wird vom Dienstanbieter oder den Dienstentwicklern durchgeführt. Um einen neuen Dienst im eHomeConfigurator zu spezifizieren, muss analog zur Gerätespezifikation im eHomeConfigurator das Editor-Panel `ServiceDefinitions` und dann der Knopf `New Services` angewählt werden. Als Beispiele wird hier der top-level Dienst des Sicherheitsdienstes verwendet. Deswegen wird in dem Beispiel auch "Security Service" in den Dialog (siehe Abbildungen 7.12), eingetragen. Auch dieser Dialog ist mit `OK` zu bestätigen. Das Ergebnis ist in Abbildung 7.13 zu sehen. Nach der Auswahl des `Security Service`-Definitionsobjekts im Editor, sind links die möglichen Aktivitäten zu sehen, von denen `Change URL` auszuwählen ist. Im erscheinenden Dialog kann dann die URL der für diesen Dienst zu ladenden Softwarekomponente angegeben werden, hier "ehsecurity.jar" (siehe Abbildung 7.14). Analog lassen sich Type (hier top für top-level-Service) und Id (hier 14) setzen, siehe Abbildung 7.15. Mittels des Knopfes mit der Aufschrift Requires Function können aus einer Liste eine benötigte Funktionalität und deren benötigte Kardinalität ausgewählt werden. Wie in Abbildung 7.16 zu sehen, wird hier `raise.alarm` mit der Kardinalität 1 ausgewählt.

Es folgt eine komplette Liste der für die Realisierung des Sicherheitsdienstes zu ent-

[1] Der angezeigte `Address`-Parameter wird nicht mehr ausgewertet. Er wurde in einer älteren Version des Spezifizierungs-Werkzeuges benutzt.

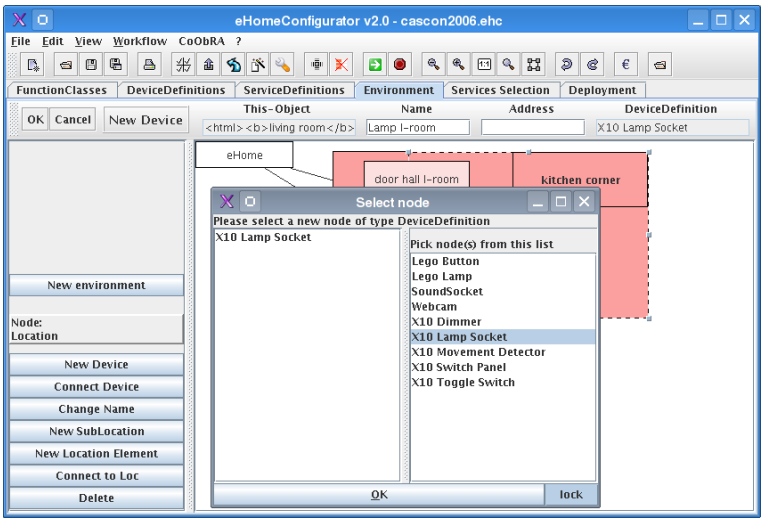

Abbildung 7.10: Verbindung von Lokationen

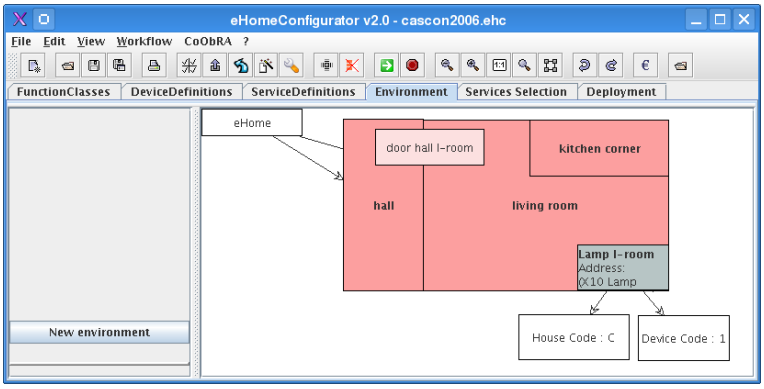

Abbildung 7.11: Minimale Umgebung mit vorinstalliertem Gerät

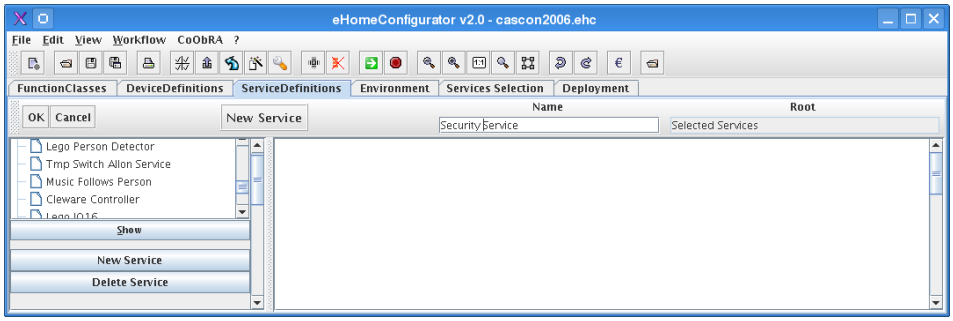

Abbildung 7.12: Anlegen eines neuen Dienstes

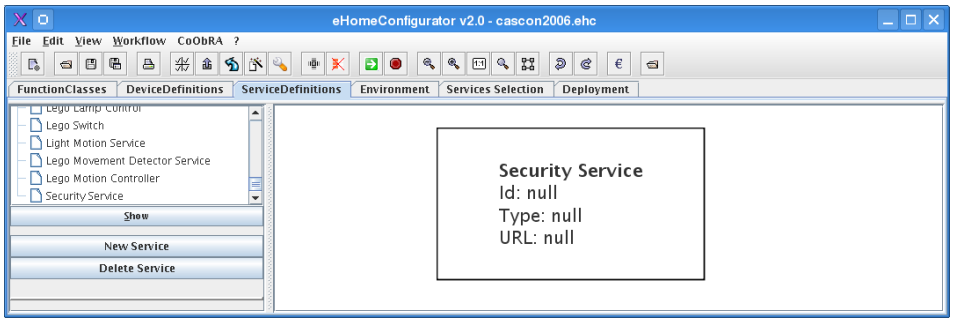

Abbildung 7.13: Ergebnis eines gerade neu angelegten Dienstes

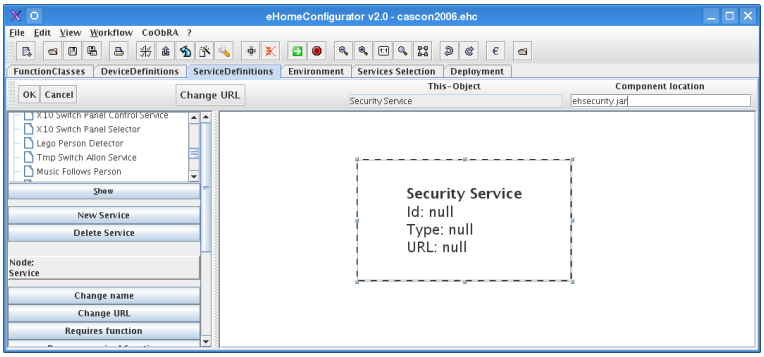

Abbildung 7.14: URL ändern in neuem Dienst

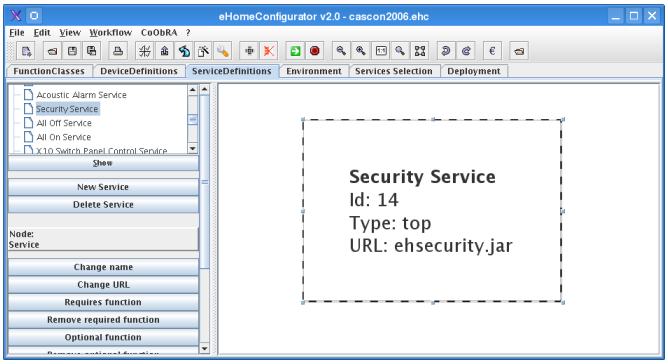

Abbildung 7.15: Attributierter neuer Dienst

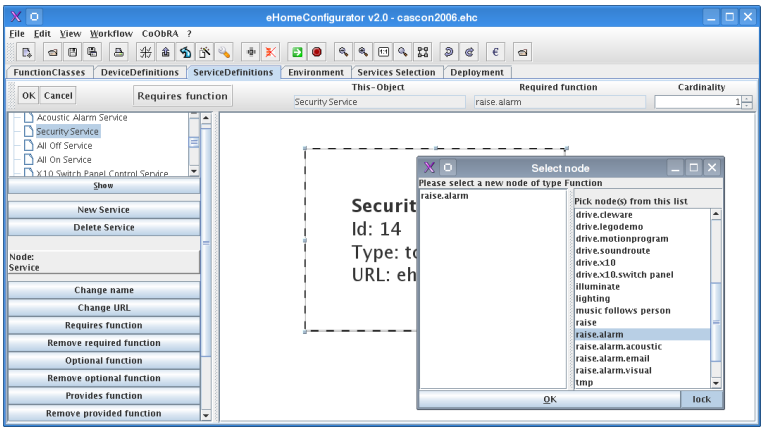

Abbildung 7.16: Hinzufügen von Funktionalitätsabhängigkeiten.

wickelnden Dienste und Unterdienste. Diese müssen innerhalb des eHomeConfigurators wie folgt spezifiziert werden:

Security: Dies ist der entsprechende top-level Dienst. Der bei aktiviertem Alarm (`detect.alarm activition`) und der Detektion eines unberechtigten Eindringens (`detect.intrusion`) Alarm auslöst (`raise.alarm`). Die Spezifikation ist in Abbildung 7.17 zu sehen.

Intrusion Detection: Der Dienst Intrusion-Detection stellt die Funktionalität `detect.intrusion` so oft für einen Raum zur Verfügung wie er benötigt wird (an der -1 zu erkennen). Siehe Abbildung 7.19. Er benötigt die Funktionalität `detect.movement`. Hier könnten auch weitere Funktionalitäten angegeben werden, wie Fußmattenkontakt-Erkennung, oder Glasbruch-Erkennung.

Raise Alarm: Abbildung 7.18 zeigt die Spezifikation des Dienstes Raise-Alarm. Dieser stellt die Funktionalität `raise.alarm` zur Verfügung und benötigt `raise.alarm.acoustic`, `raise.alarm.email` und `raise.alarm.visual`.

Alarm Activation Switch: Dieser Dienst überwacht den Inhalt einer Datei, deren URL im Attribut `Activation URL` angegeben ist. Abhängig vom Inhalt der Datei stellt er in einem State aktiviert oder nicht aktiviert zur Verfügung. Deshalb stellt er die Funktionalität `detect.alarm activation` zur Verfügung. Siehe Abbildung 7.20.

Acoustic Alarm: In Abbildung 7.21 ist die Spezifikation des Dienstes Acoustic-Alarm angegeben. Dieser ist für das Auslösen des akustischen Alarms verantwortlich. Als Attribut besitzt er eine URL, die auf eine Datei verweist, die im Alarmfall abgespielt werden soll. Dieses Attribut ist nicht global, so dass für jedes überwachte Umgebungselement eine andere Datei angegeben werden kann. Der Dienst stellt als Funktionalität `raise.acoustic.alarm` zur Verfügung und

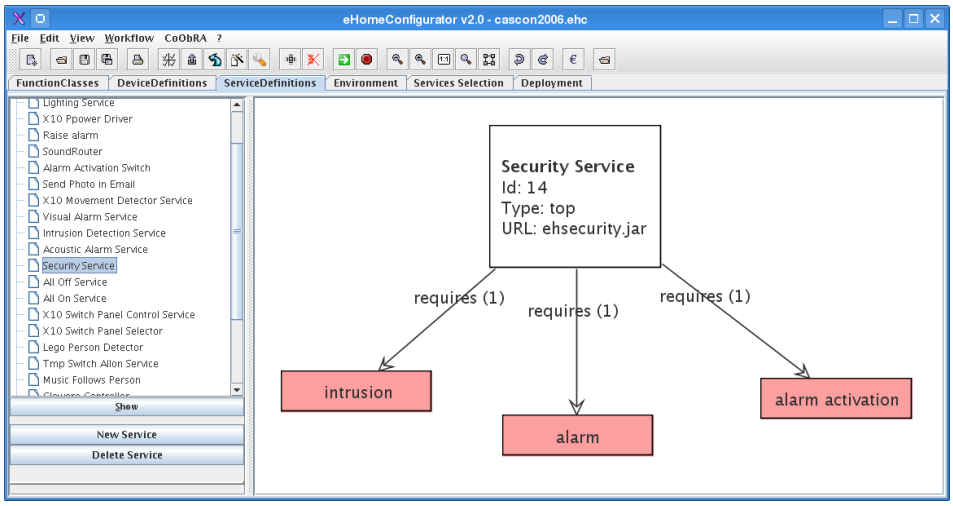

Abbildung 7.17: Spezifikation Security-Dienst

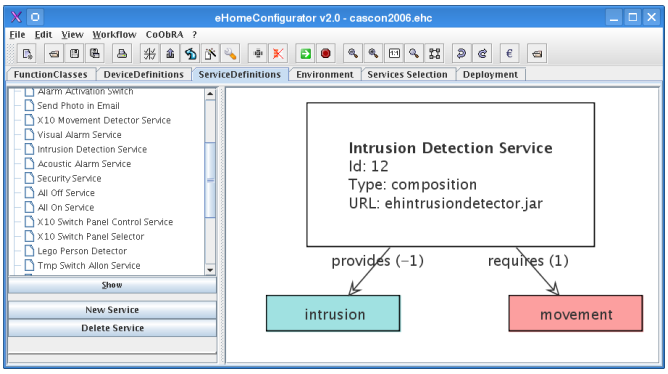

Abbildung 7.18: Spezifikation Raise-Alarm-Dienst

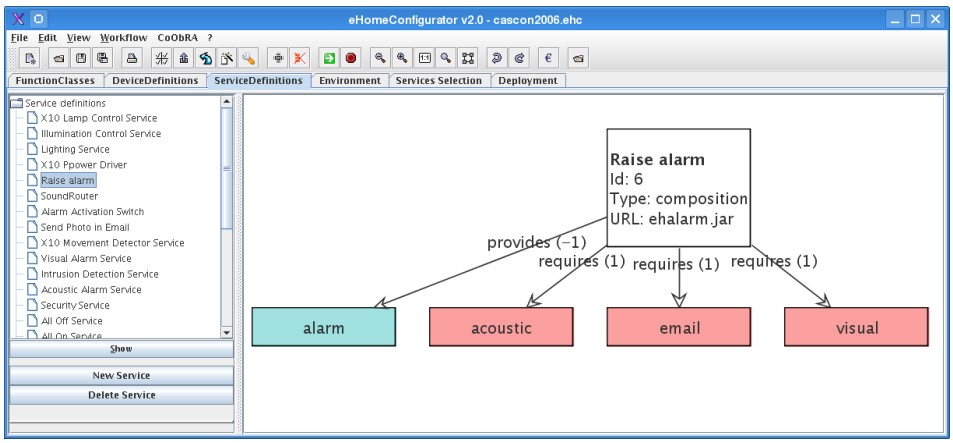

Abbildung 7.19: Spezifikation von Intrusion-Detection-Service.

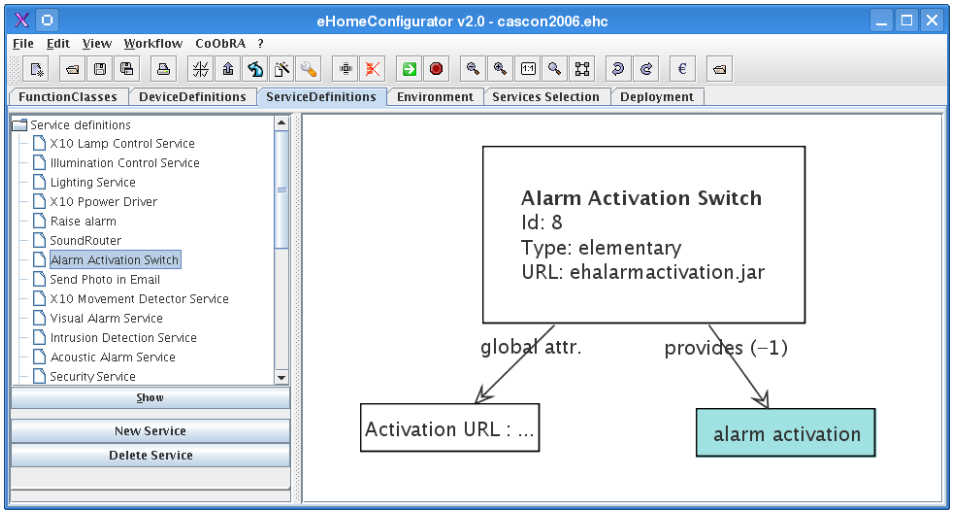

Abbildung 7.20: Spezifikation Alarm-Activation-Dienst

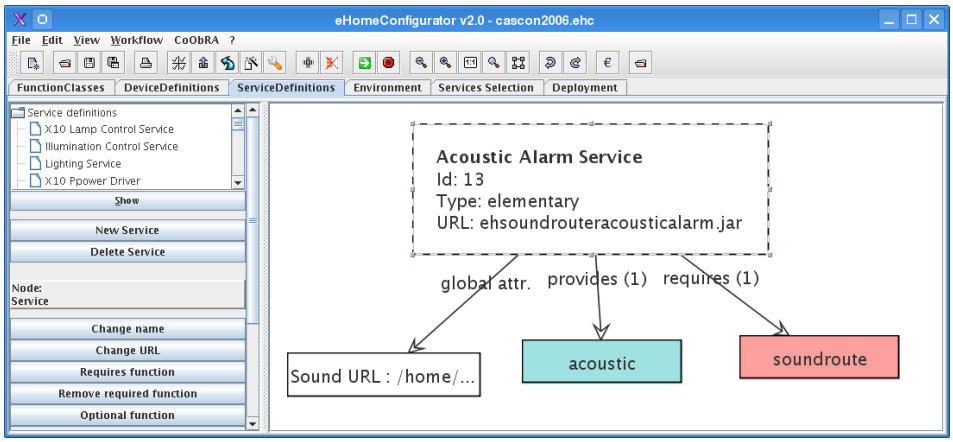

Abbildung 7.21: Spezifikation Acoustic-Alarm-Dienst

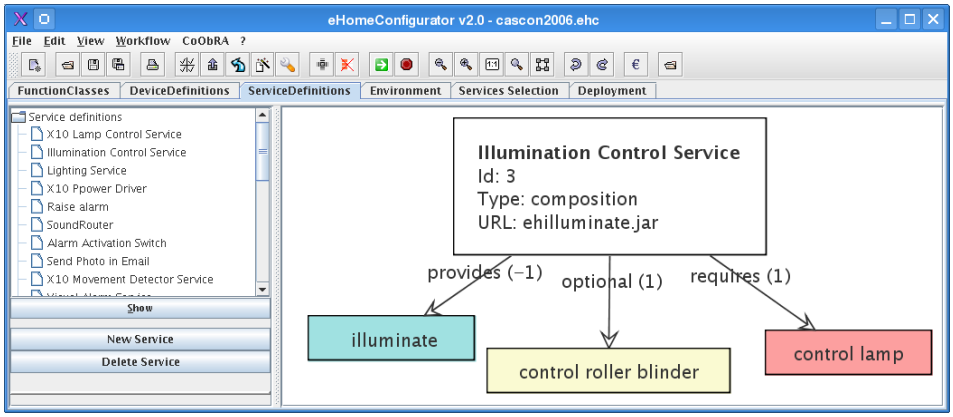

Abbildung 7.22: Spezifikation Illumination-Control-Dienst

benötigt zur akustischen Wiedergabe der angegebenen Datei die Funktionalität. `drive.soundroute`.

Illumination Control: Mittels des Dienstes Illumination-Control wird die Beleuchtung in einem konfigurierten Umgebungselement gesteuert. Wie in Abbildung 7.22 zu sehen, benötigt der Dienst die Funktionalität `control lamp` und stellt `illuminate` zur Verfügung.

Lego Lamp Control: Die Abbildung 7.23 zeigt die Spezifikation des Dienstes Lego-Lamp-Control. Dieser übernimmt die Ansteuerung einer speziellen physikalischen Legolampe. Deshalb wird in ihm auch die Kontrollbeziehung zu dem Gerät `Lego Lamp` spezifiziert. Er stellt die Funktionalität `control lamp` zur Verfügung und kann so vom Dienst Illumination-Control genutzt werden. Da der Zugriff auf die Lampe über einen speziell für den Lego-eHome-Demonstrator geschriebe-

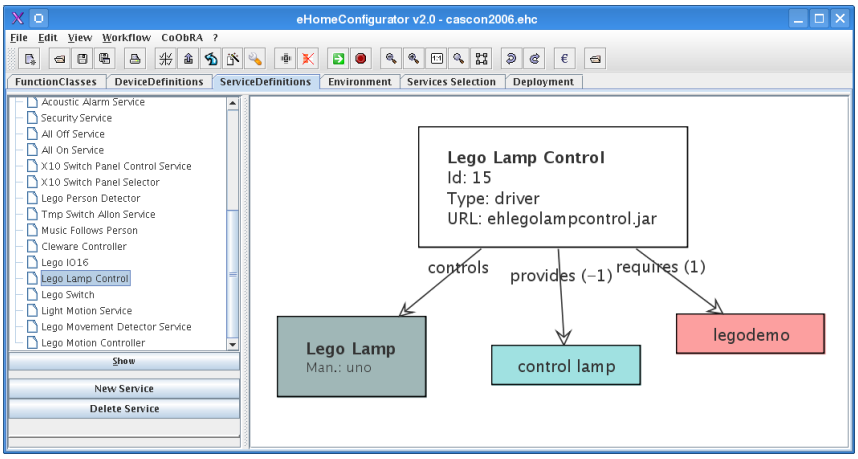

Abbildung 7.23: Spezifikation Lego-Lamp-Control-Dienst

nen Treiber realisiert wurde, benötigt der Dienst Zugriff auf diesen via der Funktionalität `drive.legodemo`.

Send Photo in Email: Die E-Mail mit einem aufgenommen Bild wird mittels dem in Abbildung 7.24 spezifizierten Dienst Send-Photo-in-Email versendet. Dieser stellt die Funktionalität raise.alarm.email zur Verfügung und hat einige Attribute:

mail text: Dies ist ein globales Attribut, welches den Text enthält, der mit jeder Mail, verschickt wird, die dieser E-Mail-Dienst erzeugt.

mail destination: Auch die Zieladresse, an wen die Mail verwendet werden soll, ist ein globales Attribut.

mail subject: Der Titel der Mail ist kein globales Attribut. So kann er später für jedes überwachte Umgebungselement einzeln angegeben werden. Somit ist es möglich für die Küche eine Mail mit dem Titel "Einbruchsalarm in der Küche" und analog für das Wohnzimmer "Einbruchsalarm im Wohnzimmer" zu generieren.

attachment url: Dies ist auch ein lokales Attribut, welches für den überwachten Raum die URL des von einer Kamera erzeugten Bildes angibt. Hier könnte auch eine Kamera als kontrolliertes Gerät angegeben werden, die dieses Attribut besitzt. Dies entspricht aber nicht der tatsächlichen Realisierung, weshalb diese Art der Beschreibung gewählt wird.

Sound Router: Der Dienst Sound-Router aus Abbildung 7.25 erlaubt die Ausgabe einer Datei auf den Computern des Heimnetzwerkes auf einer bestimmten Soundkarte. Die Attribute zur Ansteuerung dieser Soundkarte und des entsprechenden Rechners werden in der Konfiguration des Gerätes `SoundSocket` (siehe dessen Beschreibung) abgelegt. Der Dienst stellt die Funktionalität `drive.soundroute` zur Verfügung und kann so zum Beispiel von dem Dienst Acoustic-Alarm genutzt werden.

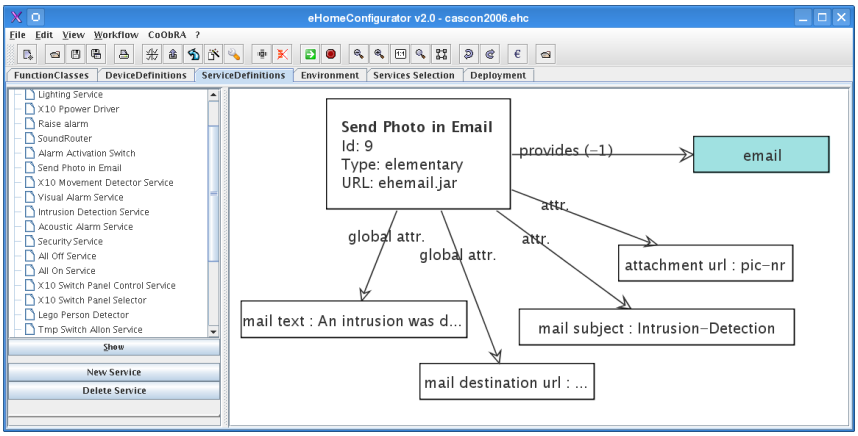

Abbildung 7.24: Spezifikation Send-Photo-in-Email-Dienst

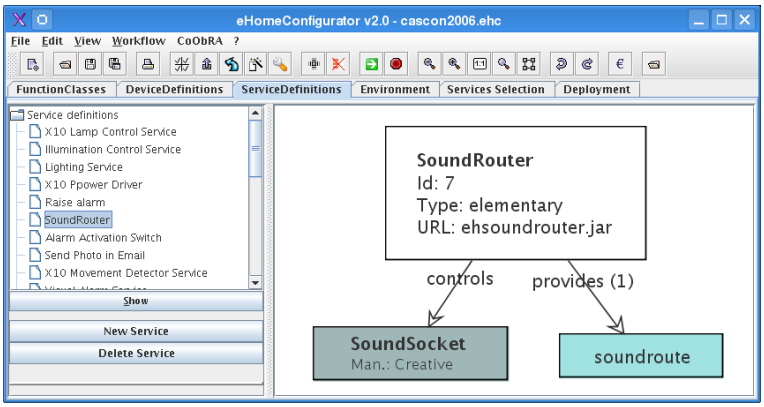

Abbildung 7.25: Spezifikation Sound Router Dienst

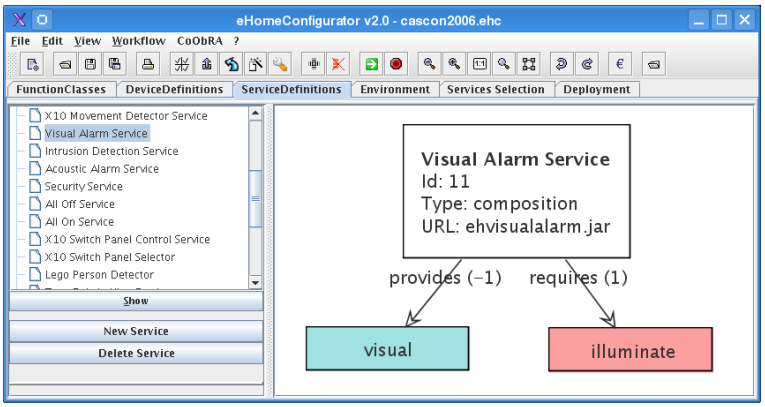

Abbildung 7.26: Spezifikation Visual-Alarm-Dienst

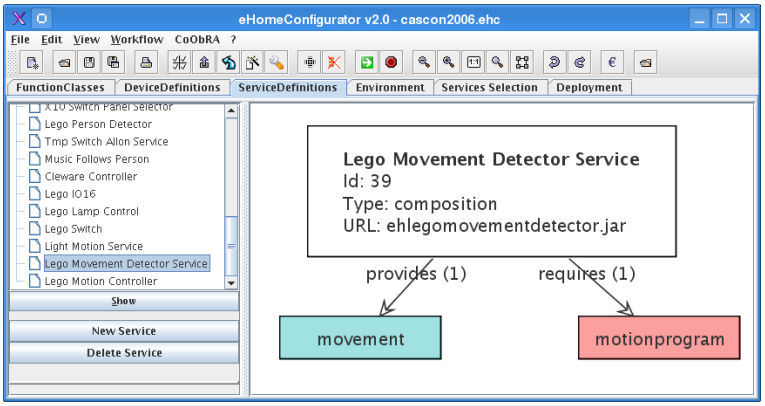

Abbildung 7.27: Spezifikation Lego-Movement-Detector-Dienst

Visual Alarm: Der in Abbildung 7.26 spezifizierte Dienst Visual-Alarm nutzt die Beleuchtungsfunktionen in einem überwachten Umgebungselement (`illuminate`), um die Lampen des entsprechenden Raumes aufblinken zu lassen. Er stellt die Funktionalität `raise.alarm.visual` zur Verfügung.

Lego Movement Detector: Der Dienst Lego-Movement-Detector nutzt ein spezielles natives Kameraüberwachungsprogramm namens Motion [Lav00], um Bewegung über die installierten Kameras des Legodemonstrators zu detektieren. Er stellt die Funktionalität `detect.motion` zur Verfügung und benötigt die Funktionalität `drive.motionprogram`, um auf die Ergebnisse des speziellen Motion-Programms zuzugreifen. Siehe für die Spezifikation Abbildung 7.27.

Lego Motion Controller: In Abbildung 7.28 ist der Dienst Lego-Motion-Controller abgebildet, der den Zugriff auf das Motion-Programm steuert. Mit ihm ist als Gerät die in dem entsprechenden Umgebungselement zu installierende und

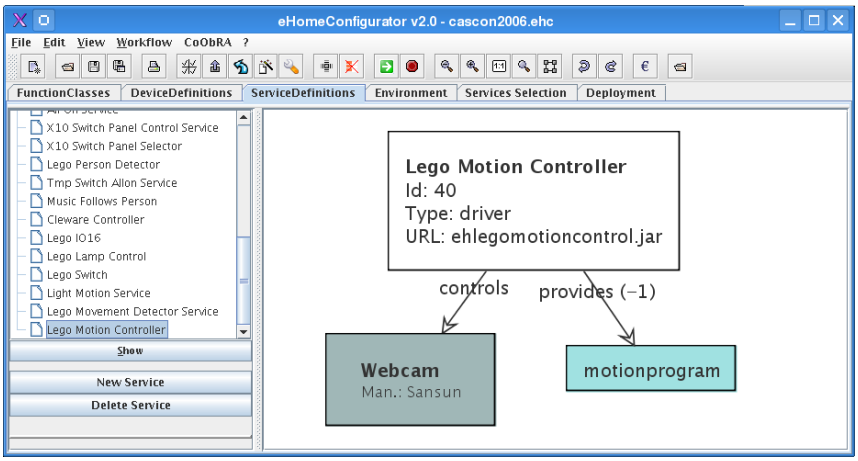

Abbildung 7.28: Spezifikation Lego-Motion-Controller-Dienst

zu überwachende Webcam (`Webcam`) verbunden. Er stellt als Funktionalität `drive.motionprogram` zur Verfügung.

7.4 Dienstentwicklung

Die Implementierung einer Softwarekomponente eines Dienstes bzw. eines OSGi-Bundles (vergleiche Abschnitt 4.3) besteht normalerweise aus vier Dateien:

1. dem Aktivator: Dieser hat immer den Namen `Activator.java` und initialisiert den Namen und einen Kommentar für das Bundle.

2. der Schnittstelle: Diese hat den Klassen bzw. Komponentennamen (zum Beispiel: `EhSecurity.java`). Hier werden die Methoden angegeben, die von dem Bundle exportiert werden.

3. der eigentlichen Implementierung: Diese trägt den Schnittstellennamen erweitert um das Kürzel `Impl` (zum Beispiel: `EhSecurityImpl.java`). Hier wird die eigentliche Funktionalität implementiert, so dass der Dienst von der Komponenten verrichtet werden kann.

4. und der Manifest-Datei: Deren Name ist der klein geschriebene Klassenname mit der Endung `.MF` (zum Beispiel: `ehsecurity.MF`). Hier werden – wie in Abschnitt 4.3 beschrieben – Im- und Exportbeziehungen, Name und Version angegeben.

7.4.1 Beispielimplementierungen

Als Beispiele werden hier die Implementierung des top-level Dienstes `Security` und des Unterdienstes `Lego-Lamp-Control` vorgestellt. Für weitere Beispiele wird hier

```
1  package ehome.ehsecurity;
2  import org.osgi.framework.BundleContext;
3  import ehome.ehdata.tools.EhActivator;
4
5  public class Activator extends EhActivator
6  {
7      public void init(BundleContext bc) {
8          this.set(new EhSecurityImpl(), "EhSecurity",
9                  "Security control service");
10     }
11 }
```

Listing 7.1: Aktivator `Activator.java` des top-level Sicherheitsdienst `Security`

auf das eHomeConfigurator-Projekt [NSM05] selbst verwiesen. Dort können die vollständigen und ungekürzten Quelltexte heruntergeladen und alle Dienstimplementationen eingesehen werden.

Security

Der top-level Dienst des Sicherheitsdienstes ist der Dienst `Security`. Er besteht aus den Dateien `Activator.java`, `EhSecurity.java`, `EhSecurityImpl.java`, `ehsecurity.MF`.

Das Listing 7.1 zeigt die Implementierung der Aktivator-Klasse. Hier werden nur die eigentliche Implementation (eine Instanz der Klasse `EhSecurityImpl`), der Name der Schnittstellenklasse und ein Kommentar übergeben. Mehr ist in einem Aktivator eines für den eHomeConfigurator erstellten Dienstes normalerweise nicht nötig. Gegebenenfalls kann der übergebene Bundle-Context in der Instanz der Implementationsklasse für einen späteren Zugriff gespeichert werden.

Wie in Listing 7.2 zu sehen, ist die Schnittstelle dieses Dienstes leer. Dies liegt daran, dass es sich hier um einen top-level Dienst (siehe Erklärung in Nomenklatur, Abschnitt 1.3) handelt, der selber Dienstschnittstellen anderer Dienste benutzt. Da die Kommunikation aber hauptsächlich über die eHome-Modell-Instanz bzw. die Konfiguration laufen sollte, sollte diese Schnittstelle für top-level Dienste und die meisten ihrer Unterdienste leer sein. Nur in Richtung der Treiberebene macht es Sinn, hier Methoden in der Schnittstelle anzubieten, da die Kommunikation der Treiberkomponenten nicht unbedingt in der Konfiguration abgebildet werden muss. Evtl. können in dieser Schnittstelle aber auch Methoden für OSGi-Komponenten angeboten werden, die nicht vom eHomeConfigurator und dessen Konfigurierungsmechanismen erfasst werden. So können die Dienste des eHomeConfigurators so programmiert werden, dass sie auch außerhalb des eHomeConfigurator-Kontextes angewandt werden können. Weiterhin stellt das Ableiten von der Klasse `EhService` sicher, dass in der implementierenden Klasse, die Methoden `init` und `execute` realisiert werden, da diese vom Deployment-Werkzeug (siehe Abschnitt 6.7) des eHomeConfigurators genutzt werden.

Listing 7.3 und 7.4 enthalten die eigentliche Implementierung des Dienstes, also die Klasse `EhSecurityImpl`. Sie implementiert die Methoden `execute` und `init`. Die `init`-Methode ist leer, da nur Unterdienste, die selber wieder Erweiterungs-Dienste, aber

```
1 package ehome.ehsecurity;
2
3 import ehome.ehdata.model.EhService;
4
5 public interface EhSecurity extends EhService
6 {
7 }
```

Listing 7.2: Die Schnittstelle `Security.java` des top-level Sicherheitsdienst `Security`

keine besonderen Treiber sind, von diesem top-level Dienst benutzt werden. Deren Kommunikationsverbindungen werden in der `execute`-Methode initialisiert. Die `execute`-Methode besteht in der Regel aus drei Schritten:

1. Der erste Schritt in der execute-Methode ist das Ausführen der Unterdienste mittels `so.executeDependingServices()` (siehe Listing 7.4, Zeile 15). Dies ist im Normalfall immer der erste Schritt, den ein Dienst ausführen sollte, wenn seine Unterdienste Erweiterungsdienste sind.

2. Im zweiten Schritt werden die Objekte gesucht, mit denen mit den Instanzen der Unterdiensten im gleichen Umgebungselement kommuniziert werden kann (siehe Listing 7.4, Zeile 16-50). Dies geschieht über eine Iteration über die benutzten Dienstobjekte (UsedServiceObjects, Zeile 17) und die Auswertung ihrer States, deren Identifier in 7.3, Zeile 14-16, definiert werden.

3. Im dritten Schritt werden `PropertyChangeListener` eingerichtet, die auf Zustandsänderungen der gerade ermittelten `State`-Objekte reagieren. Hier wird das `intrusionstate`-Objekt auf Veränderung überwacht. Das `intrusionstate`-Objekt zeigt an, ob eine unberechtigtes Eindringen erkannt wurde.

Der `PropertyChangeListener` wird in 7.3, Zeile 18-52, definiert. Er besteht aus einer Initialisierung, die die beteiligten Unterdienste abspeichert (Zeile 28-35), und der Methode, die bei einer detektierten Zustandsänderung aufgerufen wird (Zeile 38-50). Diese Zustandsänderungsmethode ermittelt den Zustand des Alarmaktivierungsdienstes, also ob die Alarmanlage überhaupt aktiviert ist, und löst im Falle der Aktivierung einen Alarm durch die Benachrichtigung des Alarm-Unterdienstes über Veränderung dessen `State`-Objekts aus.

Listing 7.5 zeigt die Manifest-Datei dieses Bundles. Dort werden, wie in Abschnitt 4.3 bereits erklärt, die Import- und Export-Beziehungen auf OSGi-Bundle-Ebene definiert. Die Beziehung zu anderen eHome-Diensten des eHomeConfigurators müssen nicht angegeben werden, da diese vom Konfigurierungswerkzeug verwaltet werden und die Kommunikation mit diesen über die eHome-Modell-Instanz in der gerade am Beispiel der `PropertyChangeListener`-Implementierung gezeigten Weise abläuft und somit die OSGi-Strukturen nicht benötigt. Bei anderen Diensten sind die Eintragung zu Import-Package in der dessen Manifest-Datei somit in der Regel identisch zu den hier vorgestell-

```
1  package ehome.ehsecurity;
2
3
4  import java.beans.PropertyChangeEvent;
5  import java.beans.PropertyChangeListener;
6  import java.util.Iterator;
7
8  import ehome.ehdata.model.ServiceObject;
9  import ehome.ehdata.model.State;
10
11
12 public class EhSecurityImpl implements EhSecurity {
13
14     static final String StateNameAlarm = "Alarm";
15     static final String StateNameActivated = "Activated";
16     static final String StateNameIntrusion = "Intrusion";
17
18     // Class to handle object-changes in the intrusion-detectors
19     private class IntrusionListener
20         implements PropertyChangeListener
21     {
22
23         ServiceObject intrusionservice,
24                       notifyservice,
25                       activationservice;
26
27
28         IntrusionListener( ServiceObject intrusionservice,
29                            ServiceObject notifyservice,
30                            ServiceObject activationservice )
31         {
32             this.intrusionservice = intrusionservice;
33             this.notifyservice = notifyservice;
34             this.activationservice = activationservice;
35         }
36
37
38         public void propertyChange(PropertyChangeEvent evt)
39         {
40             // First check, if alarm is activated
41             State actstate = activationservice.getState(
42                                         StateNameActivated );
43             if( actstate.getValue().equals("1") )
44             {   // Just toggle Alarm-Service to notify
45                 State notifystate = notifyservice.getState(
46                                                 StateNameAlarm );
47                 notifystate.setValue(
48                     notifystate.getValue().equals("1")? "0":"1");
49             }
50         }
51
52     }
53
54     ...
```

Listing 7.3: Implementierung `EhSecurityImpl.java` des top-level Sicherheitsdienst `Security` (erster Teil)

```
1       ...
2
3       /* find illumination and switch service and propagate values
4        * respectivly */
5       public boolean execute( ServiceObject so )
6       {
7           ServiceObject svc;
8           ServiceObject intrusionservice=null,
9                         notifyservice=null,
10                        activationservice=null;
11          State activationstate = null,
12                notifystate = null,
13                intrusionstate = null;
14
15          so.executeDependingServices(); // run all sub-services
16          // find services
17          Iterator usedservices = so.iteratorOfUsedServiceObjects();
18          while( usedservices.hasNext() )
19          {
20              ServiceObject usedso =
21                  (ServiceObject) usedservices.next();
22              State tmpdetectstate = usedso.getState(
23                                          StateNameActivated );
24              if( tmpdetectstate != null )
25              {
26                  activationservice = usedso;
27                  activationstate = tmpdetectstate;
28              }
29              else
30              {
31                  tmpdetectstate = usedso.getState(StateNameAlarm);
32                  if( tmpdetectstate != null )
33                  {
34                      notifyservice = usedso;
35                      notifystate = tmpdetectstate;
36                  }
37                  else
38                  {
39                      tmpdetectstate = usedso.getState(
40                          StateNameIntrusion);
41                      if( tmpdetectstate != null )
42                      {
43                          intrusionservice = usedso;
44                          intrusionstate = tmpdetectstate;
45                      }
46                  }
47              }
48          } // End: find services
49          if( activationservice == null || notifyservice == null ||
50                intrusionservice == null ) return false;
51          // Install Listeners
52          intrusionstate.addPropertyChangeListener(
53                  new IntrusionListener(intrusionservice,
54                          notifyservice, activationservice) );
55          return true;
56      }
57
58      public boolean init() { // Nothing needed for initialization
59          return true; }
60  }
```

Listing 7.4: Implementierung `EhSecurityImpl.java` des top-level Sicherheitsdienst `Security` (zweiter Teil)

```
1  Manifest-Version: Version 1.0
2  Bundle-ContactAddress: phd.ehomeconfig@mail.ulno.net
3  Bundle-Version: 0.0.1
4  Bundle-Activator: ehome.ehsecurity.Activator
5  Bundle-Name: EhLighting
6  Bundle-Vendor: Ulrich Norbisrath
7  Export-Package: ehome.ehsecurity
8  Export-Service: EhSecurity
9  Bundle-Description: Provides security control
10 Bundle-Category: rwth-i3
11 Import-Package: ehome.ehdata,
12  ehome.ehdata.model,
13  ehome.ehdata.model.exception,
14  ehome.ehdata.tools
```

Listing 7.5: `ehsecurity.MF` des top-level Sicherheitsdienst `Security`

```
1  package ehome.ehlegolampcontrol;
2
3  import org.osgi.framework.BundleContext;
4  import ehome.ehdata.tools.EhActivator;
5
6  public class Activator extends EhActivator
7  {
8      public void init(BundleContext bc) {
9          this.set(new EhLegoLampControlImpl(bc), "EhLegoLampControl",
10                 "Controls lamps of the Lego Demonstrator");
11     }
12 }
```

Listing 7.6: `Activator.java` des Dienstes `Lego-Lamp-Control`

ten Beispiel. Die Eintragungen bei Export-Package und Export-Service variieren entsprechend des Implementationsnamens.

Lego-Lamp-Control

Einer der Unterdienste des Sicherheitsdienstes ist der Dienst `Lego-Lamp-Control`. Er besteht aus den Dateien `Activator.java`, `EhLegoLampControl.java`, `EhLegoLampControlImpl.java`, `ehlegolampcontrol.MF`.

Das Listing 7.6 zeigt die Implementierung der Aktivator-Klasse. Anders als der Aktivator des `Security`-Dienstes (Listing 7.1), wird in diesem Aktivator der BundleContext an die Implementierungsinstanz übergeben. Ansonsten ist der Aufbau des Aktivators völlig analog zum Security-Dienst.

In Listing 7.7 ist die Schnittstelle zu sehen. Sie enthält auch eine Methode (`setIllumination`), die andere Dienste außerhalb des eHomeConfigurators benutzen können, um eine Lego-Lampe anzusteuern.

Listing 7.8 enthält die eigentliche Implementierung des Dienstes. Da dieser Dienst den `ClewareControl`-Treiber-Dienst[2] benutzt, enthält die `init`-Methode (siehe Lis-

[2]Cleware ist der Hersteller der IO16-Hardware-Schnittstelle [Cle05], mit der die Lego-Geräte angesteuert werden.

```
1  package ehome.ehlegolampcontrol;
2
3  import ehome.ehdata.model.EhService;
4  import ehome.ehdata.model.ServiceObject;
5
6  public interface EhLegoLampControl extends EhService
7  {
8      void setIllumination(ServiceObject so,
9                  double illuminationvalue);
10 }
```

Listing 7.7: Die Schnittstelle `EhLegoLampControl.java` des Dienstes `Lego-Lamp-Control`

ting 7.8, Zeile 58-63) eine auf der OSGi-API basierende Methode, um dieses Treiber-Bundle zu finden. Die `init`-Methode speichert die Referenz auf die Schnittstelle zum ClewareControl-Dienst in einer in der Klasse verfügbaren Referenzvariable (`clewarecontrol`, Definition in Zeile 6).

Die `execute`-Methode (Zeile 24-37) ruft diesmal keine execute-Methode der Unterdienste auf, da die Unterdienste nur allgemeine Treiberdienste sind, deren Laufzeitinformationen nicht an einzelne Dienstobjekte gebunden sind und somit nur die allgemeine Initialisierung über die `init`-Methode benötigen. Anschließend wird überprüft, ob innerhalb der Konfiguration bereits ein `State`-Objekt für den Zustand der gesteuerten Lampen verfügbar ist. Wenn ja, wird der Zustand dieses Objektes wiederhergestellt. Ist die Beleuchtung also für das aktuelle Umgebungselement eingeschaltet, so werden alle Lego-Lampen, des aktuellen Umgebungselements eingeschaltet. Analog verhält es sich, wenn der Zustand anzeigt, dass die Beleuchtung ausgeschaltet ist. Ist das `State`-Objekt nicht vorhanden, so wird es mit dem Zustand ausgeschaltet angelegt. Anschließend wird auf das eigene `State`-Objekt ein `PropertyChangeListener` installiert, der dessen Veränderungen überwacht.

Die Methode `setIllumination` (Zeile 42-56) regelt die Kommunikation mit dem `ClewareControl`-Treiber und schaltet alle zum Umgebungselement eines Dienstobjektes gehörenden Lego-Lampen.

Der `LegoLampControlListener` (Zeile 11-22) überwacht Änderungen auf dem zum aktuellen Dienstobjekt gehörenden `State`-Objekt. Wird der Zustand des State-Objektes von außen geändert, so wird diese Änderung an die `setIllumnation`-Methode propagiert und somit in Hardware sichtbar.

Listing 7.9 zeigt die Manifest-Datei dieses Bundles. Der Aufbau ist analog zu der Manifest-Datei aus dem Security-Dienst-Beispiel. Allerdings wird hier zusätzlich der OSGi-Treiber-Dienst für den ClewareControl-Treiber in Zeile 15 importiert, damit er in der `init`-Methode gefunden werden kann.

7.4.2 Neuer Dienst

Diese Beschreibung geht davon aus, dass für die Implementierung eines Dienstes das Entwicklungswerkzeug Eclipse [Ecl03] verwendet wird. Um einen neuen Dienst für die Werkzeugsuite zu erzeugen, sollten zuerst die Quelltexte des eHomeConfigurators her-

```
1  package ehome.ehlegolampcontrol;
2  ...imports...
3
4  public class EhLegoLampControlImpl implements EhLegoLampControl {
5      BundleContext bc;
6      ClewareControl clewarecontrol; // Connection to other Component
7
8      static final String StateNameIllumination="Illumination";
9      static final String AttributeNameLampNumber="IO16-Port Number";
10
11     private class LegoLampControlListener implements
12         PropertyChangeListener { // Class to handle object-changes
13         ServiceObject so;
14
15         LegoLampControlListener( ServiceObject so )
16         { this.so = so; }
17
18         /* propagate the change to all subservices */
19         public void propertyChange(PropertyChangeEvent evt) {
20             setIllumination( so, Double.valueOf(
21                 so.getState(StateNameIllumination).getValue())
22                                     .doubleValue() ); } }
23
24     /* get all depending services of the location and ensure their
25      * states react on state-changes */
26     public boolean execute( ServiceObject so ) {
27         State illuminationstate=so.getState(StateNameIllumination);
28         if (illuminationstate == null)
29         { // if not existent, create state with illumination off
30             illuminationstate = so.createState(
31                 StateNameIllumination, "0" ); }
32         setIllumination( so, Double.valueOf( // Initialize
33                 illuminationstate.getValue()).doubleValue() );
34         LegoLampControlListener lcl = // Install Change-Listener
35                                 new LegoLampControlListener( so );
36         illuminationstate.addPropertyChangeListener( lcl );
37         return true; }
38
39     /* set the Illumination for all X10 Lamps contld. by a srv.
40      * this value should be a number between 0 or 1 corr. on or off
41      * intermediate steps are for dimmable lamps. */
42     public void setIllumination(ServiceObject so,
43                 double illuminationvalue) {
44         // iterate over all devices
45         Iterator deviceiterator = so.iteratorOfControlledDevices();
46         while( deviceiterator.hasNext() ) {
47             Device dev = (Device) deviceiterator.next();
48             String lampnrstr = dev.getAttribute(
49                     AttributeNameLampNumber ).getValue();
50             int lampnr = Integer.parseInt(lampnrstr) - 1;
51             if( lampnr < 0 && lampnr >4 )
52             { return; } // Wrong value for lamp
53             if( illuminationvalue < 0.5 )
54             { clewarecontrol.setSwitchOn(lampnr,false); }
55             else
56             { clewarecontrol.setSwitchOn(lampnr,true); } } }
57
58     public boolean init() {
59         // Try to contact ClewareControl-Bundle
60         clewarecontrol = (ClewareControl) bc.getService(
61             bc.getServiceReference(
62                 "ehome.clewarecontrol.ClewareControl"));
63         return (clewarecontrol != null); }
64
65     EhLegoLampControlImpl( BundleContext bc) { this.bc = bc; }
66 }
```

Listing 7.8: Implementierung `EhLegoLampControlImpl.java` des Dienstes `Lego-Lamp-Control`

```
1  Manifest-Version: Version 1.0
2  Bundle-ContactAddress: phd.ehomeconfig@mail.ulno.net
3  Bundle-Version: 0.0.1
4  Bundle-Activator: ehome.ehlegolampcontrol.Activator
5  Bundle-Name: EhLegoLampControl
6  Bundle-Vendor: Ulrich Norbisrath
7  Export-Package: ehome.ehlegolampcontrol
8  Export-Service: EhLegoLampControl
9  Bundle-Description: Controls lamps of the Lego Demonstrator
10 Bundle-Category: rwth-i3
11 Import-Package: ehome.ehdata,
12  ehome.ehdata.model,
13  ehome.ehdata.model.exception,
14  ehome.ehdata.tools,
15  ehome.clewarecontrol
```

Listing 7.9: `ehlegolampcontrol.MF` des Dienstes `Lego-Lamp-Control`

untergeladen werden. Die Anleitung zum Herunterladen ist unter der Webseite [NSM05] zu finden. Anschließend muss im Verzeichnis `src/java/ehome` ein neues Verzeichnis für den neuen Dienst angelegt werden, zum Beispiel `ehNeuerDienst`.

In diesem Verzeichnis müssen die vier Dateien, die am Anfang dieses Kapitels und anschließend in zwei Beispielen besprochen wurden, erstellt werden, d.h.: der Aktivator, die Schnittstelle, die Manifest-Datei die eigentliche Implementation. Für den Aufbau der ersten drei Dateien können die gerade besprochenen Beispiele verwendet werden. Dort müssen wahrscheinlich nur die Namen angepasst werden. Wenn ein neuer Treiber programmiert werden soll, so müssen die Importbeziehungen in der Manifest-Datei berücksichtigt werden.

Für die eigentliche Implementation ergibt sich, wie sich schon aus den Beispielen erahnen lässt, folgender Aufbau aus drei Teilen:

1. Im ersten Teil sollten die Unterdienste des aktuellen Dienstobjektes mittels `serverobjekt.executeDependingServices()` ausgeführt werden. Dies ist nötig, falls nur Treiber als Unterdienste oder gar keine Unterdienste verwendet werden.

2. Anschließend sollten die Dienstobjekte der benutzten Unterdienste und gegebenenfalls zu überwachende `State`-Objekte bestimmt werden. Wenn der Dienst eigene `State`-Objekte zur Verfügung stellt, sollte hier weiterhin überprüft werden, ob diese bereits in der Konfiguration vorhanden sind und dann gegebenenfalls ihren physikalischen Zustand wieder herzustellen. Ansonsten sind diese hier zu erzeugen und mit sinnvollen Default-Werten zu belegen. Diese finden sich evtl. in Attributen. Wenn dies der Fall ist, müssen diese gesucht werden. Dies ist über Assoziationen zu anderen Objekten möglich, die dem aktuellen Dienstobjekt zugeordnet sind. Siehe dafür den Aufbau des eHome-Modells in Abschnitt 5.2 und insbesondere den Dienstobjekt-Kontext in Abschnitt 5.2.5.

3. Anschließend sollten die `PropertyChangeListener` eingerichtet werden, die eigene bzw. Zustandsänderungen von Unterdiensten über-

wachen und selbstverständlich deren Reaktionen in den entsprechenden `PropertyChangeListener`-Klassen implementiert werden.

Dieses Vorgehen ist nicht bindend. Es kann auch komplett die klassische Methodik aus dem OSGi-Rahmenwerk verwendet werden. Sollte dies aber auf Erweiterungs-Dienstebene geschehen, kann damit die Erweiterbarkeit des eHomeConfigurator-Dienste vermindert werden. Trotzdem war bei der Entwicklung des eHomeConfigurator-Werkzeugsuite wichtig, dem Dienstentwickler weiterhin so viele Freiheiten bei der Dienstentwicklung wie möglich zu lassen. Generell sollte aber gelten: Wenn Kommunikation über Zustandsänderung abgebildet werden kann, sollte dies getan werden.

Kompilieren die Dateien fehlerfrei, so kann mittels des Java-Projektübersetzungs-Werkzeugs *Ant* [Fou04] und der Projektdatei `build.xml` im Wurzelverzeichnis des Projektes über die Auswahl des Ant-Targets `bundles` ein neues Bundle für diesen Dienst erstellt werden.

Spezifiziert man diesen Dienst und seine benutzten Geräte nun analog zu den Abschnitten 7.1 und 7.3, so ist der neue Dienst einsatzbereit und kann konfiguriert und anschließend deployed werden.

7.5 Konfigurierung

Nachdem die eHomeConfigurator-Werkzeugsuite im Haus des Kunden gestartet wurde, muss die Umgebung des Kunden spezifiziert werden. Es wird also der Teil des Konfigurationsgraphen aufgebaut, der dem Umgebungskontext genügt. Im Falle des Lego-eHome-Demonstrators (siehe Abschnitt 2.2.2) müssen in der Umgebungsspezifikation die Räume Küche (kitchen), Wohnzimmer (living room), Schlafzimmer (bedroom), Diele (Hall), Badezimmer (bathroom) und ihre Verbindungen, also die Türen (Doors), spezifiziert werden. Das Ergebnis dieser Spezifikation ist in Abbildung 7.29 zu sehen. Hier sind keine vorinstallierten Geräte spezifiziert, d.h. in der Praxis, dass keine elektronisch steuerbaren Geräte in der Kundenumgebung vorhanden sind, die von dem jetzt zu erstellenden eHome-System genutzt werden könnten. Wenn keine Geräte – wie im hier besprochenen Beispiel – vorhanden sind, bedeutet das, dass alle benötigten Geräte, die zur Erfüllung der ausgewählten Dienste benötigt werden, von der eHomeConfigurator-Werkzeugsuite vorgeschlagen und ergänzt werden.

Im nächsten Schritt muss der Kunde den Konfigurierungsassistenten (vergleiche Abschnitt 6.6) über den korrespondierenden Knopf in der Werkzeugleiste starten (siehe Abbildung 7.30). Dieser präsentiert einen Dialog mit den vorhandenen top-level Diensten und den in Abschnitt 6.6 besprochenen Konfigurierungsstrategien. Der Kunde muss nun wählen, welche Dienste in seiner Wohnung installiert bzw. angeboten werden sollen. Der Auswahldialog ist in Abbildung 7.31 abgebildet.

Mittels den Auswahlmöglichkeiten hinter der Dienstauswahl "Select All Locations" oder "Select Possible Locations" kann bestimmt werden, ob für die Lokationsauswahl im nächsten Schritt die bereits in der Umgebungsspezifikation angegebenen Geräte berücksichtigt werden sollen. Da in dem hier besprochenen Beispiel keine angegebenen sind, wird die Auswahl hier auf "Select All Locations" belassen. Die Auswahl zwischen "All Devices" oder "Necessary Devices Only" legt fest, ob die optionalen Unterdienste mit einbezogen werden sollen oder nicht.

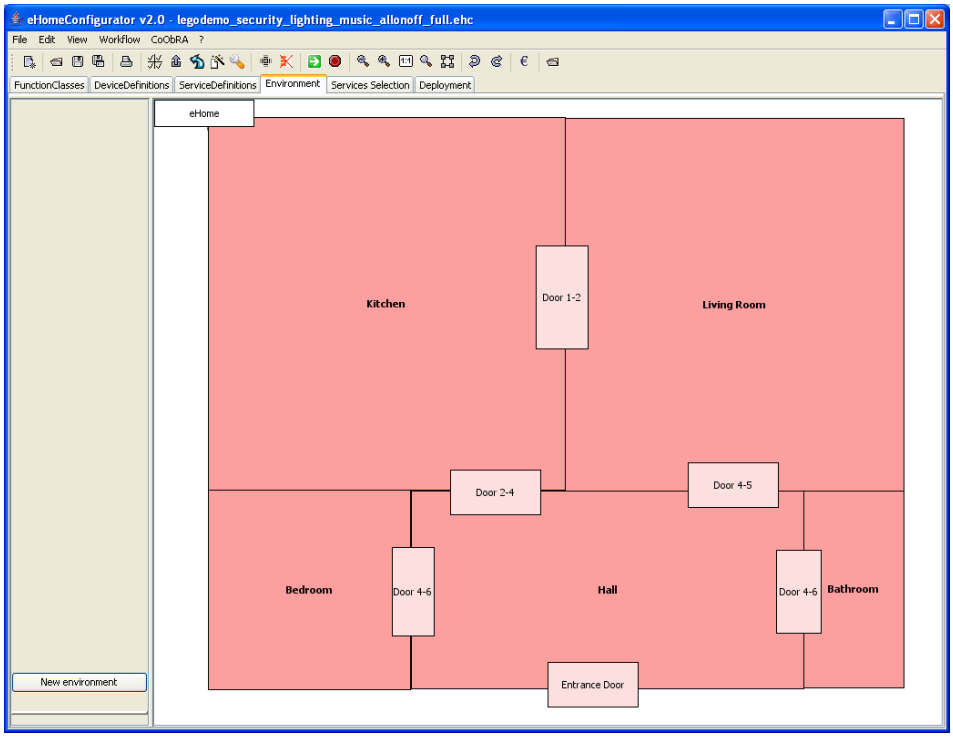

Abbildung 7.29: eHomeConfigurator: leere Umgebung

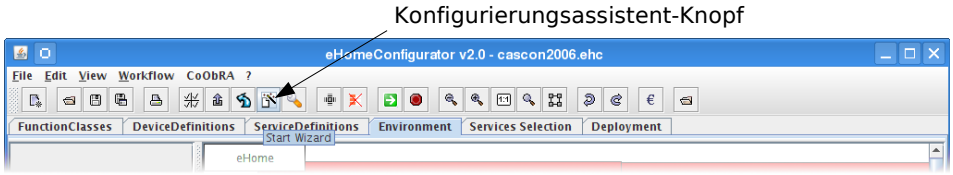

Abbildung 7.30: Der Knopf der Werkzeugleiste, um den Konfigurierungsassistenten zu starten

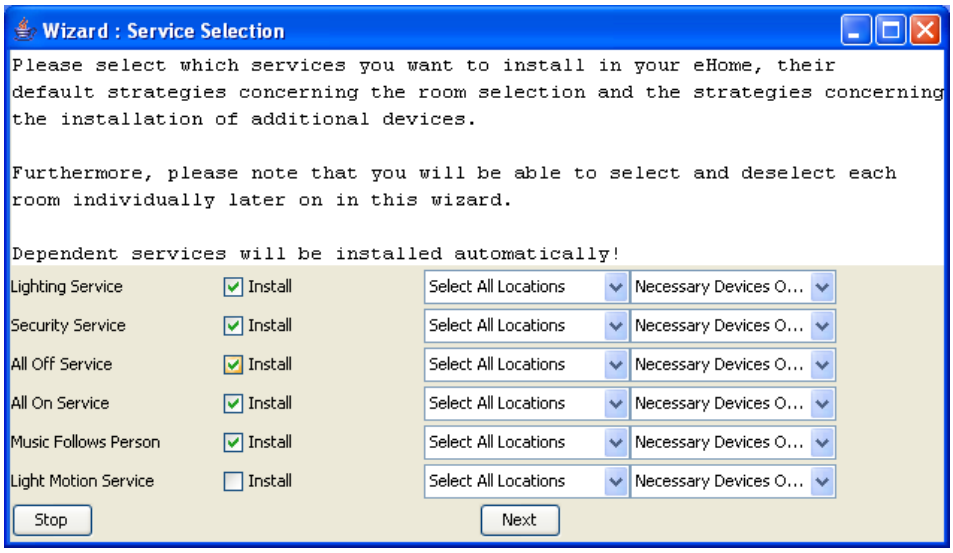

Abbildung 7.31: eHomeConfigurator: top-level Dienstauswahl

Im hier besprochenen Beispiel wählt der Bewohner die fünf ersten Dienste aus der ihm angebotenen Liste aus. Dies sind die Dienste `Lighting`, `Security`, `All Off`, `All On` und `Music Follows Person`. Für deren Erläuterung siehe Abschnitt 2.2.1 auf Seite 27. Weiterhin wählt der Bewohner "Select All Locations" und lässt die Dienste mit der minimalen Menge benötigter Geräte konfigurieren, d.h. der Kunde wählt "Necessary Devices Only".[3]

Im nächsten Schritt kann der Bewohner die Raum- oder besser Lokation-Auswahl genauer einstellen: Einige Dienste wie `AllOnService`, welcher alle Lampen der konfigurierten Umgebung mit einem Knopf anschaltet, können nur von ausgezeichneten Lokationen aus, wie dem Schlafzimmer oder der Diele, bedient werden. Sie müssen nicht von jedem Raum aus gesteuert werden können (siehe Abbildung 7.32).

Nachdem diese Information zur Verfügung gestellt wurden, kann das Konfigurierungswerkzeug der eHomeConfigurator-Werkzeugsuite berechnen, welche Software-Realisierungen zu den gewählten top-level Diensten für die gegebenen Parameter und die gegebene Umgebung existieren. Im hier gezeigten Beispiel wird dem Bewohner eine Realisierung auf Basis von Legotreiberkomponenten, welche Legogeräte benötigen, und anderen Treibern geboten, welche dann zum Beispiel X10-Geräte erfordern würden. Für das abgebildete Beispiel wäre auch eine X10-Realisierung in einem oder mehreren Räumen möglich. Das Konfigurierungswerkzeug erkennt, dass es die Beleuchtungsfunktionalität sowohl mit X10-Gerätetreibern als auch mit den Legotreibern realisieren kann. Diese Auswahl der Realisierungsmöglichkeiten wird in Abbildung 7.33 gezeigt.

Nachdem diese Auswahl durchgeführt wurde, kann das Konfigurierungswerkzeug die Berechnung der notwendigen Unterdienste und zugehörigen Softwarekomponenten, deren Abhängigkeiten und Geräte abschließen. Zur Vervollständigung der Konfiguration

[3] Dies sind die Default-Einstellungen, sie müssen also nicht explizit ausgewählt werden.

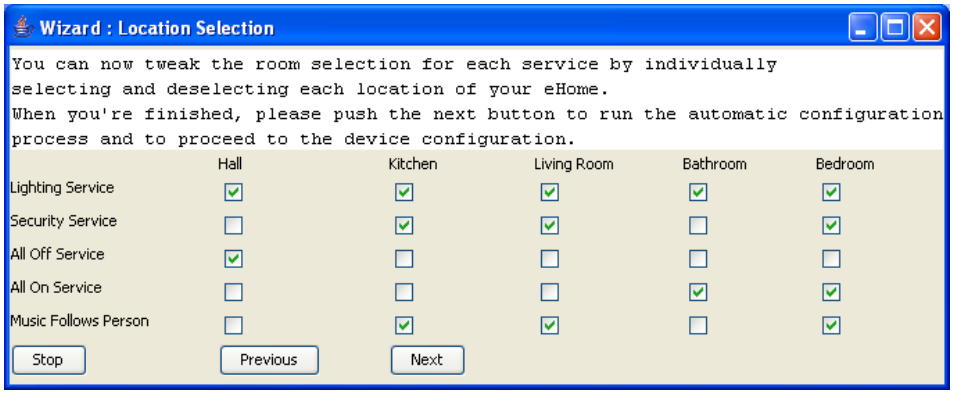

Abbildung 7.32: eHomeConfigurator: Lokationsauswahl

Abbildung 7.33: eHomeConfigurator: Realisierungsalternativenauswahl

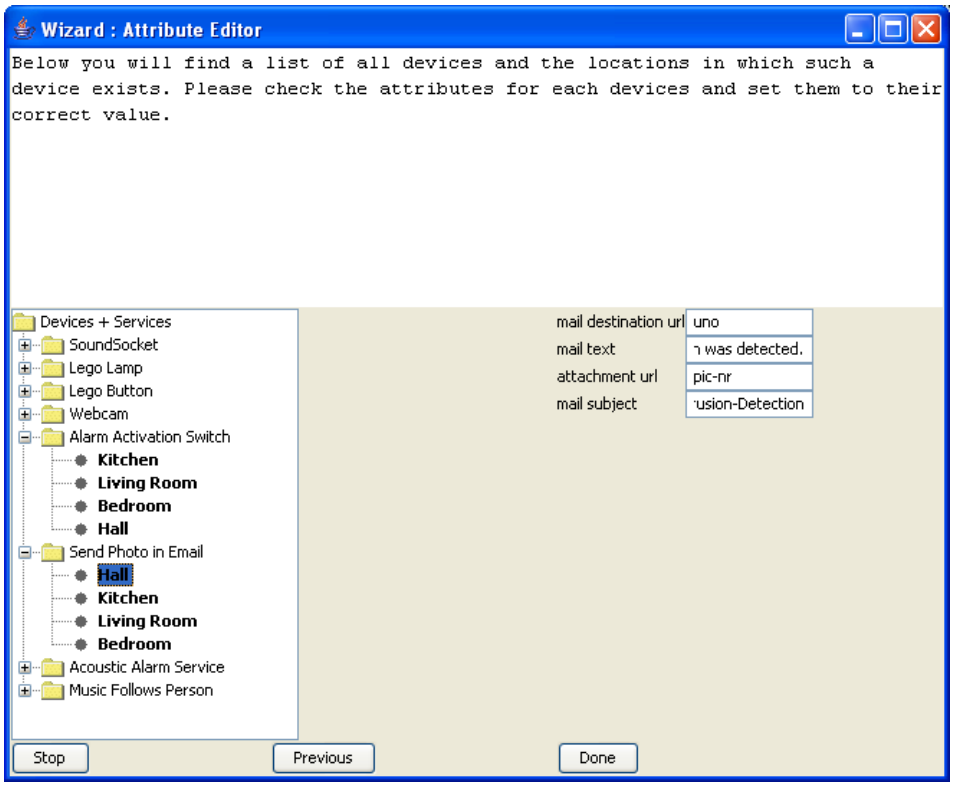

Abbildung 7.34: eHomeConfigurator: Attributeditor

fehlt jetzt nur noch die Angabe der Parameter der Geräte und besonderer Treiber. Diese können mittels dem in Abbildung 7.34 gezeigten Attributeditor eingegeben werden.

Anschließend kann der Bewohner die benötigten Geräte kaufen oder über den Anbieter beziehen. Nach deren Installation können entsprechende Adressparameter korrespondierend der Lokationen, in denen die Geräte installiert wurden, in den Attributeditor (Abbildung 7.34) eingegeben werden. Das können die Adressen der IO-Leitung zur Steuerung der Lego-Lampen sein (ganze Zahlen von 1 bis 6, zum Beispiel 2 für die Küche) oder E-Mailadressen (zum Beispiel "phd.ehomeconfig@mail.ulno.net" für die Benachrichtigung von Eindringlingsalarmen) aber auch Nachrichtentexte (zum Beispiel "intrusion detection in the kitchen") im Sicherheitsdienst. In Abbildung 7.34 wird ein Teil der Parametereingabe des Benachrichtigungsunterdienstes des Sicherheitsdienstes abgebildet.

Wie schon angedeutet, erhält der Kunde als erstes Ergebnis der Konfigurierungsphase eine Liste aller noch benötigter Geräte. Das heißt, alle diese Geräte wurden vom Konfigurierungswerkzeug der Konfiguration hinzugefügt und mit den steuernden Dienstobjekten und Lokationen assoziiert, also der Umgebung hinzugefügt. Die leere Umgebung aus Abbildung 7.29 sieht nach der Konfigurierung so aus, wie in Abbildung 7.35 abgebildet. Die Geräte müssen nun physikalisch an den angegebene Stellen, also in den entspre-

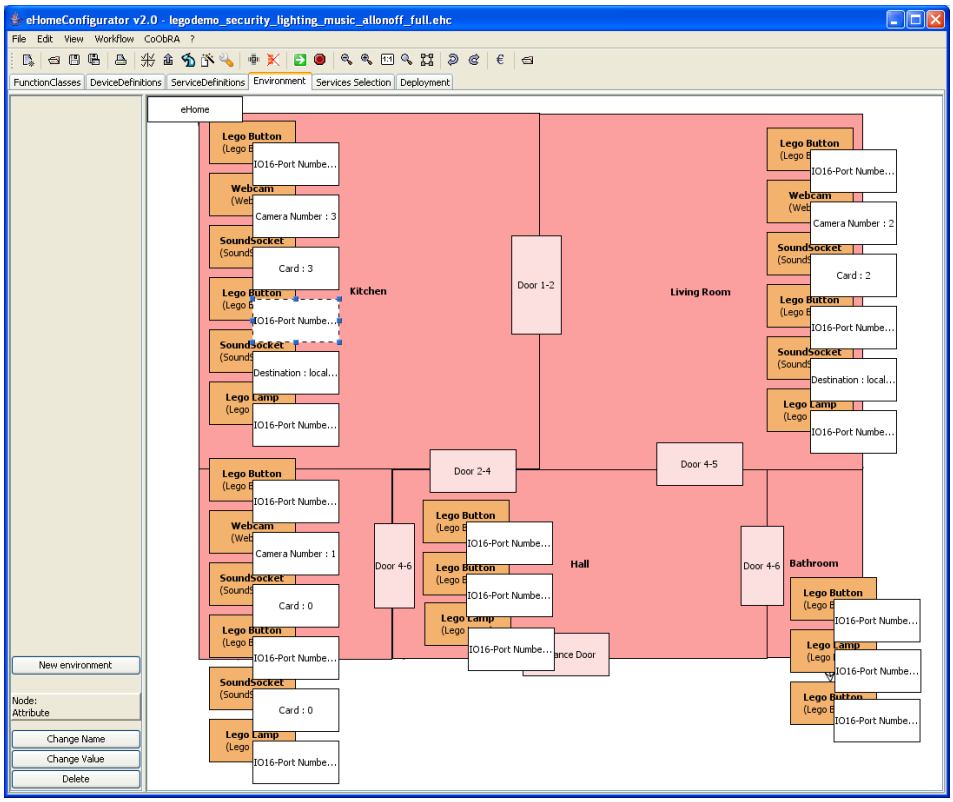

Abbildung 7.35: eHomeConfigurator: Umgebung mit benötigten Geräten

chenden Räumen, angebracht werden, bevor das tatsächliche Deployment (siehe nächster Abschnitt) durchgeführt werden kann.

Das zweite Ergebnis ist wegen seiner Komplexität normalerweise nicht für die Ansicht durch den Bewohner gedacht. Es ist eher die Visualisierung der internen Struktur der Deployment-Konfiguration, der Deployment-Graph (siehe für einen Ausschnitt Abbildung 7.36). Der Graph bietet als eHomeConfigurator-Entwickler einen guten Einstieg, um den Aufbau der internen Datenstruktur zu überprüfen und zu debuggen (siehe hierfür auch Abschnitt 7.7). Die Deployment-Konfiguration beinhaltet alle Informationen, die zur Inbetriebnahme des eHome-Systems, welches die durch den Bewohner anfangs gewählten Dienste erfüllt, benötigt werden. Nun kann diese Datenstruktur durch das Deployment-Werkzeug der eHomeConfigurator-Werkzeugsuite, den Deployer, in Betrieb genommen werden.

7.6 Deployment

Das Deployment läuft vollkommen automatisch ab. Es wird durch Druck auf den Deployment-Knopf (siehe Abbildung 7.37) ausgelöst. Nach Druck auf den Knopf star-

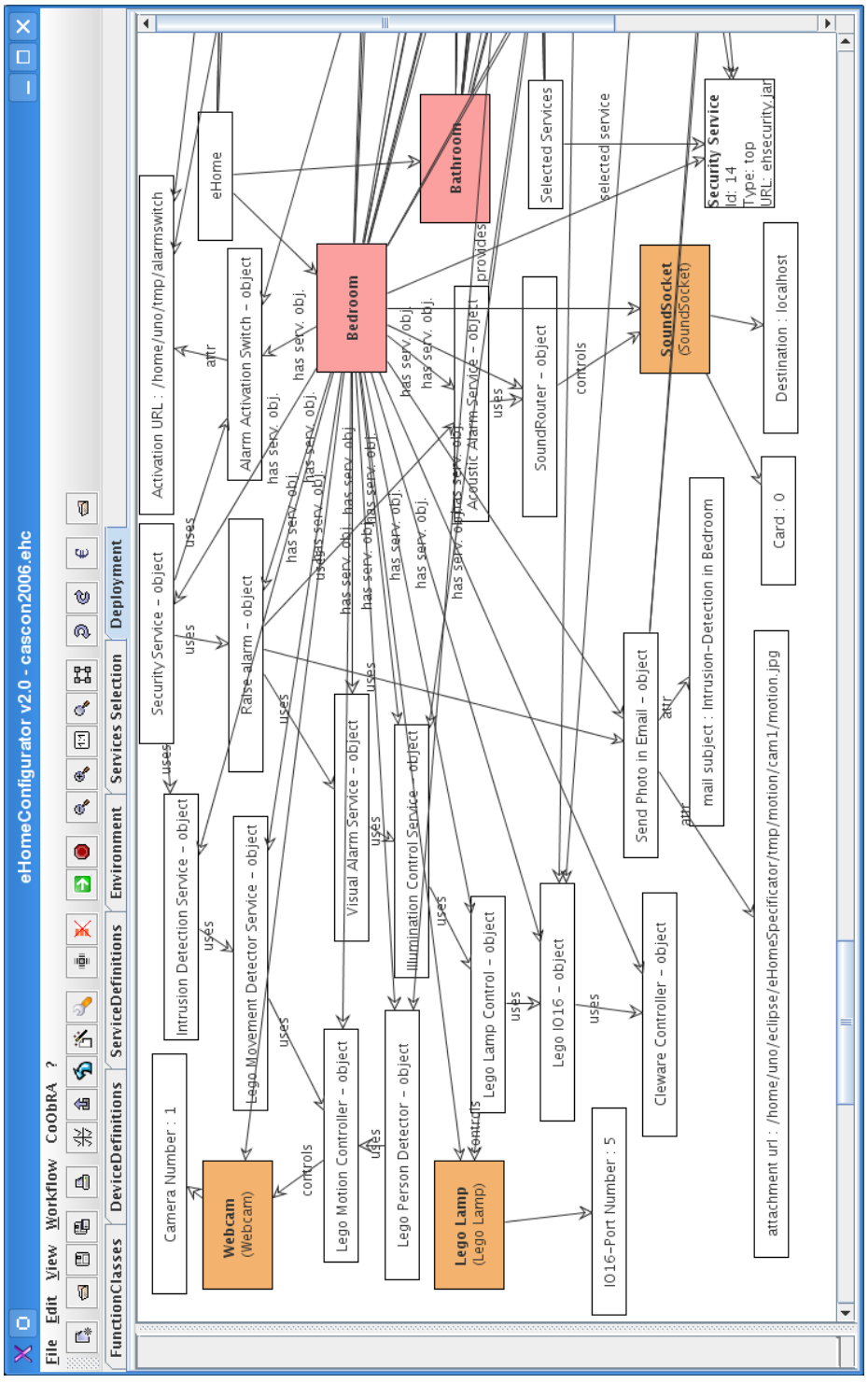

Abbildung 7.36: Sicherheitsdienst-Ausschnitt aus dem kompletten Deploymentgraphen

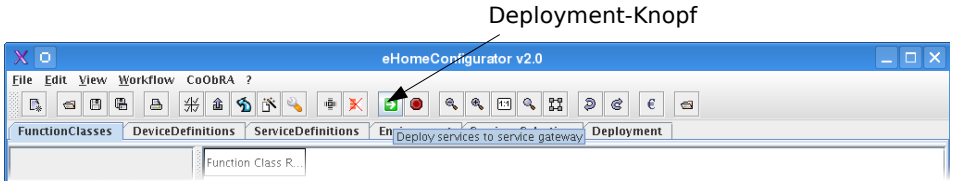

Abbildung 7.37: Knopf zum Starten des Deployment-Werkzeuges

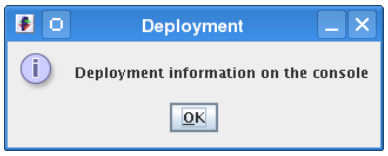

Abbildung 7.38: Dialogbox nach erfolgreichem Deployment

tet das Deployment-Werkzeug und führt ohne weitere Interaktion das Deployment durch. Für die Erläuterung des Algorithmus siehe Abschnitt 6.7. Nach Abschluss des Deployments wird die Dialogbox aus Abbildung 7.38 angezeigt und in der Konsole, in der das OSGi-Rahmenwerk gestartet wurde, ist eine Ausgabe vergleichbar mit Abbildung 7.39 zu sehen.

Ein Undeployment und die Rekonfiguration wurden bisher noch nicht implementiert.

7.7 Debugging

Neben dem Deploymentgraph (Abbildung 7.36) eignet sich der DOBS [GZ02] sehr gut zum Debuggen. DOBS ist ein Werkzeug, welches die Visualisierung des Java-Objekt-Speichers zur Laufzeit einer Java-Applikation ermöglicht. Dies erlaubt dem Entwickler, die Objekt-Struktur durchzusehen und zu durchsuchen, um sich ein Bild des Objektgraphen des Zustandsraumes der virtuellen Java-Maschine zu machen. Er wird über den in Abbildung 7.40 gezeigten Knopf gestartet und zeigt dann das gerade im eHomeConfigurator ausgewählte Objekt an.

Zum Beispiel ist in Abbildung 6.10 das Objekt des Dienstes `MusicFollowsPerson` selektiert. Nach Druck auf den DOBS-Knopf ist das Objekt zu Klasse Service im DOBS-Editor mit allen seinen Attributwerten und Methoden zu sehen (siehe Abbildung 7.41). Ausgehend von diesem Objekt ist es möglich innerhalb des DOBS dessen Kontext zu visualisieren, d.h. assoziierte Objekte anzeigen zu lassen oder sogar Attribute zu verändern oder Methoden der Objekte aufzurufen und deren Ergebnisse anzeigen zu lassen. Es ist also mit dem DOBS möglich, mit der eHome-Modell-Instanz auf der Ebene all seiner Methoden zu interagieren. Wenn eine Zustandsänderung einen physikalischen Effekt, wie das Einschalten einer Lampe, hat, so kann durch das Ändern des Zustands eines `State`-Objekts mit dem DOBS diese Änderung hervorgerufen werden. Durch Aufruf der Methode `iteratorOfServices`, `getEnvirontmentElements` und `iteratorOfAttributes` entsteht Abbildung 7.42.

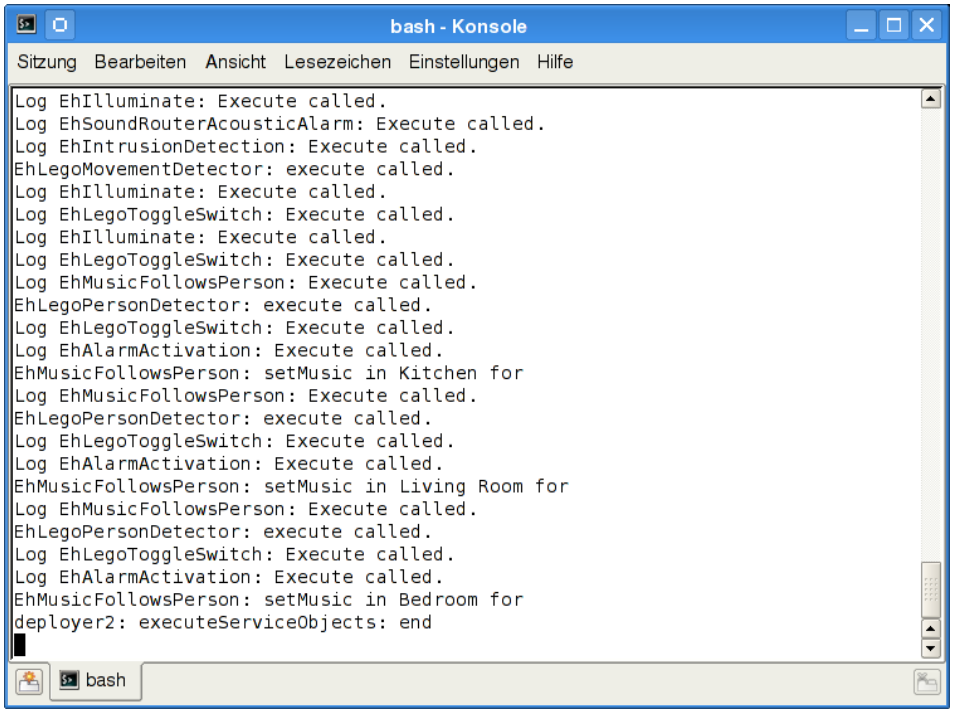

Abbildung 7.39: Konsole des Rahmenwerks nach dem Deployment

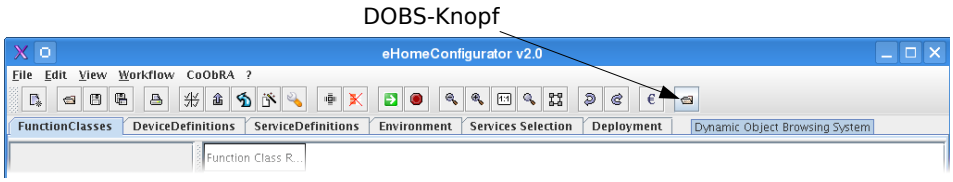

Abbildung 7.40: Knopf zum Starten des DOBS

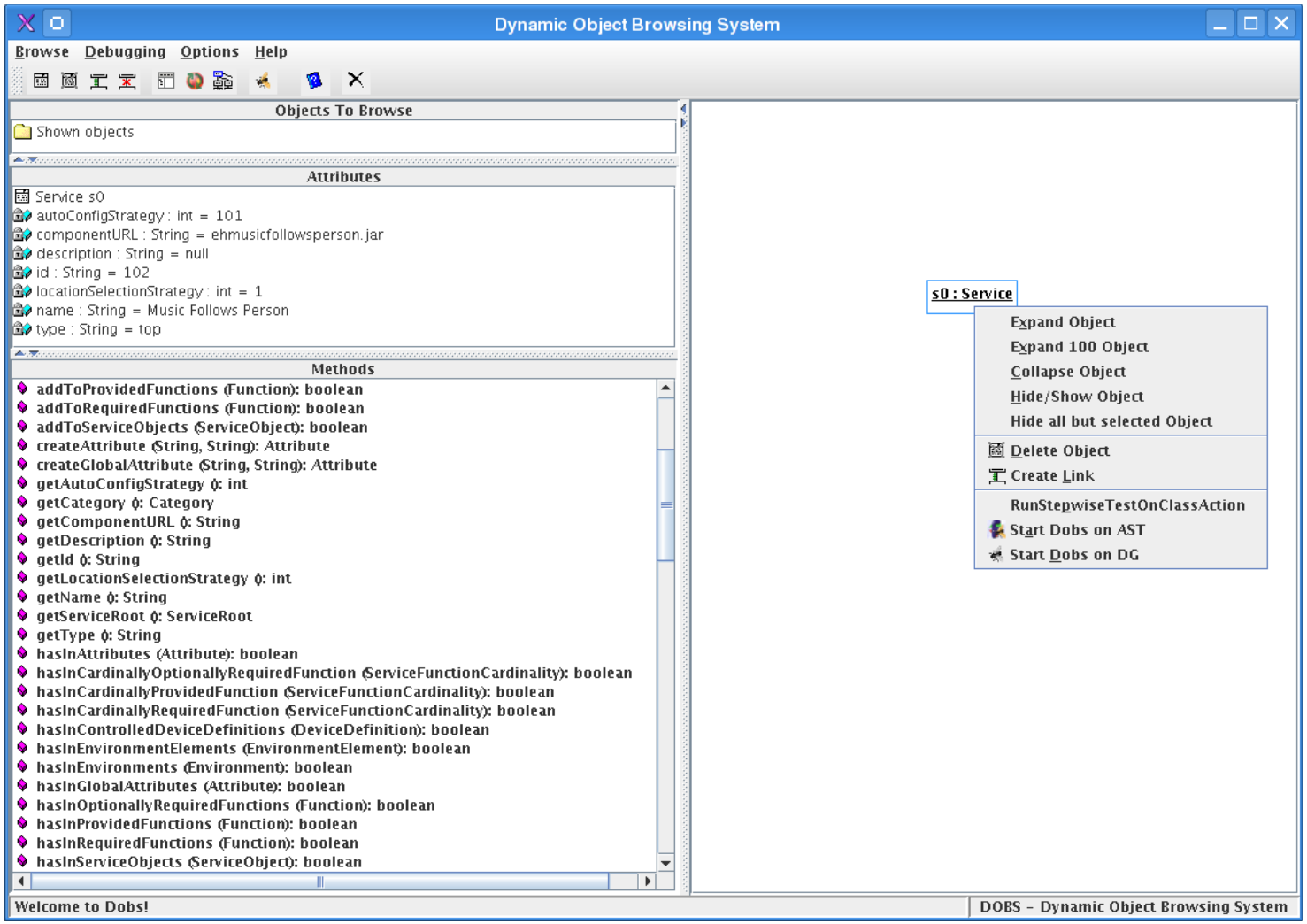

Abbildung 7.41: Bildschirmfoto des DOBS für den Musik-folgt-Person-Dienst

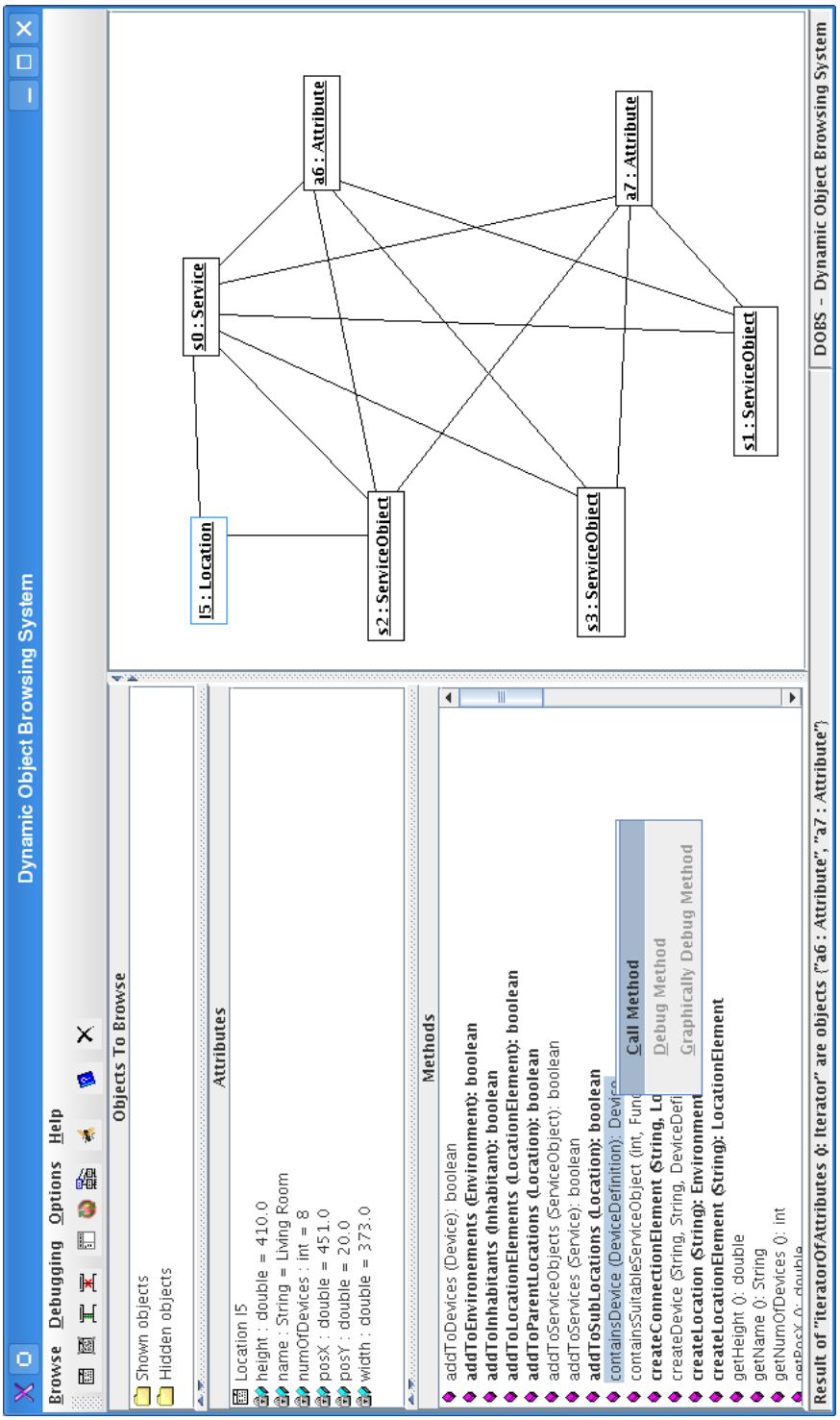

Abbildung 7.42: Bildschirmfoto des DOBS für den Musik-folgt-Person-Dienst

Es macht auch Sinn den Deploymentgraphen und DOBS parallel einzusetzen: Zuerst wird der interessierende Bereich im Deploymentgraphen genauer eingegrenzt und anschließend werden die dort liegenden Objekte mittels des DOBS genauer inspiziert.

So liefert DOBS fortgeschrittene graphische Möglichkeiten, das Programm zu debuggen und einen Überblick über den Zustandsraum des Systems zur Laufzeit zu bekommen und die Reaktionen des Systems auf die Veränderung dieses Zustandsraumes zu überwachen.

7.8 Bewertung

In der ersten Version der eHomeConfigurator-Werkzeugsuite (siehe Abschnitt 6.1 auf Seite 145) musste die Konfigurierungsphase mit Unterstützung sehr unterschiedlicher Werkzeuge durchgeführt werden, die nur wenig Einsparung bei der Entwicklungszeit brachten. Die Bedienung der aktuell vorliegenden Version des eHomeConfigurators wurde von mehreren Studenten der ersten Semester getestet. Dazu wurden Studenten der Arbeitsgruppe aber auch Studenten auf Konferenzen und Workshops [NM06b, Ulr05a], auf denen der Lego-eHome-Demonstrator präsentiert wurde, gebeten, die Konfigurierung und das Deployment nachzuvollziehen. Sie konnten alle nach einer kurzen Vorführung und Einführung von etwa einer halben bis einer Stunde, mehrere Dienste für unterschiedliche Konfigurationen (siehe die verschiedenen Aufbauten in den Abbildung 2.6 und 2.7 auf Seite 33) des Lego-eHome-Demonstrators in weniger als 15 Minuten mithilfe des in Abschnitt 6.6 auf Seite 175 beschriebenen Assistenten konfigurieren und mittels des Deployer-Werkzeugs zur Ausführung bringen. Ihnen standen die Adressdaten der einzelnen physikalisch installierten Geräte (siehe Tabelle 2.1 auf Seite 36) des Lego-eHome-Demonstrators zur Verfügung.

Die kurzen Bedienzeiten durch die Studenten sind nicht verwunderlich, da der Hauptaufwand bei der Konfigurierung die Zuordnung der richtigen Adressparameter bei der Parametrisierung ist. Eine weitaus größere Adressliste würde wahrscheinlich die Zeit vergrößern. Auch eine größere Dienstliste sowie mehr Auswahlmöglichkeiten bei gleichwertigen Unterdiensten würden diese Konfigurierungsprozedur sicher verlängern. Allerdings ist die Auswahl von gleichwertigen Diensten und die Eingabe von Geräteparametern ein für den Endbenutzer vertretbarer Aufwand. Dem gegenüber steht der mühsame Prozess der Programmierung von Diensten, des Einfügens von Geräteparametern in einen Dienstequelltext oder zumindest in XML-Dateien und einer anschließenden Kompilation und des manuellen Deployments des Systems, welcher sich eher in Personenmonaten oder gar Jahren (siehe auch die Angaben zu Entwicklungszeiten im inHaus, Abschnitt 3.2.2 auf Seite 42) als in Minuten oder Stunden messen lässt.

Diese praktischen Erfahrungen geben einen Anhaltspunkt über die Erleichterungen, die mit dem eHomeConfigurator im Bereich der Dienstekonfigurierung und des Deployments zu erzielen sind. Um eine Idee für die Erleichterung im Bereich der Diensteentwicklung zu bekommen, soll hier ein Blick auf die Länge der entwickelten Quelltexte im Vergleich zu der ersten Entwicklung von eHome-Diensten in unterschiedlichen Rahmenwerken in Kapitel 4 auf Seite 53 geworfen werden.

Dazu werden die Ergebnisse aus Abschnitt 4.4 auf Seite 120 (siehe insbesondere Tabelle 4.4) und die Ergebnisse aus Abschnitt 6.9 auf Seite 200 (siehe insbesondere Ta-

belle 6.2 auf Seite 205) betrachtet. In beiden Fällen wurde die Implementierung des vereinfachten Sicherheitsdienst analysiert, zuerst im Rahmen der manuellen Konfigurierung und dann im Rahmen der Unterstützung durch die eHomeConfigurator-Werkzeugsuite im SCD-Prozess.

Für die Implementierung des vereinfachten Sicherheitsdienstes in den drei besprochenen Rahmenwerken wurden ohne Berücksichtigung des Glue- und Strukturcodes durchschnittlich 4000 Zeilen Code geschrieben. Im Rahmen der eHomeConfigurator-Werkzeugsuite sind es nur etwa 1400 Zeilen, wenn auch hier Glue- und Strukturcode (Manifest-Dateien, Lizenzcode) vernachlässigt werden. Ein beachtlicher Teil des Codes ist durch die Konfigurierung und die Unterstützung durch das eHome-Modell nicht mehr nötig. Die Entwicklung der Dienstlogik der Erweiterungsdienste (hier ehsecurity, ehilluminate und ehintrusiondetector) erfordert nur noch 437 Javacodezeilen (ohne Lizenzen und Manifest-Dateien aber inkl. Aktivatoren) gegenüber vorher 826 Javacodezeilen (siehe Tabelle 4.3 auf Seite 120). Bei der Dienstentwicklung kann sich die Programmierung also auf die Umsetzung der eigentlichen Dienstlogik konzentrieren. Der Aufwand für das Anpassen an die Umgebung, die Kommunikation mit dem Rahmenwerk und das Verwalten von Abhängigkeiten entfällt. Die durchschnittliche Länge eines Logikdienstes, ohne den Aktivatorcode beläuft sich auf etwa 100 Zeilen. Von diesen beschreiben in der Regel etwa 10-20 Zeilen, die tatsächliche Dienstlogik (vergleiche Beispieldienstentwicklung in Abschnitt 7.4). In Anbetracht der Kürze dieses Codes, wird vom Übergang zur deklarativen Beschreibung der Dienstlogik (siehe [Kir05, KS05]), die eine unüberschaubare Komplexität bei der Ausführung der Dienste in Bezug auf definierte zeitliche Abfolge hinzufügt, in dieser Arbeit abgeraten.

Wie der Vergleich zeigt, profitieren nicht nur die Erweiterungsdienste, sondern auch die Dienste mit Treiberfunktionalität (ehemail, clewarecontrol, ehlegomovementdetector, ehlegomotioncontrol und ehlegolampcontrol): Es ergibt sich eine Reduktion von 3226 Zeilen auf 986 Zeilen. Dass hier eine noch stärkere Reduktion als bei den Erweiterungsdiensten zu verzeichnen ist, liegt daran, dass das Schalten der selbst angefertigten Legogeräte etwas einfacher ist, als das Schalten von X10-Geräten und eine geschicktere E-Mail-Versenderoutine gewählt wurde. Der Unterschied ist aber kleiner als 500 Codezeilen, so dass auch hier eine Vereinfachung in Größenordnung der Vereinfachung bei Erweiterungsdiensten verzeichnet werden kann.

Ein weiterer großer Vorteil im Rahmen der eHomeConfigurator-Werkzeugsuite ist, dass die Erweiterungsdienste in der Regel nur selten verändert werden müssen, da sie durch die Funktionalitätenabstraktion unabhängig von der verwendeten Hardware sind. Nur für neue Hardware müssen neue Treiber entwickelt werden, deren Entwicklungsaufwand aber auch, wie gerade gesehen, deutlich geschrumpft ist.

Weiterhin fällt beim Vergleich des OSGi-Beispiels (siehe Tabelle 4.3 auf Seite 120) mit der neuen Implementation auf, dass auch eine Verkürzung der Länge der Manifest-Dateien zu erkennen ist (von 356 auf 112). Dies liegt daran, dass das Konfigurierungswerkzeug den Großteil der Import- und Exportbeziehungen der Komponenten automatisch verwaltet und diese über die Kommunikation über die eHome-Modell-Instanz entkoppelt werden.

Der gesamte Deployment-Graph der Konfiguration aller top-level Dienste aus Abschnitt 2.2.1 auf Seite 27 für den Lego-eHome-Demonstrator besteht aus mehr als 200 Knoten (Ausschnitt siehe Abbildung 7.36 auf Seite 237). Die Ausgangsversion des Gra-

phen vor dem Eintritt in die Konfigurierungsphase besteht nur aus 14 Knoten. Die XML-Repräsentation der gesamten Konfiguration besteht aus mehr als 2200 Zeilen, von denen etwa 800 Zeilen Deployment-Informationen beinhalten. Das bedeutet, dass 1400 Zeilen für die Spezifikation der Funktionalitäten, Geräte, Dienste und Umgebungsinformation verwendet werden. Diese Zeile sind allerdings mit einem sehr geringen Aufwand (kleiner einem Personentag) in dem Werkzeug zu spezifizieren. Weiterhin können diese Informationen bis auf die Umgebungsinformationen in jeder anderen Umgebung, in der die gleichen Dienste verwendet werden sollen, sogar mit unterschiedlichem Hardwareeinsatz, wiederverwendet werden.

Diese Informationen und große Teile des Codes, welcher nun automatisch aus dem eHome-Modell erzeugt wird, wurden vorher fest in die Diensteimplementation kodiert. Jetzt werden diese teilweise automatisch und teilweise interaktiv generiert und können in der Konfigurierungsphase bequem angepasst werden.

All diese Beobachtungen zeigen, dass ein wesentlicher Anteil der Kodierung durch die Ergebnisse dieser Arbeit aufgehoben wird. Diese Ergebnisse sind zum einen die Formulierung des SCD-Prozesses, aber insbesondere auch die Unterstützung der Konfigurierung und der Codeentwicklung durch das eHome-Modell und den dort generierten Code.

Natürlich kann durch das Betrachten der Codezeilen nur eine ungefähre Aussage über den Gewinn durch den Einsatz des eHomeConfigurators und des SCD-Prozesses gemacht werden. Dennoch sind eine signifikante Aufwandsreduktion in der initialen Entwicklungsphase und der Wiederverwendung aber erst recht in der Betreuung von eHome-Systemen und der Übertragung in andere Umgebungen durch den Einsatz der eHomeConfigurator-Werkzeugsuite zu erkennen.

Kapitel 8

Verwandte Arbeiten

Dieses Kapitel widmet sich dem Vergleich der in dieser Arbeit vorgestellten Ansätze mit ähnlichen Problemstellungen der Unterstützung des Entwicklungsprozesses und insbesondere der Konfigurierung. Da die Problemstellungen anderer Arbeiten nicht deckungsgleich mit dieser sind, werden ähnliche Teilbereiche zu der vorliegenden Arbeit in Bezug gesetzt und bewertet.

8.1 Visionen und eHome-Umgebungen

Es existieren eine Reihe von Projekten, die sich mit smarten Umgebungen, wie auch eHomes welche sind, befassen [Wal]. Eine Unterstützung des Entwicklungsprozesses steht bei diesen Projekten allerdings sehr selten im Vordergrund. Es geht hier meist darum, neue Dienste zu schaffen und neue Visionen zu finden. Einige der Umgebungen, die sich speziell mit physikalischen eHomes beschäftigen, wurden auch bereits für den Stand der Technik in Abschnitt 3 vorgestellt.

8.1.1 Szenarien für Ambient Intelligence

Im Jahr 2001 veröffentlichte die IST (Information Society Technology) Advisory Group (ISTAG) [IST01a] ein Szenarienpapier [IST01b] welches die Möglichkeiten für Ambient Intelligence im Jahre 2010 aufzeichnen sollte. Die ISTAG berät die Europäische Kommission bezüglich Strategien zur Priorisierung von Forschungsschwerpunkten in den Forschungs-Rahmenprogrammen. Es werden vier Szenarien vorgestellt. Ihnen gemein ist, dass es immer eine vermittelnde Assistenz gibt, die beim Bewegen einer Person in einer Umgebung Authentifizierungs- und Verhandlungsaufgaben übernimmt. So soll beim Fernreisen ein Verzicht auf Notebook, Beamer, Drucker, oder Mobiltelefon möglich werden. Das Reisevisum soll sich im Internet besorgen lassen und die Zollabfertigung einfach durchlaufen werden können, da alle Checks im Vorbeigehen von der am Körper getragenen Kommunikationseinheit und der lokalen Infrastruktur erledigt werden. Autos werden automatisch reserviert und warten abholbereit an einer Parkbox, die sich akustisch bemerkbar macht. Das Auto öffnet sich beim Herangehen und die eingebaute Fahrassistenz

unterstützt bei der Fahrt zum Hotel. Während der Fahrt lassen sich Telefongespräche per Stimmkommando entgegen nehmen. Das Verkehrsleitsystem kann mit der Kommunikationseinheit des Fahrers bzw. dem gefahrenen Auto interagieren. So kann es den Fahrer durch bestimmte bezahlbare Dienste vor andern Autos bevorzugen und schneller vorankommen lassen. Bei Smoggefahr kann aber auch die Maximalgeschwindigkeit aller Autos dynamisch heruntergeregelt werden. Am Hotel angekommen kann das Zimmer automatisch das Profil des Reisenden übernehmen und Raum-Temperatur, Licht, Musik und Filmauswahl entsprechend den persönlichen Vorlieben einstellen. Sprachsteuerung spielt eine zentrale Rolle in den Szenarien. Durch sie lassen sich das Einlaufen eines Bades aber auch verschiedene Arten von Kommunikationsverbindungen initiieren. Bei solcher Kommunikation vermittelt die Umgebungsintelligenz und erlaubt das synchronisierte Betrachten von medialen Inhalten und die gleichzeitige Diskussion oder das Arbeiten mit diesen. Die Geräte können gewisse Symptome am Körper registrieren und Vorschläge zur Kompensation machen. Auch sind sie in der Lage, beim Einkaufen auf spezielle Angebote, die den Träger tatsächlich interessieren, hinzuweisen. Die Vision geht sogar so weit, dass ein Teil der Kommunikation von den entsprechenden Geräten selbst übernommen werden kann, um lästige Anrufer[1] zum Auflegen zu bewegen und nur in Notfällen durchzustellen. Die Geräte sollen in der Lage sein, vom Träger zu lernen und so ihr Verhalten, insbesondere das Kommunikationsverhalten, an die Gepflogenheiten des Trägers anzupassen. Natürlich sind diese Geräte in der Lage, automatisch Bezahlungen vorzunehmen und gewünschte Authentifizierungen bei Kontrollen durchzuführen. In Arbeitsgruppen können die Geräte so weit gehen, dass sie die Zusammenarbeit der Teilnehmer organisieren und die Gruppenzusammensetzungen regulieren, um ein optimales Arbeits- und Lernergebnis zu erreichen.

Der Aufwand der Programmierung und die dynamische Verschaltung der unzähligen Dienste, die hier zum Einsatz kommen müssen, wird hier gar nicht angesprochen. Die Forschung im Bereich der Ambient Intelligence scheint sehr in Richtung der erweiterte Mobiltelefonie zu zielen.

Die Annahme, Software für solche Geräte bis 2010 umzusetzen ist sicher etwas gewagt. Auch sind einige Fähigkeiten wie das Abweisen lästiger Personen weniger wünschenswert, da die Informatik sicher noch etwas brauchen wird, um Computer zu entwickeln, die die Folgen einer solchen Reaktion adäquat abschätzen können. Auch wenn die Szenarien sicher herausfordernd sind, so fehlt insbesondere die Betrachtung von Verhalten im Fehlerfalle. Was passiert also, wenn eine Authentifizierung nicht funktioniert, was passiert, wenn ein erlerntes Verhalten nicht gewünscht ist?

Die Wahl der Szenarien belegt, wie groß die Not in diesem Bereich ist, Anwendungsszenarien und neue Dienste zu finden. Dies ist einer der Gründe, warum sich die vorliegende Arbeit eher auf die bereits realisierbaren Dienste aus dem Bereich der Gerätesteuerung, Sicherheit und Multimedia bezieht.

[1] Als ein solcher lästiger Anrufer wird in dem Papier tatsächlich die eigene Frau angegeben. Ich weise ausdrücklich darauf hin, dass das keine eigenen Idee, sondern nur ein Zitat aus einem hochrangigen EU-Papier ist, welches zur Evaluation für die Freigabe von mehreren Millionen Euro benutzt wurde.

8.1.2 Amigo

Mit Amigo [Ami05] ist ein Projekt ins Leben gerufen worden, welches ebenfalls die Entwicklung im eHome-Bereich vorantreiben soll. Beteiligt sind 15 Partner aus ganz Europa, zu denen unter anderem Philips, Microsoft und die France Telecom zählen. Das Projekt hat eine Laufzeit von 42 Monaten und ist mit einem Budget von 24 Millionen Euro ausgestattet.

Ziel von Amigo ist die Überwindung der Heterogenität von Geräten aus vier verschiedenen Marktsegmenten:

- Mobile Geräte: Handys, PDAs, Notebooks, ...
- PCs: Desktop Rechner, Drucker, Monitore, ...
- Haushaltsgeräte: Waschmaschinen, Trockner, ...
- Elektronik: Fernseher, Stereoanlagen, ...

Diese vier Bereiche sind so strikt voneinander getrennt, dass in jedem dieser Sektoren ein anderer Hersteller marktbeherrschend ist. Versuche, in andere Marktsegmente vorzudringen, fallen diesen Unternehmen jedoch meist schwer. Darüber hinaus entsprechen die Geräte der verschiedenen Bereiche unterschiedlichen Standards. Für eine vollständige Vernetzung im eHome muss diese Trennung jedoch überwunden werden.

Erklärtes Ziel von Amigo ist die Erarbeitung neuer Schnittstellen und Protokolle, um Geräte aus allen vier Sektoren vernetzen zu können. Es soll erforscht werden, welche Mehrwertdienste auf diese Weise realisierbar sind, wobei auch die Skalierbarkeit solcher Systeme betrachtet werden soll. Die Konfiguration aller Geräte soll dabei möglichst automatisch durchgeführt werden können.

8.1.3 The Aware Home

The Aware Home [KOA+99] ist ein Projekt, dessen Fokus auf Ubiquitous Computing liegt [Wei91]. Es wird erforscht, welche Dienste in einer bereits konfigurierten und ideal ausgestatteten Umgebung möglich sind. Die Umgebung ist dabei so konzipiert, dass über die Bewohner einer Beispielwohnung möglichst viele Informationen gesammelt und ausgewertet werden. Diese dienen dazu, Modelle des menschlichen Verhaltens zu konstruieren, die Programme bei ihrer Entscheidungsfindung unterstützen.

8.1.4 Intelligent Room Project

Das *Intelligent Room Project* [Bro97] ist eine interaktive Umgebung, die alltägliche Verhaltensweisen der Menschen, die sich in dieser aufhalten, analysiert. Die Beobachtungen werden mit Kameras und Mikrofonen vorgenommen. Programme versuchen das Gesprochene und die Handlungen der Menschen auszuwerten. Fokus des Projekts ist die Erforschung von Künstlicher Intelligenz (KI), die durch die Umgebung bereitgestellt werden soll.

8.1.5 EasyLiving

Schwerpunkt des *EasyLiving-Projekts* [BMK+00] ist die Entwicklung von Architekturen und Techniken für intelligente Umgebungen. Ziel des Projekts ist es, die Umgebung um neue Funktionen zu erweitern. Beispiele dafür sind automatische Lichtkontrolle, das Abspielen von Musik am Aufenthaltsort des Nutzers und Darstellung von Informationen auf einem dem Benutzer am nächsten gelegenen Bildschirm. Interaktion mit der Umgebung geschieht dabei durch eine Vielzahl von Fernbedienungen. Die Umgebung ist dabei vorkonfiguriert und Geräte und Applikationen können durch spontane Vernetzung zur Laufzeit hinzugefügt werden.

8.1.6 The Adaptive House

Das *Adaptive House* [Moz98] ist ein Forschungsprojekt mit Fokus auf der Entwicklung eines Hauses, dass sich durch Beobachtung selbsttätig an die Benutzer anpasst. Ein Ziel des Projekts ist, jegliche Benutzeroberfläche oder explizite Steuerung obsolet zu machen. Dazu werden die Handlungen der Benutzer aufgezeichnet und daraus Profile entwickelt. Diese sollen nach Erreichen eines guten Profils manuelle Aktionen der Benutzer automatisieren. Die Raumtemperatur soll beispielsweise nach ausreichender Beobachtung automatisch anhand der ermittelten Daten eingestellt werden. Greift der Benutzer in die automatisch eingestellte Temperatur ein, dann wird diese Information in zukünftige Entscheidungen mit einbezogen. Der Ansatz benutzt Konzepte der Künstlichen Intelligenz (KI) und wendet diese in einer vorkonfigurierten Umgebung an.

8.1.7 Microsoft's Home Entertainment Strategy

In "Directions on Microsoft" (2/2005) ist die "Microsoft Home Entertainment Strategy" beschrieben [Ros05]. Dieses Dokument beschreibt, wie Microsoft versuchen möchte, den Windows-PC als Mittelpunkt sämtlicher Unterhaltungselektronik im modernen Heim zu etablieren. Er übernimmt hier eine ähnliche Funktion, wie das Service-Gateway in eHome-Systemen. Es sollen also die wesentlichen Dienste auf einem zentralen Server laufen und kleine Geräte, sogenannte spokes, sich mit diesem in möglichst einfacher Weise verbinden lassen. Angespornt durch den Erfolg von Apple in diesem Bereich, versucht auch Microsoft mit einer größeren Protokoll- und Geräteunterstützung das Marktsegment der Multimediadienste im eHome zu besetzen. Als Framework ist hier der Windows Media Player geplant, der in dieser Domäne eine einfache Ansteuerung und Diensteentwicklung erlaubt. Microsofts Aktivitäten im eHome-Bereich beschränken sich allerdings momentan auf den Multimedia-Bereich, da dieser als einzig lukrativer Bereich angesehen wird. Domänen überspannende Dienste werden nicht betrachtet. Somit findet sich in diesem Papier auch keine Information, wie für solche Dienste die Entwicklung vereinfacht werden könnte.

8.2 Unterstützende Infrastruktur

8.2.1 Stanford Interactive Workspaces und iROS

Der *iRoom* ist ein Labor des *Interactive Workspaces Projekts* [JFW02] und betrachtet die Interaktion zwischen PCs, Laptops, PDAs und anderen transportablen Geräten in einem einzelnen Konferenzraum. Dieser Raum ist mit mehreren Bildschirmen und einer Vielzahl von Eingabemöglichkeiten ausgestattet. Dazu zählen berührungssensitive Oberflächen, Mikrofone und Kameras. Ziel des Projekts ist es, nicht die Umgebung intelligenter zu machen, indem diese auf den Benutzer reagiert, sondern es wird versucht, eine verbesserte Arbeitsumgebung zu schaffen. Diese soll dem Benutzer ermöglichen, intuitiv und mit geringem Aufwand seine Aufgaben zu verrichten. Dabei soll ihm eine Vielzahl von Interaktionsmöglichkeiten mit und zwischen den Geräten bereitgestellt werden.

Dieser Ansatz beschäftigt sich nicht mit Dienstekomposition, er ist also sehr geräteorientiert. Zwar wird das hier eingesetzte Kommunikationsrahmenwerk iROS, welches auf einem Event-Heap [JF02] basiert, explizit für den Einsatz in smart Homes gepriesen, dennoch handelt es sich um reines ungetyptes Kommunikationsrahmenwerk. Dieses ist plattformübergreifend und sehr ressourcensparend. Es erleichtert aber nicht die Programmierung komplexer Erweiterungs-Dienste, da die angenommene Dienstarchitektur, wie in der Arbeit von Michael Kirchhof [Kir05], flach ist.

8.2.2 SmartHome User Interface

Das *SmartHome User Interface* (SUI) [DY96] stellt eine Möglichkeit zur Steuerung von Geräten und Diensten einer eHome-Umgebung über das Internet zur Verfügung. Der Begriff **SmartHome** stammt aus dem englischsprachigen Raum und wird dort synonym zum Begriff eHome verwendet. Bei SUI handelt es sich um eine zum HTML 3.0 Standard [W3C] konforme Webseite, die zusätzlich von Java Gebrauch macht. Auf diese Weise können Dienste und Geräte mit Hilfe jedes Webbrowsers, der diese HTML-Version unterstützt, visualisiert und gesteuert werden. Darüber hinaus existiert auch die Möglichkeit, die eHome-Umgebung dreidimensional darzustellen. Zu diesem Zweck wird vom VRML 1.0 Standard [Web03] Gebrauch gemacht.

Im Unterschied zu dem in dieser Arbeit vorgestellten Werkzeug existiert jedoch keinerlei Unterstützung des Modellierungs- und Konfigurierungsprozesses. Zudem geht der hier verfolgte Ansatz davon aus, dass ein direktes Zugreifen des Bewohners auf die installierten Dienste und Geräte nur in Ausnahmefällen notwendig ist. Für diese Fälle bietet jedoch auch der eHomeConfigurator Steuerungsmöglichkeiten, auf die aus der Ferne zugegriffen werden kann, solange Zugang zum an das Internet angebundene Gateway besteht. Eine dreidimensionale Darstellung der Umgebung wird allerdings nicht zur Verfügung gestellt, so dass SUI in dieser Hinsicht einen Vorteil bietet.

Ein kostensenkender Effekt durch den Einsatz von SmartHome User Interface ist ebenfalls unwahrscheinlich, da die von Experten durchzuführenden, kostenintensiven Prozesse in keiner Weise erleichtert oder beschleunigt werden.

8.2.3 Java Context Awareness Framework (JCAF)

Java Context Awareness Framework, oder kurz JCAF [Bar05], ist der Name eines auf Java basierenden Rahmenwerks für Applikationen, die auf Sensordaten ihrer Umgebung zurückgreifen (*Context-Aware Applications*). Es besteht aus zwei Teilen, einer Laufzeit-Infrastruktur und einem Rahmenwerk zur Programmierung, welches die Schnittstellen zur Entwicklung neuer Dienste zur Verfügung stellt.

Die Laufzeitumgebung von JCAF folgt vier Prinzipien:

- **verteilte** und **kooperierende** Dienste
- **ereignisbasierte Infrastruktur**
- **Sicherheit**
- **Erweiterbarkeit**

Mit Verteilung und Kooperation ist nicht der parallele Einsatz mehrerer solcher Rahmenwerke gemeint, sondern lediglich eine Kapselung verschiedener Dienste, welche gleichzeitig auf dem Rahmenwerk laufen. Solche Dienste sind ereignisbasiert, das heißt, dass sie auf bestimmte Ereignisse der Umgebung warten und reagieren. Hierbei kann es sich beispielsweise um das Erreichen einer gewissen Uhrzeit oder um die Auslösung eines Sensors handeln. Da nur autorisierte Dienste Zugriff auf die Ereignisse und Daten der Umgebung haben sollen, sind bestimmte Sicherheitsaspekte realisiert. Der Unterpunkt Erweiterbarkeit bezieht sich auf das dynamische Starten, Stoppen und Austauschen von Diensten zur Laufzeit, ohne das Rahmenwerk zu diesem Zweck neu starten zu müssen.

JCAF wurde mit dem Ziel entwickelt, es in der Forschung und in der Lehre einzusetzen. Es bietet im Vergleich zu OSGi keinerlei Vorteile, so dass es fraglich ist, warum bei der Wahl des auf einem Gateway laufenden Rahmenwerks JCAF der Vorzug gegeben werden sollte, da OSGi eindeutig einen größeren Funktionsumfang bietet.

8.3 Prozess-, Entwicklungs- und Konfigurierungsunterstützung

8.3.1 Constraint Satisfaction Probleme

Der Begriff *Constraint Satisfaction Problem* (CSP) [RN03], der aus der Künstlichen Intelligenz herrührt, bezeichnet die Lösung von Aufgaben anhand der Spezifikation von Randbedingungen (*Constraints*). Die grundsätzliche Vorgehensweise beruht darauf, zu einer Menge von Variablen Randbedingungen anzugeben, die deren Wertebereich einschränken. Die angegebenen Randbedingungen können die Werte, die Variablen haben dürfen, auf mehrere Weisen einschränken. Ein Constraint kann auf eine beliebige Menge von Variablen Bezug nehmen und für diese durch eine Funktion definieren, wann dieses Constraint erfüllt ist. Eine einfache Definition eines Constraints legt beispielsweise die Summe aller Variablen auf einen konstanten Wert fest. Ein Constraint Satisfaction Problem ist erfüllt, wenn es eine Belegung der Variablen gibt, die alle Constraints erfüllen.

Der Ansatz, den Constraint Satisfaction Probleme verfolgen, ist eine deklarative Spezifikation aller Randbedingungen. Damit daraus eine Lösung entsteht, werden Algorithmen angewendet, die als Ergebnis eine Lösungsmenge von gültigen Belegungen der Variablen liefern. Einfache CSPs haben Variablen mit einem begrenzten Wertebereich, der aus diskreten Werten, wie beispielsweise den natürlichen Zahlen, besteht. Eine Strategie zur Lösung eines solchen diskreten und endlichen CSP ist es, alle möglichen Belegungen jeder Variable auf Erfüllbarkeit zu testen. Verbesserte Algorithmen versuchen den Baum der zu testenden Möglichkeiten durch geeignete Heuristiken zu verkleinern. Allen diesen Ansätzen gemein ist jedoch die exponentielle worst-case Komplexität. Die Anzahl der Lösungen ist exponentiell zur Menge der Variablen.

Existiert kein begrenzter Wertebereich der Variablen, weil diese beispielsweise die zeitliche Abfolge zweier Ereignisse darstellen und die Zeit keinen beschränkten Wertebereich besitzt, ist es nicht möglich alle Variablenbelegungen sukzessive auf Erfüllbarkeit zu testen. Die Anzahl zu testender Möglichkeiten ist in diesem Fall unbeschränkt. Es gibt für solche CSPs Lösungswege, die hier jedoch nicht beschrieben werden. Ein solches CSP ist auch nur lösbar, falls die Variablen linear sind. Für CSPs mit nichtlinearen Variablen existiert kein Lösungsalgorithmus.

Die Umformulierung des Ansatzes dieser Arbeit in ein CSP würde keinen Vorteil bringen. Die vergleichsweise einfache Gewinnung von Lösungen wird mit hohem Rechenaufwand erkauft. Darüber hinaus müsste ein großer Aufwand für die Erfassung der strukturellen Informationen getrieben werden, damit eine ähnlich intuitive Eingabemöglichkeit wie die in dieser Arbeit vorgestellte ermöglicht wird. Die hier verwendete graphbasierte Darstellung der Spezifikation müsste in Constraints umformuliert werden, oder so umstrukturiert werden, dass daraus automatisch eine Constraintmenge herleitbar ist.

8.3.2 Automatic Configuration Of Component-Based Distributed Systems

Die Dissertation *Automatic Configuration Of Component-Based Distributed Systems* [Kon00] beschäftigt sich mit der automatischen Konfiguration und Rekonfiguration von komponentenbasierten verteilten Systemen. Als Anwendungsfälle werden die Konfiguration eines Betriebssystems und eines Multimedia-Verteilsystems betrachtet.

Zentraler Bestandteil der Arbeit ist die Verwaltung der Abhängigkeiten zwischen Komponenten. Statische Abhängigkeiten zwischen Komponenten lassen sich vor der Laufzeit in einem dafür erstellten Textformat spezifizieren. Die vorgestellte automatische Konfiguration basiert auf den spezifizierten statischen Abhängigkeiten. Aus diesen Informationen werden COMPONENT CONFIGURATORS erstellt, welche zur Laufzeit der Komponenten aktiv sind.

Der erste betrachtete Anwendungsfall der Dissertation stellt einen Netzwerk-PC dar. Ein Benutzer, der sich auf diesem einloggt, soll ein System vorfinden, auf dem die von ihm aufgelisteten Applikationen lauffähig sind. Die Applikationen besitzen als statische Abhängigkeiten Listen von benötigten Komponenten, welche über das Netzwerk auf den jeweiligen PC, an dem sich der Benutzer einloggt, geladen werden. Die Verwaltung der herunterladbaren Komponenten geschieht zentral über ein *Component Repository*, einen addressierbaren Dateispeicher. Da dieser Anwendungsfall eine relativ gute Überdeckung

mit dem hier vorgestellten Ansatz besitzt und der zweite Anwendungsfall der Multimedia-Distribution einen geringen, wird nur ein Vergleich zum ersten gezogen.

Die von Applikationen, Komponenten und dem Betriebssystem spezifizierbaren benötigten Komponenten zerfallen in drei Kategorien:

- Art der Hardwareressourcen die zur Ausführung vorhanden sein müssen
- Kapazität der Hardwareressourcen die benötigt werden
- Benötigte Softwarekomponenten

Der Ansatz ist vergleichbar mit dem in dieser Arbeit vorgestellten. Die benötigten Hardwareressourcen sind mit der Abhängigkeit eines Dienstes von Geräten identifizierbar. Wird ein Dienst im eHomeConfigurator zur Konfigurierung ausgewählt, so wird in dieser durch das Konfigurierungswerkzeug sichergestellt, dass die benötigten Hardwareressourcen und abhängigen Softwarekomponenten vorhanden sind. Das Szenario dieser Arbeit erfordert jedoch im Unterschied zu dem oben genannten Ansatz, dass fehlende Hardwareressourcen beschafft werden müssen, damit ein Dienst lauffähig ist. Die benötigten Softwarekomponenten werden in dieser Arbeit in der Konfigurierungsphase anhand der spezifizierten Abhängigkeiten zur Konfiguration also dem Deploymentdokument hinzugefügt.

Die Component Configurators sind für die Kommunikation zwischen Komponenten zur Laufzeit zuständig. Die Schaffung der Component Configurators wurde notwendig, da die Anwendungsfälle der Dissertation nicht für Komponentenrahmenwerke erstellt wurden. Die in der Dissertation verwendeten Methoden zur Rekonfiguration mittels mobiler Agenten werden in dieser Arbeit durch das Komponentenrahmenwerk abgedeckt. Diese entsprechen dem Hinzufügen weiterer Komponenten und dem Deinstallieren von Komponenten.

8.3.3 Integrierte Low-Cost eHome-Systeme – Prozesse und Infrastrukturen

Die Dissertation *Integrierte Low-Cost eHome-Systeme – Prozesse und Infrastrukturen* von Michael Kirchhof beschreibt eHome-Systeme in Bezug auf die mit ihnen verbundene elektronische Geschäftsabwicklung. In dieser Arbeit wird der gesamte Geschäftsprozess von der Bestellung eines eHome-Dienstes durch den Kunden bis hin zur Installation und Wartung des Dienstes beim Kunden genau beschrieben.

In der Arbeit befinden sich auch Vorschläge, den Entwicklungsprozess zu vereinfachen, indem die Implementation der Dienste nicht mehr imperativ sondern regelbasiert durchgeführt wird. Diese Strategie hat sich in der Praxis als nicht erfolgreich erwiesen, da zeitnahe Reaktion wegen des immanenten Indeterminimus in solchen regelbasierten Sprachen nicht gegeben waren. Es konnte also passieren, dass eine Lampe angeschaltet wurde, diese aber erst nach mehreren Sekunden tatsächlich geschaltet wurde, da keine Reihenfolge der Fakten festgelegt werden konnte. Auch war die Formulierung der Regeln nicht so einfach, wie es suggeriert wird. Durch die untypisierte Übergabe der Daten, wegen der Limitierung der regelbasierten Sprache auf Strings, stieg die Fehleranfälligkeit

auf ein nicht mehr erträgliches Maß. Deshalb wurde in der vorliegenden Arbeit auch ein imperativer objektorientierter Ansatz verfolgt.

Weiterhin ermöglicht die Dienstentwicklung keine Komposition von Diensten aus Unterdiensten, da dort nur eine dreischichtige flache Dienstarchitektur aus Geräteebene, Treiberebene und Dienstebene vorgesehen ist (vergleiche PowerArchitecture).

8.3.4 A Task-Driven Design Model for Collaborative AmI Systems

Arroyo et al. stellen ein Aufgaben-getriebenes Design-Modell für ambient intelligente Systeme vor [AGLH06]. Dieses erlaubt die Repräsentation von Konzepten wie Aufgaben oder Gesetzen und physikalischen Objekten wie Geräte, die in Aufgaben eingesetzt werden können, oder Computern und Menschen, die Aufgaben ausführen. Die Kommunikation arbeitet auf einem Blackboard (eine Art Event-Heap oder Tupel-Raum), welcher die Informationen des gesamten Systems speichert. Dies erlaubt eine sehr flexible Art der Kommunikation, da jeder Teil des Systems gerade für den Teil der Informationen auf dem Blackboard verantwortlich ist, der für ihn relevant ist. Dies entspricht dem Ansatz in dieser Arbeit, in der auch alle Informationen des gesamten Systems in der eHome-Modell-Instanz bzw. der Konfiguration abgespeichert werden und von den einzelnen Dienstobjekten nur der Teil verwaltet wird, der auch für dieses Objekt relevant ist.

Auch die Modellierung abstrakter Aufgaben, physikalischer Objekte und Relationen ähnelt dem in dieser Arbeit vorgestellten eHome-Modell. Allerdings ist das Modell Arroyos nicht in einen Entwicklungs- und Konfigurierungsprozess eingebettet und unterstützt somit nicht die Entwicklung von Dienstkomponenten oder – wie dort genannt – abstrakter Aufgaben, das Auflösen von Abhängigkeiten und die Parametrisierung, der einzelnen Elemente wie Dienste oder Ressourcen.

In Bezug auf Kontrakte der einzelnen Modelleinheiten bietet das dort vorgestellte Modell eine höhere Ausdrucksstärke als das hier vorgestellte eHome-Modell. Diese könnte in weiterführenden Arbeiten in das eHome-Modell integriert werden.

Da es bisher noch keine Werkzeuge gibt, die auf dem von Arroyo vorgestellten Modell arbeiten, können diese nicht mit dem eHomeConfigurator verglichen werden.

8.3.5 SUPPLE

Bei SUPPLE [GW04] handelt es sich um ein Werkzeug zur automatischen Erstellung von Benutzeroberflächen für Geräte in einer eHome-Umgebung. Zu diesem Zweck benötigt SUPPLE drei Eingaben:

- eine **funktionale Spezifikation**,
- ein **Modell des Gerätes** und
- ein **Modell des Benutzers**.

Mit Hilfe der funktionalen Spezifikation wird festgelegt, welche Datentypen zwischen Benutzer und Gerät ausgetauscht werden müssen. Hierzu wurde eigens eine Beschreibungssprache entwickelt, mit deren Hilfe Datentypen und Wertebereiche festgelegt werden können. Betrachtet man beispielsweise eine Stereoanlage, so handelt es sich hierbei unter anderem um Zahlenwerte für Lautstärke, Balance, Höhen und Tiefen.

Im Gerätemodell sind alle manipulierbaren Vorrichtungen (engl. `Widgets`) des betreffenden Gerätes notiert. Zu diesen existiert eine Kostenfunktion, welche abschätzt, wie aufwendig eine Manipulation für den Benutzer mit Hilfe der vom Gerät zur Verfügung gestellten Interaktionsmöglichkeiten ist. Im betrachteten Beispiel einer Stereoanlage ist der Lautstärkeregler ein solches Widget. Im Vergleich zu den unter einer Klappe versteckten Balance-, Bass- und Höhenreglern, ist er leichter zu erreichen, so dass seine Kosten niedriger sind.

Typische Anwenderaktivitäten sind im Benutzermodell festgehalten. Eine solche typische Verhaltensweise ist das Einschalten der Stereoanlage, das Auswählen einer entsprechenden Audioquelle, wie beispielsweise des CD-Players und das Starten des Abspielvorgangs.

Zur automatischen Oberflächengenerierung durchsucht SUPPLE alle möglichen Anordnungskombinationen nach einer kostenminimalen Lösung. Die Suche erfolgt anhand einer Heuristik, welche zusätzlich durch den Branch & Bound-Ansatz [HO02] optimiert wird. Zusätzlich können bestimmte Entscheidungen des Algorithmus' von Hand gesetzt werden. Auf diese Weise kann bei der Erstellung der Benutzeroberfläche Einfluss auf das Layout genommen werden.

SUPPLE ist in Java [Sunb] implementiert, so dass sich mit Hilfe dieses Werkzeugs auf allen Plattformen, für welche eine Java Laufzeitumgebung existiert, Oberflächen erstellen lassen. Hierzu zählen auch Geräte mit knappen Ressourcen, wie beispielsweise Personal Digital Assistents (PDAs). Auf eine Nutzbarkeit von SUPPLE im Zusammenhang mit diesen Geräten wurde besonderer Wert gelegt.

Im Unterschied zu dem in dieser Arbeit vorgestellten Werkzeug richtet sich SUPPLE ausschließlich an Experten. Für diese kann es bei der Erstellung von grafischen Benutzeroberflächen zur Steuerung von Geräten oder Diensten nützlich sein. Allerdings trägt es nur geringfügig zur Kostensenkung bei, da ein Experteneinsatz nach wie vor notwendig ist. Auf eine Wiederverwendbarkeit der Spezifikationen und Modelle wurde nicht explizit geachtet, so dass auch hier keine Kosten eingespart werden können. Vielmehr verstärkt sich der Effekt, dass Dienste für jedes eHome erneut geschrieben werden müssen, da von spezifischen Geräten nicht abstrahiert werden kann.

8.3.6 ECT

Das *Equip Component Toolkit* (ECT) [GT04] zielt wie die vorliegende Arbeit ebenfalls auf eine Kostensenkung bei der Installation von eHome-Umgebungen. Zu diesem Zweck bietet es Unterstützung bei der Dienstentwicklung, welche als Zusammenschaltung heterogener Hard- und Software-Komponenten betrachtet wird. Ziel dieses Werkzeuges ist eine Vereinfachung dieses Verschaltungsprozesses, so dass er auch durch Endanwender, also den eHome-Bewohner durchgeführt werden kann.

ECT etabliert verschiedene Abstraktionsschichten, welche sich in ihrer Komplexität unterscheiden und sich daher an unterschiedliche Benutzer richten.

- Kernimplementation
- Komponentenprogrammierung
- Komponentenkomposition

Auf unterster Ebene befindet sich die Kernimplementation, die lediglich für versierte und erfahrene Experten von Interesse ist. Diese beschäftigt sich mit den internen Details der betrachteten Umgebung. Es handelt sich folglich um eine Hardware-nahe Abstraktionsschicht, welche zu den auf unterster Ebene modellierten Unterdiensten des hier präsentierten Ansatzes korrespondieren.

Die Ebene der Komponentenprogrammierung ist ebenfalls nur für Software-Entwickler von Interesse, die jedoch nicht über ein so tiefes Verständnis verfügen müssen, wie die Experten, die sich mit der Kernimplementation beschäftigen. Diese Schicht entspricht im eHomeConfigurator einer Dienstkomposition.

Auf höchster Ebene, der Komponentenkomposition, kann auch ein Laie die zuvor programmierten Komponenten mit Hilfe einer grafischen Oberfläche zu einer eHome-Installation zusammenfügen.

Das Equip Component Toolkit weist viele Parallelen zu dem in dieser Arbeit vorgestellten eHomeConfigurator auf, insbesondere wird auch die Komposition von Diensten unterstützt. Im Zentrum einer solchen Komposition stehen bei ECT jedoch keine Funktionen, sondern sogenannte *Komponenten*, bei welchen es sich auf unterster Hierarchieebene um Geräte handelt. Darüber hinaus wird in Bezug auf die betrachteten Geräte keine explizite Behandlung von Kardinalitäten angeboten. Während die Architektur von ECT auf drei Hierarchieebenen festgelegt ist, können mit Hilfe des eHomeConfigurators beliebig viele Schichten von Diensten erzeugt werden.

Im direkten Vergleich bietet der Ansatz im Rahmen des eHomeConfigurators ein breiteres und flexibleres Spektrum an Unterstützung bei der Dienstentwicklung. Der an den Benutzer gestellte Anspruch bei der Konfigurierung im eHomeConfigurator ist ebenfalls geringer, da er sich mit den Komponenten, also den Unterdiensten, nicht auskennen muss. Er wählt lediglich anhand einer Liste die zu installierenden Dienste aus und folgt den Anweisungen des Assistenten.

8.3.7 CAMP

Bei CAMP [THA04] handelt es sich um ein Werkzeug, welches Benutzer ohne jegliche Programmiererfahrung dazu befähigen soll, eigene Dienste zu erstellen. Zu diesem Zweck wurde eine spezielle grafische Benutzeroberfläche erstellt, auf der der Anwender vorgegebenen Worten zu Sätzen zusammenbaut, die das Verhalten eines Dienstes beschreiben oder neue Worte definieren. Die vorgegebenen Wörter sind in verschiedene Klassen unterteilt, welche größtenteils mit einem Fragewort korrespondieren.

- **Wer:** ich, jeder, niemand, Kinder
- **Was:** Bilder, Audio, Video
- **Wo:** Küche, Bad, Schlafzimmer, überall, nirgendwo
- **Wann:** immer, nie, vor, nach, während
- **Allgemein:** aufnehmen, abspielen, ansehen, Lautsprecher, 18, 20

Zur Definition des Wortes *Abendessen* kann man aus den vorgegebenen Wörtern den Satz *"In der Küche nach 18 Uhr und vor 20 Uhr."* bilden. Auf ähnliche Art und Weise lassen

sich Dienste beschreiben. *"Während des Abendessens dürfen Kinder nie ein Video abspielen."* definiert beispielsweise einen Dienst, der den Kindern im Haus das Ansehen eines Videos während des Abendessens verbietet.

CAMP benutzt einen Parsing-Mechanismus, um die Sätze zu zerlegen und zu analysieren, welches Verhalten damit beschrieben wird. Es werden Redundanz, Konflikte und Zweideutigkeiten erkannt und auf unterschiedliche Art und Weise behandelt. Redundante Information wird verworfen, während zur Auflösung von Konflikten derjenigen Definition Vorrang gegeben wird, die auf einer höheren Ebene vom Benutzer eingegeben worden ist. *"Während des Abendessens vor 18 Uhr und nach 20 Uhr..."* löst einen Konflikt aus, da die Definition von Abendessen eine Uhrzeit zwischen 18 und 20 Uhr festlegt. In diesem Fall hat die auf einer höheren Ebene gelegene Beschreibung *"vor 18 Uhr und nach 20 Uhr"* Vorrang. Zweideutigkeit und fehlende Parameter werden durch das Setzen auf Standardwerte aufgelöst.

Der von CAMP verfolgte Ansatz unterscheidet sich von dem des eHomeConfigurators. Hier werden Dienste nicht vom Endbenutzer erstellt, sondern von einem Fachmann. Der Anwender wählt lediglich eine Reihe von Diensten aus und wird durch den Installationsprozess geleitet.

CAMPs Ausdrucksstärke stößt relativ schnell an ihre Grenzen. Zur vollständigen Beschreibung aller Dienste einer realen eHome-Umgebung, dass heißt inklusive des Ausschlusses verschiedener Räume und der gleichzeitigen Benutzung mehrerer Geräte, wird die Formulierung einer großen Menge von komplizierten Sätzen notwendig. Darüber hinaus ist die Verwendung der erstellten Sätze lediglich in einer bestimmten Umgebung möglich. Nur die Definitionen einiger Wörter können in einem anderen eHome wiederverwertet werden.

8.3.8 Konfigurationsmanagement und Deployment in der Software-Technik

Dieser Abschnitt beschäftigt sich mit Arbeiten, die auch die Begriffe der Konfiguration und des Deployments verwenden.

Konfigurationsmanagement spielt eine wichtige Rolle im Bereich der Softwaretechnik. Es repräsentiert alle Bereiche angefangen von der Varianten-Kontrolle über das Change-Management, Build-Management bis zum Versions- und Abhängigkeits-Management. Ein Software Produkt ist kein einheitliches Objekt, vielmehr besteht es aus einer Vielzahl verschiedener Software-Artefakte, wie Anforderungsspezifikation, Design-Dokumentation, Module, Testfälle, Benutzerhandbücher und Projekt-Pläne. Um Elemente, die zu einem Software-Produkt gehören zu definieren, werden deren Zusammenhänge durch eine Konfiguration [Bal97] beschrieben. Ein Dokument, welches diese Beschreibung enthält wird Konfigurationsdokument genannt. Die Verwaltung und Dokumentation dieser Dokumente wird *Configuration Management* genannt.

Die hier auftretenden Abhängigkeiten sind nicht nur hierarchisch, sondern bilden beliebige Graphen. Während eines der Design-Ziele sein sollte, die Verästelung in einem solchen Graphen möglichst gering zu halten, kann dies in der Praxis nicht immer erreicht werden.

Deshalb ist es ein adäquates Mittel, Abhängigkeiten, Versionen und Varianten mit Graphen zu beschreiben [WC98]. Der Konfigurations-Versionen-Graph besteht aus Ver-

sionen, welche durch Nachfolgerelationen miteinander verknüpft sind. Versionen werden sowohl für Dokumente als auch für Konfigurationen verwaltet. Der Konfigurationsgraph repräsentiert eine Momentaufnahme einer Menge von abhängigen Komponenten. Diese besteht also aus Komponenten-Versionen und deren Abhängigkeiten. Der *Konfigurationsobjektgraph* repräsentiert versionsunabhängige Strukturinfomationen. Er besteht aus Dokumenten (oder Subkonfigurationen), welche miteinander verbunden sind. Diese Verbindungen beschreiben ihre Abhängigkeiten. Für jede Komponentenversion (Versions-Abhängigkeit) in einem Konfigurationsgraphen muss ein korrespondierendes Komponentenobjekt (Objekt-Abhängigkeit) in dem Konfigurationsobjektgraphen existieren. Der Konfigurationsobjektgraph gibt eine strukturelle Übersicht und hilft bei der Verwaltung von Konfigurationsfamilien. Er besteht aus einem unveränderlichen Teil, der in allen Versionen enthalten ist, und einem veränderlichen Teil, dessen Elemente nur in einigen Versionen enthalten sind [Wes99].

Das Konfigurationsproblem lässt sich im Allgemeinen nur schwer lösen. In einigen spezifischen Problemdomänen kann dieses Problem durch Werkzeuge unterstützt werden. Die notwendige Werkzeugintegration [SB93] hat sich als schwierig herausgestellt. Wie in [Wes99] zu sehen, gibt es eine Menge von Werkzeugen, die unterschiedliche Gebiete des Konfigurationsmanagements unterstützen.

Einige anspruchsvolle Werkzeuge (zum Beispiel: PROGRES [Sch91, SWZ99], Agg [ERT99] oder Fujaba [Zün99]) für diese Unterstützung basieren auf Graph-Ersetzungssystemen[2] und bieten eine natürliche Repräsentation für Management-Aufgaben.

Neben dem Konfigurationsmanagement spielt auch das Gebiet des Software Deployments eine große Rolle im Bereich des Software-Engineerings. Üblicherweise versteht man unter Konfigurationsmanagement die Disziplin, die Evolution eines Softwareproduktes während seiner Entwicklungsphase unterstützt. Software Deployment ist die Disziplin, die die Evolution eines Software Produktes nach der Entwicklung unterstützt [Hoe01]. Das heißt, dass also die Aufgaben der Installation, des Updatens, der Rekonfiguration und der Adaption der Software beim Kunden darunter fallen. [HHW99].

Die hier vorliegende Arbeit berührt sowohl den Bereich des Konfigurationsmanagements als den des Deployments. Versionsverwaltung wird allerdings hier vernachlässigt.

[2] Für weitere Informationen zu Graphgrammatiken siehe [EPS73] .

Kapitel 9

Zusammenfassung und Ausblick

Wie in der Zielsetzung (Abschnitt 1.5) gefordert, hat die Arbeit in vier Schritten einen neuen Zugang zur Entwicklung von eHome-Systemen gegeben.

1. Der Entwicklungsprozess von eHome-Systemen wurde so umgestaltet, dass er in einem vielfältigen Wohnungsmarkt ohne eine Kostenexplosion eingesetzt werden kann.
2. Ein eHome-Modell zur Beschreibung aller Zustände der Konfiguration eines eHome-Systems von seiner Konzeption über den Betrieb bis zur Deinstallation wurde konstruiert.
3. Die Erstellung sogenannter Erweiterungsdienste wird durch das Modell und die Funktionalitäten-basierte Komposition unterstützt.
4. Die in dieser Arbeit entwickelte Werkzeugsuite eHomeConfigurator unterstützt die Phasen Spezifikation, Konfigurierung und Deployment eines eHome-Systems.

9.1 Synopse

Für eHome-Systeme gibt es bereits zahlreiche Forschungs- und Hobbyprojekte, sowie einige Spezialanfertigungen mit erfolgreichen Umsetzungen. Die dieser Arbeit zugrunde liegende Annahme besteht allerdings darin, dass diese Vorgehensweise in eine Sackgasse führt, da die hier bei jedem neuen Projekt für die Softwareentwicklung anfallenden Kosten für einen Massenmarkt immer zu hoch sein werden. Deswegen schlägt diese Arbeit einen Weg vor, den Aufwand und somit auch die Kosten, die diese Softwareentwicklung verursacht, so zu verringern, dass insbesondere eine einmal entwickelte eHome-Lösung in unterschiedlichen Zielumgebungen eingesetzt werden kann. Das heißt, dass zwar initial Kosten entstehen, diese aber sinken, je mehr Zielsysteme mit dieser Lösung eingerichtet werden. Es müssen so gut wie keine Anpassung mehr im Sinne von Softwareentwicklung sondern nur noch im Sinne von einer teilweise automatisierten Konfigurierung durchgeführt werden. Diese Konfigurierung ist so gestaltet, dass sie auch von einem normalen Benutzter, der den Umgang mit Computern gewöhnt ist, durchgeführt werden kann.

Wegen des Entwicklungsaufwandes, der nur mit Expertenhilfe zu erledigen ist, lohnen sich automatisierte Gebäudelösungen bisher nur für große Bürokomplexe oder für Massenanfertigungen ein und desselben Haustyps mit denselben Diensten. Um das große Angebot unterschiedlicher Umgebungen und Kundenwünsche in Privathäusern und -wohnungen bedienen zu können, eignet sich die bisherige Vorgehensweise nicht.

Um zu zeigen, dass insbesondere die Softwareentwicklung effizienter möglich ist, wurde in dieser Arbeit anhand von sechs Diensten und drei Demonstratoren dieser Prozess untersucht und optimiert. Diese Dienste beinhalten unter anderem zwei komplexe Erweiterungsdienste. Sie vereinen Funktionalitäten aus unterschiedlichen Anwendungsdomänen – wie Unterhaltung und Sicherheit – zu neuen Funktionalitäten, die nur aus der Kombination dieser Anwendungsdomänen entstehen können. Diese Kombinationsmöglichkeit ist bisher am Markt nur in Ansätzen vorhanden.

Um den Prozess neu zu strukturieren wurde der eHome-SCD-Prozess (Spezifizierungs-, Konfigurierungs- und Deployment-Prozess) entworfen. Dessen Idee ist es, eine iterative Kette prozeduraler Techniken zu etablieren, die die Erstellung eines eHome-Systems so weit wie möglich automatisiert. Damit wird es möglich, Dienste a-priori komponentenbasiert zu entwickeln und zu spezifizieren und dann für unterschiedliche Umgebungen zusammenzustellen und zu konfigurieren. Das heißt, dass die Transformation einer normalen Wohnung in ein eHome nach diesem Prozess nur noch initial, also beim ersten Einsatz, die Entwicklung von Dienstsoftware erfordert. Diese kann durch den Dienst-Provider im Vorfeld entwickelt werden. Anschließend ist beim Kunden nur die Konfigurierung und das Deployment erforderlich.

Durch die Konstruktion eines auf den Lebenszyklus von Diensten in eHome-Systemen zugeschnittenen eHome-Modells wird sowohl die teilweise automatisierte Konfigurierung als auch die Dienstentwicklung unterstützt. Zur Realisierung der angesprochenen Werkzeugkette wurde die Werkzeug-Suite eHomeConfigurator implementiert. Sie stellt die Werkzeuge zur Spezifikation, zur Konfigurierung, zum Deployment und die Methoden zum Zugriff auf eine eHome-Modell-Instanz, die Konfiguration, bereit.

Durch den Einsatz von Funktionalitäten-basierter Komposition kann ein Großteil der Auflösung der Dienstabhängigkeiten und der Zusammenstellung der Komponenten vom Konfigurierungs-Werkzeug automatisch durchgeführt werden. Eine vollkommene Automatisierung wird nicht erreicht, da der Kunde bei diesem Ansatz zu Beginn Dienste auswählen muss und auch bei gleichwertigen Lösungen entscheiden muss, welche gewählt wird. Weiterhin müssen Parameter, die das Werkzeug nicht automatisch ermitteln kann, vom Kunden angegeben werden.

Anhand der Beispielumgebungen und der untersuchten Dienste war es möglich zu verifizieren, dass die eHomeConfigurator-Werkzeugsuite die eingeführten Konzepte umsetzt und – zumindest für die in dem Rahmen dieser Arbeit möglichen Dienstauswahlen – in allen betrachteten Umgebungen eine Vereinfachung des Entwicklungsprozesses ermöglicht, da ab der zweiten Umgebung nur noch die manuelle Umgebungsspezifikation und Parametrisierung nötig ist. Anhand eines Vergleichs der eHomeConfigurator-gestützten Entwicklung mit komponentenbasierter Entwicklung und manueller Konfigurierung konnte eine deutliche Verringerung des Aufwandes bei der Codeentwicklung nachgewiesen werden.

Somit bietet diese Arbeit eine Möglichkeit der Vereinfachung und Beschleunigung des Entwicklungsprozesses für eHome-Systeme. Der vorgestellte Ansatz vermeidet wie-

derholt geschriebenen Code und bietet eine größere Wiederverwendungsmöglichkeit von Code als bisherige Lösungen. Die vorgestellte Lösung wird sich auf Grund der vorhanden Komplexität bei den Beispieldiensten und Umgebungen auf andere Umgebung erweitern lassen.

Die als Opensource-Projekt angelegte eHomeConfigurator-Werkzeugsuite wird aktiv weiterentwickelt. Sie stellt die Grundlage von bisher zwei weiteren Dissertationen dar und ist deshalb ein sehr lebendiges Projekt.

9.2 Ausblick

Die hier vorgestellten Konzepte und Werkzeuge bieten die Basis für eine vereinfachte Entwicklung von eHome-Systemen, die dann mit erheblich geringeren Kosten am Markt angeboten werden können. Es sind aber viele Möglichkeiten der Weiterentwicklung noch nicht ausgeschöpft. Deshalb sollen hier einige mögliche Folgeentwicklungen und Projekte vorgestellt werden:

Virtueller Demonstrator: Um Dienste zu testen, die Hardware voraussetzen, die nicht vorhanden ist, bietet sich die Simulation eines eHomes mit entsprechender Hardware an. Dies ist zwar nicht so überzeugend wie ein in echter Hardware realisierter Demonstrator oder gar ein wirkliches Haus, bietet aber die Möglichkeit auch die anderen in Abschnitt 2.1.2 vorgestellten Dienste zu implementieren und zu testen. So könnte die Einsatzfähigkeit der eHomeConfigurator-Werkzeugsuite auch für diese anderen Dienste verifiziert werden. Ein erster Prototyp eines solchen Simulators bzw. virtuellen Demonstrators wurde in den letzten Monaten bereits in einem Praktikum der eHome-Gruppe an der RWTH-Aachen [RWT] unter der Leitung von Ibrahim Armac und Daniel Retkowitz entwickelt. Dieser arbeitet bisher mit einer halb-dreidimensionalen Aufsicht für die Darstellung. Hier würde eventuell der Übergang zur komplett drei-dimensionalen Darstellung einen noch besseren Eindruck vermitteln. Für ein Beispiel eines Hotelzimmers mit Bad in diesem Demonstrator siehe Abbildung 9.1.

Modellerweiterungen: Bisher wurden im eHome-Modell nicht die tatsächlichen *Verbindungen* und *Leitungen* zwischen den einzelnen physischen Geräten modelliert. So könnten dann auch die verschiedenen verwendeten Bussysteme berücksichtigt werden und damit genauere Kostenabschätzungen durchgeführt werden.

Weiterhin werden die relativ neu eingeführten *ValueHolder* für States und Attribute, die außer Strings auch beliebig getypte Daten enthalten können, bisher von den Diensten kaum genutzt (siehe Abschnitt 5.2). Diese Erweiterung sollte ausgeschöpft und evaluiert werden.

Dienstkategorien: Bisher werden Dienstbeschreibungen bei der Spezifikation in eine unsortierte Liste eingefügt. Hierarchische Dienstkategorien würden auf Dauer, insbesondere bei einem wachsenden Dienstangebot, die Übersicht bei der Verwaltung und Auswahl der Dienste erhöhen.

Abbildung 9.1: Screenshot eines Hotelzimmers mit Bad in virtuellem Demonstrator (eHome-Simulator)

Statische Konfliktanalyse: Die Erfahrung der deklarativen Programmiersprachen aus [Kir05] kann für eine Resourcenbeschreibung in der Dienstespezifikation verwendet werden. Damit können die Ressourcen genauer als mit den Kardinalitäten beschrieben werden. Dies würde eventuell eine *statische Konfliktanalyse* ermöglichen.

Neue Graphendarstellung: Da die Programmierer der benutzten Graphenbibliothek JGraph [Com] die Lizenzpolitik geändert haben, sollte zur Vermeidung von Lizenzproblemen analog zu [FMP06] die Graphendarstellung auf die Graphendarstellung GEF [Con06] von Eclipse [Ecl03] übertragen werden.

Allgemeine generative komponentenbasierte Softwareentwicklung: Die hier eingeführte generative komponentenbasierte Softwareentwicklung eignet sich auch zur Entwicklung komponentenbasierter Software in anderen Bereichen. Eine Generalisierung des eHomeConfigurators zu einem Werkzeug für allgemeine generative komponentenbasierte Softwareentwicklung wäre ein mögliches Folgeprojekt. Für die ersten Ideen eines solchen Projektes, siehe die Webseite des ComponentConfigurators unter [Nor07].

Metrik für Dienstauswahl: Bisher muss bei Unterdiensten, die die gleiche Funktionalität anbieten noch derjenige ausgewählt werden, der verwendet werden soll. Dies kann über eine Kostenmetrik, die die beanspruchten Geräte einbezieht, automatisch geschehen. Zumindest könnte der Benutzer eine Strategie oder Metrik vorgeben, wie die Wahl zu erfolgen hat. Dies würde die noch nötige Interaktion mit dem Konfigurierungswerkzeug weiter reduzieren und somit die Konfigurierung weiter automatisieren.

Kontrakte: Durch die Benutzung von Kontrakten oder insbesondere auch parametrischer Kontrakte [RHH05, Reu01] könnten das Verhalten und auch die möglichen Zustände von Diensten genauer spezifiziert werden. Weiterhin könnte damit auch die Schnittstelle bei der Dienstkomposition berücksichtigt werden, was bisher nicht der Fall ist und somit eine genauere Absprache der Dienstentwickler erfordert. Dies wäre eine Erweiterung der Funktionalitäten-basierten Komposition, die eine genauere und zuverlässigere Dienstekomposition ermöglichen würde.

Schnittstellen zum semantic Web: Der ontologiebasierte Ansatz könnte wieder integriert werden und so eine Unterstützung für den Im- und Export von OWL-Beschreibungsdateien ermöglicht werden. Dies würde eventuell auch die Integration und Bereitstellung von Webservice-Techniken erlauben.

Künstliche Intelligenz: Der Einsatz von künstlicher Intelligenz zur Erkennung der Neigung von bestimmten Benutzern birgt neben ethischen Konflikten auch viele Konflikte im technischen Bereich. Insbesondere treten diese zu Tage, wenn mehr als ein Teilnehmer in der betrachteten Wohnung lebt. Ihr Einsatz führt oft zu unerwarteten und auch unerwünschten Ergebnissen. Wenn man die oft unerwünschte Assistenzunterstützung in bekannten Office-Anwendungen auf das eigene Heim überträgt, so kann man ein wenig die Auswirkung einer solchen fragwürdigen Unterstützung in den eigenen vier Wänden ermessen.

Von hier ist der Schritt zu missbräuchlicher Überwachung und dem Verlust der Privatssphäre im Sinne von Orwells "1984" oder Maschinen, die nur noch die Riten der Lust im Sinne von Huxleys "Brave new World" unterstützen, nicht mehr weit. Deshalb sollten Erweiterungen der eHome-Konfigurierung um Ansätze der künstlichen Intelligenz sehr kritisch betrachtet und weiterhin am Nutzen des Kunden orientiert entwickelt werden. Eventuell könnte sie zur Unterstützung der Strategieauswahl oder der Dienstsuche verwendet werden.

Es gibt Experimente, Neuronale Netze im eHome einzusetzen, so dass die Schnittstelle für den Einsatz solcher Systeme offen gehalten werden sollte (siehe [Moz98]).

Verteilung: Um seine persönlichen Vorlieben und Konfigurationseinstellungen in andere Umgebung mitnehmen zu können, müssen das verteilte Deployment und Konfigurationsprofile untersucht werden. Die Dissertation von Ibrahim Armac in der eHome-Gruppe der RWTH-Aachen [RWT] beschäftigt sich mit diesen Problemen.

Vereinfachte Spezifikatorgenerierung: Die Master-Arbeit von Priit Salumaa untersucht Ansätze, um die Spezifikatorgenerierung weiter zu unterstützen und den Verzicht auf die Activities-Beschreibungsdatei und die Translatorprogrammierung zu ermöglichen. Dies wurde an einigen Beispielen durchgeführt und könnte auf den gesamten eHomeConfigurator übertragen werden. Für weitere Informationen siehe [Sal05].

Mit der Führung des Projektes als Opensource-Projekt steht die Weiterführung jedem Interessierten offen. Die bisherigen Ergebnisse und auch die laufenden Arbeiten in der

eHome-Forschungsgruppe der RWTH-Aachen [RWT] bestätigen, dass der gewählte Ansatz eine gute Grundlage für weiter Forschungsarbeiten und Entwicklungen bildet. Es wäre erfreulich, wenn diese Arbeit weitere Forschungsgruppen oder Firmen ermutigen würde, in den eHomeConfigurator oder eines seiner Nachfolgeprojekte zu investieren.

9.3 Schlusswort

Die vier am Anfang der Arbeit gesteckten Ziele wurden erreicht. Mittels der hier vorgestellten Konzepte und Methoden ist es möglich, den Entwicklungsprozess für eHome-Systeme erheblich zu vereinfachen. Dies wurde an Beispielen demonstriert und durch Messungen bestätigt. Die entwickelte eHomeConfigurator-Werkzeugsuite bietet mit ihren Werkzeugen Unterstützung für die Spezifikation, für eine teilweise automatisierte Konfigurierung und für ein automatisches Deployment von eHome-Systemen.

Wie für viele andere neue Technologien gilt auch für das eHome, dass seine Einführung in den Massenmarkt zahlreiche neue Möglichkeiten eröffnet, aber zugleich aufgrund des neuen "Wohnparadigmas" neue Risiken bergen kann, die von Kritikern als Bedrohung empfunden werden können.

Dennoch eröffnet das modulare Konzept der "intelligenten" Verknüpfung zahlreicher und vielfältiger Dienste eine große Vielfalt neuer Funktionalitäten, die, geschickt eingesetzt, nicht nur das Leben und Wohnen angenehmer machen, sondern auch die Sicherheit erhöhen können. Es wird interessant sein zu beobachten, wie Neugier und spielerische Offenheit gerade der jüngeren Generationen, unterstützt durch Werkzeuge zur vereinfachten Konfigurierung wie der eHomeConfigurator-Werkzeugsuite, dem eHome zum Erfolg verhelfen werden.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[Aar03] AARTS, Emile ; RUYTER, Boris de (Hrsg.): *365 days' Ambient Intelligence research in HomeLab*. Eindhoven : Philips research, Neroc Eindhoven B.V., 2003. – 36 S. – (c) Royal Philips Electronics

[ACKM03] ALONSO, Gustavo ; CASATI, Fabio ; KUNO, Harumi ; MACHIRAJU, Vijay: *Web Services: Concepts, Architectures and Applications*. 1st edition. Springer, 2003. – 354 S. – ISBN 3540440089

[AGLH06] ARROYO, R. F. ; GEA, M. ; L., J. ; HAYA, P. A.: A Task-Driven Design Model for Collaborative AmI Systems. In: *Proceedings of the CAiSE'06 Workshops and Doctoral Consortium*, Presses universitaires de Namur, 2006. – ISBN 2–87037–525–5, S. 969–983

[Akh05] AKHOUNDI, Arash: *Verteiltes Automatisches Deployment von eHome-Konfigurationen*, Rheinisch-Westfälische Technische Hochschule Aachen, Diplomarbeit, März 2005

[Ami05] AMIGO: *Ambient Intelligence for the networked home environment*. `http://www.hitech-projects.com/euprojects/amigo/index.htm` (23.03.2007), 2005

[AMW99] ANDERSON, Mark ; MARTIN, Karl ; WALSH, Tom: *The Residential Gateway: Expanding the Horizons of Home Networking*. `http://citeseer.ist.psu.edu/419782.html`, 1999

[And] ANDREW ORLOWSKI: *The Register: Bread as a display device - we have pictures*. `http://www.theregister.co.uk/2001/06/04/bread_as_a_display_device/` (23.03.2007),

[Bal97] BALZERT, Helmut: *Lehrbuch der Softwaretechnik Band 2 - Software-Qualitätssicherung und Unternehmensmodellierung*. Spektrum Akademischer Verlag, 1997. – ISBN 3–8274–0065–1

[Bar05] BARDRAM, Jakob E.: The Java Context Awareness Framework (JCAF) – A Service Infrastructure and Programming Framework for Context-Aware Applications. In: GELLERSEN, Hans (Hrsg.) ; WANT, Roy (Hrsg.) ; SCHMIDT, Albrecht (Hrsg.): *Proc. of the 3rd International Conference*

on Pervasive Computing (Pervasive 2005). Munich, Germany, 2005 (LNCS)

[BC03a] BIEBER, Guy ; CARPENTER, Jeff: *Openwings Connector Service Specification Ver. 1.0 Final*. http://www.openwings.org (10.05.2004), 2003

[BC03b] BIEBER, Guy ; CRUMPTON, Kathleen: *Openwings Install Service Specification Ver. 1.0 Final*. http://www.openwings.org (10.05.2004), 2003

[Bec01] BECK, Dominique: *EIB evolution towards Konnex*. http://www.eib-scientific.de (9.8.2002), 2001

[Bei05] BEISHEIM HOLDING GMBH: *FutureLife – Das Haus der Zukunft*. http://www.futurelife.ch/ (21.03.2007), 2005

[BHM+04] BOOTH, David ; HAAS, Hugo ; MCCABE, Francis ; NEWCOMER, Eric ; CHAMPION, Michael ; FERRIS, Chris ; ORCHARD, David: Web Services Architecture / W3C (World Wide Web Consortium). 2004. – Forschungsbericht. – http://www.w3.org/TR/ws-arch/ (23.3.2007)

[Bie02] BIEBER, Guy: *Openwings Blueprints Ver. 1.1*. http://www.openwings.org/download/other/Openwings_Blueprints.pdf (23.03.2007), Aug 2002

[Bie03] BIEBER, Guy: *Openwings Context Specification Ver. 1.0 Final*. http://www.openwings.org (10.05.2004), 2003

[BJSW02] BÖHLEN, Boris ; JÄGER, Dirk ; SCHLEICHER, Ansgar ; WESTFECHTEL, Bernhard: UPGRADE: Building Interactive Tools for Visual Languages. In: CALLAOS, Nagib (Hrsg.) ; HERNANDEZ-ENCINAS, Luis (Hrsg.) ; YETIM, Fahri (Hrsg.): *Proceedings of the* 6^{th} *World Multiconference on Systemics, Cybernetics, and Informatics (SCI02)* Bd. I (Information Systems Development I). Orlando, Florida, USA : IIIS, Juli 2002. – ISBN 980–07–8150–1, S. 17–22

[BMK+00] BRUMITT, Barry ; MEYERS, Brian ; KRUMM, John ; KERN, Amanda ; SHAFER, Steven A.: EasyLiving: Technologies for Intelligent Environments. In: *HUC '00: Proceedings of the 2nd international symposium on Handheld and Ubiquitous Computing*, Springer, 2000. – ISBN 3–540–41093–7, S. 12–29

[BMW] BMW: *BMW ConnectedDrive*. http://www.connected-drive.de (21.03.2007),

[BNC03] BIEBER, Guy ; NELSON, Mark ; CHANG, Lon: *Openwings Interface Definition Specification Ver. 1.0 Final*. http://www.openwings.org (10.05.2004), 2003

[BPR00] BRÜGGE, Bernd ; PFLEGHAR, Ralf ; REICHER, Thomas: Internet Framework for Cooperative Buildings. In: *EIB Event*, 2000

[Bro97] BROOKS, R. A.: The Intelligent Room project. In: *CT '97: Proceedings of the 2nd International Conference on Cognitive Technology (CT '97)*, IEEE Computer Society, 1997. – ISBN 0–8186–8084–9, S. 271–278

[BT03] BIEBER, Guy ; THRASH, Brian: *Openwings Security Specification Ver. 1.0 Final*. `http://www.openwings.org` (10.05.2004), 2003

[Böc06] BÖCKERS, Philipp: *Automatisierte Tests für Zutrittskontrollsysteme*, Rheinisch-Westfälische Technische Hochschule Aachen, Diplomarbeit, August 2006

[Bör93] In: BÖRSTLER, J.: *IPSEN: An Integrated Environment to Support Development for and with Reuse*. Ellis-Horwood, 1993. – ISBN 0–13–063918–4, S. 134–140

[CB03a] CARPENTER, Jeff ; BIEBER, Guy: *Openwings Component Service Specification Ver. 1.0 Final*. `http://www.openwings.org` (10.05.2004), 2003

[CB03b] CARPENTER, Jeff ; BIEBER, Guy: *Openwings Overview Ver. 1.0 Final*. `http://www.openwings.org` (10.05.2004), 2003

[CE99] CZARNECKI, Krzysztof ; EISENECKER, Ulrich W.: Components and generative programming. (1999), S. 2–19. `http://dx.doi.org/http://doi.acm.org/10.1145/318773.318779`. – DOI http://doi.acm.org/10.1145/318773.318779. ISBN 3–540–66538–2

[Cer] CERAMI, Ethan: *Web Services Essentials*. 1st. O'Reilly. – 304 S. – ISBN 978–0–596–00224–4

[CG01] CHEN, Kirk ; GONG, Li: *Programming Open Service Gateways with Java Embedded Server Technology*. Addison-Wesley Professional, 2001. – 480 S. – ISBN 0–201–71102–8

[Cle05] CLEWARE-GMBH: *USB-IO16 Ein/Ausgabegerät mit 16 Kanälen über USB*. `http://www.cleware.de/p-usbio16.html` (21.03.2007), 2005

[Com] COMMUNITY, JGraph: *JGraph Swing Component*. `http://jgraph.com/jgraph.html` (23.03.2007),

[Com00] COMMUNITY, SVN: *SVN - Subversion*. `http://subversion.tigris.org/` (21.03.2007), Jun 2000

[Con06] CONSORTIUM, Eclipse: *GEF - Graphical Editing Framework*. `http://www.eclipse.org/gef/` (21.03.2007), 2006

[Deb] DEBIAN.ORG: *Debian Linux*. http://www.debian.org (22.03.2007),

[Den04] DENNIS REEDY: *Project Rio*. http://rio.java.net.org (20.03.2007), 2004

[DGZ04] DIETHELM, Ira ; GEIGER, Leif ; ZÜNDORF, Albert: Systematic Story Driven Modeling, a case study. In: *In Proc. of Workshop on Scenarios and state machines: models, algorithms, and tools (ICSE 2004)*. Edinburgh, Scotland, May 2004

[DJMZ05] DOSTAL, Wolfgang ; JECKLE, Mario ; MELZER, Ingo ; ZENGLER, Barbara: *Service-orientierte Architekturen mit Web Services - Konzepte - Standards - Praxis*. 1. Auflage. Heidelberg - München : Elsevier - Spektrum Akademischer Verlag, 2005. – ISBN 3827414571

[DY96] DIJK, Rolf Raven E. ; YGGE, Fredrik: SmartHome User Interface: Controlling Your Home Through the Internet. In: *ISES* 8 (1996)

[Ech02] ECHELON CORPORATION: *CEA-709.1-B: Control Network Protocol Specification*. Januar 2002

[Ecl03] ECLIPSE CONSORTIUM: *Eclipse*. http://www.eclipse.org (21.03.2007). http://www.eclipse.org. Version: 2003

[Edw00] EDWARDS, W. K.: *Core Jini*. Prentice Hall, 2000

[EF03] EBERHART, Andreas ; FISCHER, Stefan: *Web Services Grundlagen und praktische Umsetzung mit J2EE und .NET*. Carl Hanser Verlag, 2003. – ISBN 3–446–22530–7

[EPS73] EHRIG, H. ; PFENDER, M. ; SCHNEIDER, H. J.: Graph-grammars: an algebraic approach. In: *Proceedings IEEE Conf. on Automata and Switching Theory*, Iowa, 1973, S. 167–180

[ERT99] ERMEL, Claudia ; RUDOLF, Michael ; TAENTZER, Gabriele: The AGG approach: language and environment. In: *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools*. River Edge, NJ, USA : World Scientific Publishing Co., Inc., 1999. – ISBN 981–02–4020–1, S. 551–603

[FF04] FORMAN, Ira R. ; FORMAN, Nate: *Java Reflection in Action*. Manning Publications, 2004

[FFM] FFMPEG: *FFMPEG Multimedia System*. http://ffmpeg.mplayerhq.hu/ (23.03.2007),

[FGH+] FOGELL, Jennifer ; GRUBER, Olivier ; HABECK, Ted ; HARGRAVE, BJ ; KAEGI, Simon ; KRIENS, Peter ; LIPPERT, Martin ; MCAFFER, Jeff ; NIEFER, Andrew ; RAPICAULT, Pascal ; WATSON, Tom ; WEBSTER, Matthew ; YAMASAKI, Ikuo: *equinox*. http://www.eclipse.org/equinox/ (21.03.2007),

[FMP06] FUSS, Christian ; MOSLER, Christof ; PETTAU, Marcel: RePLEX: A Model-Based Reengineering Tool for PLEX Telecommunication Systems. In: *Proc. of 3rd International Workshop on Graph Based Tools (GraBaTs 2006)*, 2006. – to appear

[FNTZ98] FISCHER, Thorsten ; NIERE, Jörg ; TORUNSKI, Lars ; ZÜNDORF, Albert: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In: ENGELS, G. (Hrsg.) ; ROZENBERG, G. (Hrsg.): *Proc. of the 6^{th} International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, Springer, November 1998 (LNCS 1764), S. 296–309

[Fou04] FOUNDATION, Apache S.: *The Apache Ant Project*. `http://ant.apache.org` (23.03.2007), 2004

[Fra] *Frauenhofer Institut für Mikroelektronische Schaltungen und Systeme*. `http://www.ims.frauenhofer.de` (23.03.2007),

[GBV99] GANN, David ; BARLOW, James ; VENEBLES, Tim: *Digital Futures: Making Homes Smarter*. Chartered Institute of Housing, 1999. – 164 S.

[Gen] GENERAL DYNAMICS DECISION SYSTEMS: *Openwings*. `http://www.openwings.org` (23.03.2007). `http://www.openwings.org`

[GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[Gig00] *Gigaspace Technologies*. `http://www.gigaspaces.com` (23.03.2007), 2000

[GIR] GIRA: *Gira Homeserver*. `http://www.gira.de/homeserver` (21.03.2007),

[GNU91] GNU PROJECT: *GNU General Public Licence*. Version 2.1. `http://www.gnu.org/copyleft/gpl.html`, feb 1991

[GNU99] GNU PROJECT: *GNU Lesser General Public Licence*. `http://www.gnu.org/copyleft/lesser.html`. `http://www.gnu.org/copyleft/lesser.html`. Version: Version 2.1, Februar 1999

[Gon01] GONG, Li: JXTA: a network programming environment. In: *Internet Computing, IEEE* 5 (2001), May-June, Nr. 3, S. 88–95. `http://dx.doi.org/10.1109/4236.935182`. – DOI 10.1109/4236.935182

[GT04] GREENHALGH C., IZADI S., MATHRICK J., Humble J. ; TAYLOR, I.: ECT: A Toolkit to Support Rapid Construction of Ubicomp Environments. In: *Proc. of the Sixth International Conference on Ubiquitous Computing (UBICOMP'04)*, 2004

[GW04] GAJOS, Krzysztof ; WELD, Daniel S.: SUPPLE: automatically generating user interfaces. In: *IUI '04: Proceedings of the 9th international conference on Intelligent user interface*. New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–815–6, S. 93–100

[GZ02] GEIGER, Leif ; ZÜNDORF, Albert: Graph Based Debugging with Fujaba. In: *Electronic Notes in Theoretical Computer Science* 72 (2002), Nr. 2

[Hau] HAUSINFORMATIONSSYSTEME: *Homepage*. `http://www.hausinformationssysteme.de/` (23.03.2007),

[HAV01] HAVI INC.: *HAVi 1.1 Specification of the Home Audio/Video Interoperability Architecture*. Mai 2001

[HBS+02] HAPNER, Mark ; BURRIDGE, Rich ; SHARMA, Rahul ; FIALLI, Joseph ; STOUT, Kate: *Java Message Service*, April 2002. – `http://java.sun.com/products/jms` (23.03.2007)

[HHW99] HALL, Richard S. ; HEIMBIGNER, Dennis ; WOLF, Alexander L.: A cooperative approach to support software deployment using the software dock, 1999. – ISBN 1–58113–074–0, S. 174–183

[Hil02] HILES, Scott: *Linux X10 Universal Device Driver*. `http://wish.sourceforge.net/` (23.03.2007), 2002

[HLM05] HELAL, Abdelsalam ; LEE, Choonhwa ; MANN, William C.: Assistive Environments for Individuals with Special Needs. In: COOK, Diane J. (Hrsg.) ; DAS, Sajal K. (Hrsg.): *Smart Environments*. Department of Computer Science and Engineering, The University of Texas at Arlington, Box 19015, Arlington, TX 76019, USA, 2005 (Wiley-Interscience Series in Discrete Mathematics and Optimization), S. 361–383

[HO02] HROMKOVIC, Juraj ; OLIVA, Waldyr M.: *Algorithmics for Hard Problems*. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2002. – ISBN 3–540–44134–4

[Hoe01] HOEK, André van d.: Integrating Configuration Management and Software Deployment. In: *Proceedings of the Working Conference on Complex and Dynamic Systems Architecture (CDSA 2001)*, 2001

[Hol97] HOLLIDAY, C. R.: The Residential Gateway. In: *IEEE Spectrum* 34 (1997), Nr. 5, S. 29–31

[inH05] INHAUS DUISBURG: *Innovationszentrum Intelligentes Haus Duisburg*. `http://www.inhaus-zentrum.de` (23.03.2007), 2005

[IPB+06] IVOV, Emil ; PELOV, Alex ; BURCH, Brian ; MINKOV, Damian ; LORCHAT, Jean ; ANDRE, Martin ; KUNTZ, Romain ; STAMCHEVA, Yana: *SIP Communicator*. `http://www.sip-communicator.org/` (21.03.2007), 2006

[IST01a] *IST Advisory Group (ISTAG), European Commission*. `http://cordis.europa.eu/ist/istag.htm` (21.03.2007), 2001

[IST01b] IST ADVISORY GROUP (ISTAG), EUROPEAN COMMISSION: *Scenarios for Ambient Intelligence in 2010*. 2001. – Final Report. Compiled by K. Ducatel, et al. Feb-2001. EC. Brussels, 2001. ISBN 9289407352.

[JF02] JOHANSON, B. ; FOX, A.: The Event Heap: a coordination infrastructure for interactive workspaces. In: *Proc. Fourth IEEE Workshop on Mobile Computing Systems and Applications*, 2002, S. 83–93

[JFW02] JOHANSON, Brad ; FOX, Armando ; WINOGRAD, Terry: The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. In: *IEEE Pervasive Computing* 1 (2002), Nr. 2, S. 67–74. `http://dx.doi.org/http://dx.doi.org/10.1109/MPRV.2002.1012339`. – DOI http://dx.doi.org/10.1109/MPRV.2002.1012339. – ISSN 1536–1268

[JLY04] JIANG, Li ; LIU, Da-You ; YANG, Bo: Smart home research. In: *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on* Bd. 2, 2004, S. 659–663vol.2

[JN03] JIAO, Yu ; NORBISRATH, Ulrich: Dynamische Komponententechnologien für Limited Devices. In: K. IRMSCHER, K.-P. FÄHNRICH (Hrsg.): *13. ITG/GI-Fachtagung Kommunikation in verteilten Systemen, KiVs 2003*, VDI, 2003. – ISBN 3–8007–2753–6, S. 189–196

[JW03] JERONIMO, Michael ; WEAST, Jack: *UPnP Design by Example*. Intel Press, 2003. – 481 S. – ISBN 0–9717861–1–9

[Jäg00] JÄGER, Dirk: UPGRADE - A Framework for Graph-Based Visual Applications. In: NAGL, Manfred (Hrsg.) ; SCHÜRR, Andy (Hrsg.) ; MÜNCH, Manfred (Hrsg.): *Proceedings Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE'99)*. Kerkrade, The Netherlands : Springer, September 2000 (LNCS 1779). – ISBN 3–540–67658–9, S. 427–432

[KA91] KOTLER, Philip ; ARMSTRONG, Gary: *Principles of Marketing*. Englewood-Cliffs : Prentice-Hall, 1991. – ISBN 0–13–691247–8

[Kir05] KIRCHHOF, Michael: *Integrierte Low-Cost eHome-Systeme – Prozesse und Infrastrukturen*. Shaker, 2005. – 346 S. – ISBN 3–8322–4776–9

[Kli04] KLINKE, Markus: *Instanziierung von eHome-Diensten*, Rheinisch-Westfälische Technische Hochschule Aachen, Diplomarbeit, Mai 2004

[KLZ+05] KARASULU, Alex ; LORITSCH, Berin ; ZIEGELER, Carsten ; DEMURU, Matteo ; RODRIGUEZ, Enrique ; FURFARI, Francesco ; CERVANTES, Humberto ; LENZI, Stefano ; OFFERMANS, Marcel ; BERGMAN, Noel ; PAULS, Karl ; HALL, Richard ; WALLEZ, Sylvain ; BENNETT, Timothy

; LEE, Trustin ; WALKER, Rob ; FRENOT, Stephane ; SANTILLÁN, Manuel ; RUIZ, Jose L. ; DUEÑAS, Juan C. ; DONSEZ, Didier: *Apache Felix*. http://incubator.apache.org/felix/ (21.03.2007), aug 2005

[KNNZ99] KLEIN, Thomas ; NICKEL, Ulrich ; NIERE, Jörg ; ZÜNDORF, Albert: From UML to Java And Back Again / University of Paderborn. Paderborn, Germany, September 1999 (tr-ri-00-216). – Forschungsbericht

[Kno04] *Knopflerfish OSGi*. http://www.knopflerfish.org (23.03.2007), 2004

[KNS04] KIRCHHOF, Michael ; NORBISRATH, Ulrich ; SKRZYPCZYK, Christof: Towards Automatic Deployment in eHome Systems: Description Language and Tool Support. In: MEERSMAN, Robert (Hrsg.) ; TARI, Zahir (Hrsg.): *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, Proceedings, Part I*, Springer, 2004 (LNCS 3290). – ISBN 3–540–23663–5, S. 460–476

[KOA+99] KIDD, Cory D. ; ORR, Robert ; ABOWD, Gregory D. ; ATKESON, Christopher G. ; ESSA, Irfan A. ; MACINTYRE, Blair ; MYNATT, Elizabeth D. ; STARNER, Thad ; NEWSTETTER, Wendy: The Aware Home: A Living Laboratory for Ubiquitous Computing Research. In: *CoBuild '99: Proceedings of the Second International Workshop on Cooperative Buildings, Integrating Information, Organization, and Architecture*, Springer, 1999. – ISBN 3–540–66596–X, S. 191–198

[Kon00] KON, Fabio: *Automatic Configuration Of Component-Based Distributed Systems*, University of Illinois, Diss., 2000

[KP88] KRASNER, Glenn E. ; POPE, Stephen T.: A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. In: *J. Object Oriented Program.* 1 (1988), Nr. 3, S. 26–49. – ISSN 0896–8438

[Kre] KREMPL, Stefan: *T-Com-Haus: RFID und WLAN fürs Mood-Management*. http://www.heise.de/newsticker/meldung/56929 (23.03.2007),

[Kre04] KREIENBRINK, Ingo: *Klassifikation und Suchstrategien in eHome-Szenarien*, Rheinisch-Westfälische Technische Hochschule Aachen, Diplomarbeit, Juni 2004

[KS05] KIRCHHOF, Michael ; STINAUER, Philipp: Service Composition for eHome Systems: A Rule-based Approach. In: MOSTÉFAOUI, Soraya K. (Hrsg.) ; MAAMAR, Zakaria (Hrsg.): *Ubiquitous Computing - Proceedings of the 2^{nd} International Workshop on Ubiquitous Computing (IWUC 2005)*, INSTICC Press, 2005. – ISBN 972–8865–24–4, S. 28–38

[KWB03] KLEPPE, Anneke ; WARMER, Jos ; BAST, Wim: *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003

[Lau04] LAU, Oliver: Vom Technikpalast zum Wohnautomat. In: *c't Magazin für Computertechnik* 5 (2004), S. 114

[Lav00] LAVRSEN, Kenneth J.: *Motion - a software motion detector*. `http://motion.sourceforge.net/` (21.03.2007), 2000

[Lev03] LEVINSON, Meridith: All-in-One Appliance: THE REFRIGERATOR. In: *CIO Magazine - Trendlines* (2003)

[Mal05] MALIK, Adam: *Informationserfassung für automatisches Deployment von eHome-Systemen*, Rheinisch-Westfälische Technische Hochschule Aachen, Diplomarbeit, Februar 2005

[MBB02] MELLOR, Stephen ; BALCER, Marc ; BALCER, Marc J.: *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley, 2002

[McB02] MCBRIDE, Brian: Jena: a semantic Web toolkit. In: *Internet Computing, IEEE* 6 (2002), Nov.-Dec., Nr. 6, S. 55–59. `http://dx.doi.org/10.1109/MIC.2002.1067737`. – DOI 10.1109/MIC.2002.1067737

[McB04] MCBRIDE, Brian: *An Introduction to RDF and the Jena RDF API*. `http://jena.sourceforge.net/tutorial/RDF_API/` (23.03.2007), 2004

[Mic03] MICROSOFT CORPORATION: *Microsoft .NET Framework 1.1 Class Library Reference*. Microsoft Press, 2003. – ISBN 0–7356–1555–1

[Moz98] MOZER, Michael: The neural network house: An environment that adapts to its inhabitants. In: *Proceedings of the American Association for Artificial Intelligence Spring Symposium on Intelligent Environments*, AAAI Press, 1998. – ISBN 1–57735–047–2, S. 110–114

[Nag96] NAGL, Manfred (Hrsg.): *Building Tightly Integrated Software Development Environments: The IPSEN Approach*. Springer, 1996 (LNCS 1170). – ISBN 3–540–61985–2

[NARS06] NORBISRATH, Ulrich ; ARMAC, Ibrahim ; RETKOWITZ, Daniel ; SALUMAA, Priit: Modeling eHome Systems. In: *MPAC '06: Proceedings of the 4th international workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC 2006)*. New York, NY, USA : ACM Press, 2006. – ISBN 1–59593–421–9, S. 4

[New02] NEWCOMER, Eric: *Understanding Web Services - XML, WSDL, SOAP, and UDDI*. Addison-Wesley Professional, 2002. – ISBN 0–201–75081–3

[NFM00] NOY, N. ; FERGERSON, R. ; MUSEN, M.: *The knowledge model of Protege-2000: Combining interoperability and flexibility*. citeseer.ist.psu.edu/noy01knowledge.html. Version: 2000

[NM06a] NORBISRATH, Ulrich ; MOSLER, Christof: Component-Based Development for eHome Systems. In: *Proc. of the International Conference on Computational Methods in Science and Engineering (ICCMSE'06)*. Chania, Crete, Greece, Oct 2006 (Lecture Series of Computer and Computational Sciences (LSCCS)). – ISSN 1573–4196

[NM06b] NORBISRATH, Ulrich ; MOSLER, Christof: Functionality configuration for eHome systems. In: *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research (CASCON '06)*. New York, NY, USA : ACM Press, 2006, S. 8

[NNZ00] NICKEL, Ulrich ; NIERE, Jörg ; ZÜNDORF, Albert: The Fujaba Environment. In: *Proc. 22nd International Conference on Software Engineering (ICSE 2000)*. Limerick, Ireland : acm press, 2000, S. 742–745

[Nor07] NORBISRATH, Ulrich: *ComponentConfigurator*. http://componentconfigurator.ulno.net (21.03.2007), Jan 2007

[NSM05] NORBISRATH, Ulrich ; SALUMAA, Priit ; MALIK, Adam: *eHomeConfigurator*. http://ehomeconfig.ulno.net, 2005

[NSSK05] NORBISRATH, Ulrich ; SALUMAA, Priit ; SCHULTCHEN, Erhard ; KRAFT, Bodo: Fujaba based tool development for eHome systems. In: *Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004)* Bd. 127, Elsevier, 2005 (Electronic Notes in Theoretical Computer Science), S. 89–99

[Obj02] OBJECT MANAGEMENT GROUP, INC.: The Common Object Request Broker: Architecture and Specification, Revision 2.6.1. 2002. – Specification. – http://www.omg.org (14.6.2002)

[Obj04] OBJECT MANAGEMENT GROUP: *MDA Specifications*. http://www.omg.org/mda/specs.htm. Version: März 2004

[Ope] OPEN SERVICES GATEWAY INITIATIVE ALLIANCE: *Open Services Gateway Initiative Alliance Homepage*. http://www.osgi.org, Abruf: 11.07.2005

[Ope01] OPEN SERVICES GATEWAY INITIATIVE ALLIANCE: *OSGi Service Platform*. http://www.osgi.org/osgi_technology/download_specs.asp (21.03.2007), oct 2001. – Release 2

[Ope02] OPEN SERVICES GATEWAY INITIATIVE: *OSGi Service Platform Specification*. http://www.osgi.org/osgi_technology/download_specs.asp (21.3.2007), 2002

[Ope03a] OPEN SERVICES GATEWAY INITIATIVE ALLIANCE: *OSGi Service Platform*. http://www.osgi.org/osgi_technology/download_specs.asp (21.3.2007), apr 2003. – Release 3

[Ope03b] General Dynamics Decision Systems, INC.: *Openwings Tutorial*. http://www.openwings.org (10.05.2004), 2003

[OSG04] OSGI, Oscar: *Oscar Website*. http://oscar-osgi.sourceforge.net (21.03.2007), 2004

[Phi02] PHILIPS: *Homelab*. http://www.research.philips.com/technologies/misc/homelab/index.html ((23.03.2007)), 2002

[PP02] PATEL, D. ; PETERSON, I. D.: Hype and Reality in the Future Home. In: *BT Technology Journal* 20 (2002), Nr. 2, S. 106–115

[Proa] PROSYST SOFTWARE AG: *mBedded Server 5.x*. http://www.prosyst.de/solutions/osgi.html (03.03.2005),

[Prob] PROSYST SOFTWARE GMBH (Hrsg.): *Dokumentation ProSyst mBedded Server 5.1*. ProSyst Software GmbH

[PSM00] PRAGNELL, Mark ; SPENCE, Lorna ; MOORE, Roger: *The Market Potential for Smart Homes*. Joseph Rowntree Foundation, 2000. – 64 S.

[Ree02] REEDY, Dennis: *Rio Operational String*. http://rio.java.net.org (20.03.2007), 2002

[Ree03] REEDY, Dennis: Project Rio, A Dynamic Adaptive Network Architecture, Overview. (2003). – http://www.comp.lancs.ac.uk/computing/research/mpg/reflection/papers/rio-overview.pdf (23.03.2007)

[Reu01] REUSSNER, Ralf H.: *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Logos Verlag, Berlin, 2001. – ISBN 3–89722–783–5

[RFI] *Radio frequency identification (RFID) Technology*. http://www.aimglobal.org/technologies/rfid/ (30.1.2005),

[RHH05] REUSSNER, Ralf ; HAPPE, Jens ; HABEL, Annegret: Modelling Parametric Contracts and the State Space of Composite Components by Graph Grammars. In: *FASE*, Springer-Verlag, 2005, S. 80–95

[Rio03] *Rio Tutorial*. http://rio.java.net.org (20.03.2007), Bestandteil der Rio-Installation, 2003

[RN03] RUSSELL, Stuart ; NORVIG, Peter: *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 2003. – ISBN 0–13–080302–2

[Ros05] ROSOFF, Matt: Microsoft's Home Entertainment Strategy. In: *Directions on Microsoft* (2005), S. 6–35

[RWT] RWTH AACHEN UNIVERSITY OF TECHNOLOGY, DEPARTMENT OF COMPUTER SCIENCE 3: *eHome Group*. `http://www-i3.informatik.rwth-aachen.de/ehome` (23.03.2007),

[Sal05] SALUMAA, Priit: *Model Transformations in eHome Systems*, Rheinisch-Westfälische Technische Hochschule Aachen, Master Thesis, 2005

[SB93] SCHEFSTRÖM, D. (Hrsg.) ; BROEK, G. van d. (Hrsg.): *Tool Integration*. John Wiley & Sons, Inc., 1993 (Wiley Series in Software Based Systems)

[SBC03] SMITH, Michael ; BIEBER, Guy ; CARPENTER, Jeff: *Openwings Management Service Specification Ver. 1.0 Final*. `http://www.openwings.org` (10.05.2004), 2003

[Sch91] SCHÜRR, Andreas: *Operationelles Spezifizieren mit Programmierten Graphersetzungssystemen*, RWTH Aachen, Diss., 1991. – 461 S. – Dissertation RWTH Aachen

[Sch03] SCHNEIDER, Christian: *CASE Tool Unterstützung für die Delta-basierte Replikation und Versionierung komplexer Objektstrukturen*, Technische Universität Braunschweig, Diplomarbeit, April 2003

[Sch05a] SCHMIDT, Matthias: *Smarthouse*. `http://www.schmidt213.eu` (21.03.2007), 2005

[Sch05b] SCHWERDTNER, Tim: *Strategien zur Informationserfassung für eHome Systeme*, Rheinisch-Westfälische Technische Hochschule Aachen, Diplomarbeit, 2005

[SGG02] SILBERSCHATZ, Abraham ; GAGNE, Greg ; GALVIN, Peter B.: *Operating System Concepts*. 6th edition. New York : John Wiley & Sons, Inc., 2002. – ISBN 0–471–41743–2

[Skr04] SKRZYPCZYK, Christof: *Beschreibungssprache für eHome-Konfigurationen*, Rheinisch-Westfälische Technische Hochschule Aachen, Diplomarbeit, April 2004

[SRSAGM00] SANCHEZ-REILLO, Raul ; SANCHEZ-AVILA, Carmen ; GONZALEZ-MARCOS, Ana: Biometric Identification through Hand Geometry Measurements. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 22 (2000), Nr. 10, S. 1168–1171. `http://dx.doi.org/http://dx.doi.org/10.1109/34.879796`. – DOI http://dx.doi.org/10.1109/34.879796. – ISSN 0162–8828

[Suna] SUN MICROSYSTEMS, INC.: *Java 2 Platform, Micro Edition (J2ME) Homepage*. `http://java.sun.com/j2me/index.jsp`, Abruf: 10.07.2005

[Sunb] SUN MICROSYSTEMS, INC.: *Java Homepage*. `http://java.sun.com` (23.03.2007),

[Sunc] SUN MICROSYSTEMS, INC.: *The Community Resource for Jini Technology*. `http://www.jini.org` (23.03.2007),

[Sun97] SUN MICROSYSTEMS, INC.: JavaBeans Specification. 1997. – Specification. – `http://java.sun.com/products/javabeans/docs/spec.html` (19.5.2004)

[Sun98] SUN MICROSYSTEMS, INC.: *The Java Security Architecture for JDK 1.2*. `http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/securityspec.doc.html` (25.11.2003), 1998

[Sun99a] SUN MICROSYSTEMS: *Jini Architectural Overview*. `http://www.sun.com/software/jini/whitepapers/architecture.html` (2.3.2005), Januar 1999

[Sun99b] SUN MICROSYSTEMS, INC.: *JAR File Specification*. `http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html` (27.01.2004), 1999

[Sun03] SUN MICROSYSTEMS, INC.: *Enterprise Java Beans Specification 2.1 Final Release*. `http://java.sun.com/products/ejb/docs.html` (23.3.2007), November 2003

[Sun05] SUN MICROSYSTEMS, INC.: Java Remote Method Invocation (RMI) Specification. 2005. – Specification. – `http://java.sun.com/j2se/1.5.0/docs/guide/rmi/spec/rmiTOC.html` (23.03.2007)

[SWZ99] SCHÜRR, Andy ; WINTER, Andreas J. ; ZÜNDORF, Albert: PROGRES: Language and Environment. In: ROZENBERG, G. (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation* Bd. 2. Singapore : Applications, World Scientific, 1999

[Szy02] SZYPERSKI, Clemens: *Component Software*. 2nd edition. Addison-Wesley/ACM Press, 2002. – 608 S. – ISBN 0–201–74572–0

[T-C] T-COM: *T-Com Haus schließt seine Pforten - 18 000 Neugierige besuchten die Wohnwelt von morgen*. `http://www.t-com.de/SCMS_TCOM_de+DE_presse_T-Com+haus_downloads_downloadseite.page` (20.03.2007),

[T-C05] T-COM: *T-Com Haus*. `http://www.t-com-haus.de/` (26.9.2005), 2005

[TC] T-COM: *T-Com Haus Downloads*. `http://www.t-com.de/SCMS_TCOM_de+DE_presse_T-Com+haus_downloads_downloadseite.page` (20.03.2007),

[THA04] TRUONG, Khai N. ; HUANG, Elaine M. ; ABOWD, Gregory D.: CAMP: A Magnetic Poetry Interface for End-User Programming of Capture Applications for the Home. In: *Ubicomp*, 2004, S. 143–160

[The05] THE OSGI ALLIANCE: *OSGi Service Platform Core Specification*. http://www.osgi.org/osgi_technology/download_specs.asp (21.03.2007), aug 2005. – Release 4

[TS05] T-SYSTEMS: Wohnen in der digitalen Welt. In: *Best Practice - Ausgabe für den Mittelstand* (2005), S. 17–18

[Ulr05a] ULRICH NORBISRATH, PRIIT SALUMAA, ADAM MALIK: *eHome Specification, Configuration, and Deployment*. http://ubicomp.org/ubicomp2005/programs/demos.shtml (23.03.2007), 2005. – Demonstration Paper D15 on UbiComp2005

[Ulr05b] ULRICH NORBISRATH, PRIIT SALUMAA, ADAM MALIK: *Specification, Configuration, and Deployment in eHome Systems*. http://math.ut.ee/~peeter_l/seminar/eelmised/05k/ehome.html (23.03.2007), 2005

[Uni05] UNIVERSITY OF KASSEL: *eDobs*. Germany : http://www.se.e-technik.uni-kassel.de/se/index.php?edobs (21.03.2007), 2005

[UPn00] UPNP-FORUM: *UPnP Device Architecture*. Version: Juni 2000. http://www.upnp.org/download/UPnPDA10_20000613.htm, Abruf: 19.06.2005

[W3C] W3C (WORLD WIDE WEB CONSORTIUM): *HyperText Markup Language (HTML)*. http://www.w3.org/MarkUp (23.3.2007),

[Wal] WALLBANK, Nat: *A Requirements Analysis of Infrastructures for Ubiquitous Computing Environments*

[WC98] WESTFECHTEL, Bernhard ; CONRADI, Reidar: Version Models for Software Configuration Management. In: *ACM Computing Surveys* 30 (1998), Juni, Nr. 2

[Web03] WEB3D CONSORTIUM: *Virtual Reality Modeling Language (VRML)*. http://www.web3d.org/x3d/specifications/vrml/ (23.03.2007), 2003

[Wei91] WEISER, Mark: The computer for the 21st century. In: *Scientific American* 265 (1991), Nr. 3, S. 94–104

[Wes99] WESTFECHTEL, Bernhard: *Models and Tools for Managing Development Processes*. Springer, 1999 (LNCS 1646). – 418 S. – ISBN 3–540–66756–3

[WWB97] WAHRIG, Gerhard ; WAHRIG-BURFEIND, Renate: *Der kleine Wahrig - Wörterbuch der deutschen Sprache*. Gütersloh : Bertelsmann Lexikon Verlag GmbH, 1997

[X1004] X10: *Protocol Specification*. `http://www.marmitek.com/en/basisimages/x10_protocol.PDF` (23.03.2007), 2004

[Xim] XIMBIOT: *Concurrent Versions System*. `http://www.cvshome.org/` (23.03.2007),

[Xne] XNET-COMMUNICATIONS: *Haus der Zukunft - Futurelife*. `http://www.xdsnet.de/futurelife/` (21.03.2007),

[Zie04] ZIEGLER, Peter-Michael: Hightech-Wohnen als Beruf - Die Waschmaschine vom Auto aus starten. In: *c't Magazin für Computertechnik* 11 (2004), S. 46–47

[Zün99] ZÜNDORF, Albert: *FUJABA (From UML to Java and Back Again)*. `http://www.fujaba.de`, 1999

[Zün05] ZÜNDORF, Albert: Story driven modeling a practical guide to model driven software development. In: *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, 2005, S. 714–715

Index

Anhang A

Lebenslauf

Ulrich Norbisrath
Pikk 74-22
EE-50603 Tartu
Estonia (Estland)

Geburtsdatum	22.07.1973
Geburtsort	Aachen
Homepage	http://ulno.net
Gymnasium	Inda-Gymnasium Kornelimünster
Schulabschluss	Allgemeine Hochschulreife (Abitur)
Studium	01.10.1992 – 28.08.1995 Studium der Mathematik an der Rheinisch-Westfälischen Technischen Hochschule Aachen, Vertiefungsgebiet praktische Mathematik, Nebenfach Informatik 15.02.1995 – 14.07.2000 Studium der Informatik an der Rheinisch-Westfälischen Technischen Hochschule Aachen, Vertiefungsgebiet Softwaretechnik, Nebenfach Mathematik
Hochschulabschluss	Vordiplom Mathematik (28.08.1995) Diplom Informatik (14.07.2000)
Nebentätigkeit	01.07.1994 – 31.03.1999 Hilfswissenschaftler der aps GmbH, European Center of Mechatronics, Aachen
Berufstätigkeit	01.05.1999 – 28.02.2002 Selbständiger Einzelunternehmer mit der Firma WeSolv'IT 01.03.2002 – 29.02.2004 Doktorandenstipendium des Graduiertenkollegs Software für mobile Kommunikationssysteme 01.03.2004 – 31.10.2006 Fachgruppenreferent der Fachgruppe Informatik der Rheinisch-Westfälischen Technischen Hochschule Aachen