Projektgruppe

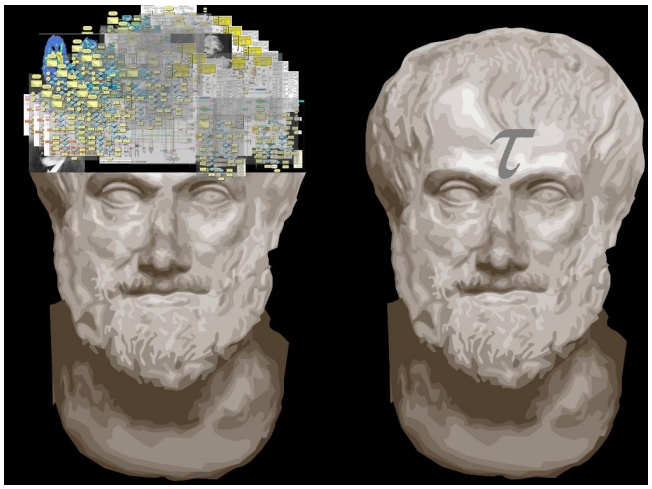
POETS

Process-Oriented Enterprise Transaction Systems

Wintersemester 2006/2007
Sommersemester 2007

Endbericht

Lehrstuhl XIV, FB Informatik



Dokumentation PG POETS

23. Oktober 2007

Endbericht

http://poets.cs.uni-dortmund.de
pg-poets@lists.cs.uni-dortmund.de

23. Oktober 2007

Vorwort

Die Projektgruppe 509 *Process-Oriented Enterprise Transaction Systems* (POETS) fand im Rahmen des Informatikstudiums am Lehrstuhls 14 des Fachbereichs Informatik an der Universität Dortmund in enger Zusammenarbeit mit dem Fraunhofer-Institut für Software- und Systemtechnik ISST in Dortmund während des Wintersemesters 2006/2007 und Sommersemesters 2007 statt.
Die Leitung der Projektgruppe wurde von Prof. Dr. Jakob Rehof übernommen und ist von Dipl.-Inform. Markus Doedt und Dipl.-Inform. Martin Sugioarto betreut worden.
Die Projektgruppe setzte sich aus den Studenten

Youssef Ben Cheikh,
Mohammed Bentaghlia,
Ansgar Flack,
Moritz Hofmann,
Sven-Torben Janus,
Eugen Krotov,
Sebastian Meinecke,
Stephan Schulte,
Dominik Spitzer,
Sebastian Steinbuß
und Markus Wandzik zusammen.

Projektgruppe 509 (POETS)

http://poets.cs.uni-dortmund.de/
pg-poets@lists.cs.uni-dortmund.de

Lehrstuhl 14

Universität Dortmund
Fachbereich Informatik
Lehrstuhl 14
44221 Dortmund
http://ls14-www.cs.uni-dortmund.de/

Fraunhofer-Institut für Software- und Systemtechnik ISST

Fraunhofer-Institut für Software- und Systemtechnik (ISST)
Institutsteil Dortmund
Emil-Figge-Straße 91
44227 Dortmund
http://www.do.isst.fraunhofer.de/

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Zusammenfassung

Dieser Endbericht der Projektgruppe 509 mit dem Titel *Process-Oriented Enterprise Transaction Systems (POETS)* des Lehrstuhls 14 des Fachbereichs Informatik der Universität Dortmund dient dazu, den bisherigen Verlauf und die Ergebnisse der Lehrveranstaltung aus dem Wintersemester 2006/2007 und dem Sommersemester 2007 vorzustellen.
Im ersten Semester wurden nach der Einarbeitung in das Thema, Konzepte und Methoden erarbeitet, um die Projektgruppenziele zu erreichen. Die Möglichkeiten der beteiligten Softwareplattformen wurden ergründet und die notwendigen Erweiterungen konzipiert und teilweise bereits implementiert.
Im zweiten Semester wurde die Implemetierung des Systems fortgeführt und beendet. Es wurden wichtige Erkenntnisse gesammelt über die Fähigkeiten des Systmes im Bereich Business Process Reengineering (BPR)und Continuous Process Improvement (CPI). Die Fähigkeiten des Systems im Bereich Logging und Monitoring wurden erweitert. Es wurde eine Softwareschnittstelle erstellt, die den Anschluss von Sensoren aus dem *SmarterWohnen* Umfeld an das System zulässt.
Das Projekt *SmarterWohnen*, das Grundlage unserer Arbeit ist, ist ein breites und interessantes Themenfeld. Die technischen Vorgaben und Möglichkeiten wurden mit den Methoden, welche die Informatik bietet, untersucht und strukturiert.
Das System wurde bewertet und mit ähnlichen Systemen verglichen. Außerdem wurde eine Bewertung der Projektorganisation erstellt.

Teil I
Motivation

1 Themenbeschreibung

Zur Unterstützung von Arbeitsabläufen in Unternehmen werden häufig Softwaresysteme eingesetzt. Dabei werden konkrete Arbeitsabläufe aus der Anwendungsdomäne in eine modellbasierte Sichtweise überführt. Die auf diese Weise als Geschäftsprozesse identifizierten Abläufe unterstützen und regeln in ihrer Implementierung den Workflow von Geschäftsvorfällen und Anwendungen.

Im Bereich der Modellierung und Implementierung stehen sich zwei scheinbar gegensätzliche Konzepte gegenüber: Die prozessorientierte Sichtweise sowie das Konzept der ECA-Regelsysteme. Die Prozesssicht unterstützt die Abbildung dieser Abläufe bereits bei der Modellbildung und bietet darüberhinaus eine strukturierte anwendungsbezogene Sicht auf die zu realisierende Aufgabe. ECA-Regelsysteme bieten hingegen die Möglichkeit, gerade bei stark durch Ereignisse gekennzeichneten Abläufen eine funktionelle Realisierung zu erhalten. Doch während der prozessbasierte Ansatz bei Prozessen, die durch eine komplexe Berücksichtigung von Ausnahmefällen gekennzeichnet sind, zu schlecht beherrschbaren Modellen und Realisierungen führen kann, stellen in ECA-Regelsystemen diese Ausnahmen Ereignisse dar, für die diese Regelsysteme offensichtlich das Mittel der Wahl sind. Werden ECA-Regelsysteme hingegen eingesetzt, um komplexe Arbeitsabläufe zu implementieren, so erhält der Entwickler ein durch unzählige Regeln gekennzeichnetes System. Das Verhalten dieses Systems kann in seiner Gesamtheit nur schwer nachvollzogen werden, da Methoden des Model Checking nicht anwendbar sind; außerdem bestimmt die Auswertungsreihenfolge der Regeln maßgeblich das beobachtete Verhalten. Bei solchen komplexen Abläufen ist scheinbar die Prozesssicht vorzuziehen. Es stellt sich also die Frage, für welche Arbeitsabläufe die Modellierung und Implementierung als Prozess geeignet ist und bei welchen Anwendungsfällen das ECA-Regelsystem die bessere Wahl darstellt.

SmarterWohnen ist ein Projekt, welches auf Grundlage der Ausstattung von Wohnräumen mit intelligenten Haustechnikkomponenten darauf aufsetzende Dienstleistungen bereitstellt. Ein informationslogistisches Softwaresystem verwendet in Wohnungen installierte Sensoren und Aktuatoren und bietet Bewohnern Dienste aus den Bereichen Sicherheit, Gesundheit und Komfort an. Beispielsweise werden Brände und Einbrüche erkannt und Bewohner und Dienstleister informiert. Arbeitsabläufe, die zur Bereitstellung solcher Dienste realisiert werden müssen, entsprechen in vielen Aspekten den bereits genannten Unternehmensabläufen. Somit stellt sich auch im Projekt *SmarterWohnen* die Frage nach der geeigneten Implementierung. Soll die Realisierung prozessbasiert durchgeführt werden oder sind ECA-Regelsysteme vorzuziehen? Eine weitere Möglichkeit ist eine Kombination beider Konzepte, bei der abhängig von der Natur des Arbeitsablaufs die Methode zur Realisierung verwendet wird, welche die meisten Vorteile bietet. Möglicherweise können Arbeitsabläufe in eine prozessbasierte und eine ECA-Regelbasierte Komponente aufgeteilt werden, um die optimale Lösung zu erhalten.

Das Fraunhofer-Institut für Software- und Systemtechnik in Dortmund entwickelt das o. g. informationslogistische Softwaresystem und sieht bei der Realisierung der Arbeitsabläufe sowohl den ECA-Regelbasierten Ansatz, als auch die prozessorientierte Sicht vor.

Im Rahmen der Projektgruppe POETS sollen zunächst die im Projekt *SmarterWohnen* definierten Arbeitsabläufe zur Bereitstellung der damit verbundenen Dienstleistungen analysiert und anschließend als Arbeitsprozess im *jABC* modelliert und implementiert werden. Das *jABC* ist ein vom Lehrstuhl 5 entwickeltes Modellierungswerkzeug, welches insbesondere Personen ohne Programmierkenntnisse die Möglichkeit bietet, Abläufe graphisch darzustellen und anschließend auszuführen. Die auf diese Weise implementierten Prozesse des *SmarterWohnen*-Projektes ersetzen somit die Realisierung auf der informationslogistischen Plattform des ISST. Das *jABC* ist weiterhin als Subsystem an die SmarterWohnen-Plattform anzubinden, um die Ablaufsteuerung der Prozesse erfolgreich auslagern zu können. Hierzu sind sowohl das *jABC* als auch die informationslogistische Plattform entsprechend zu erweitern und eine geeignete Kommunikationslösung zwischen den beiden Systemen zu implementieren.

Auf diese Weise wird ein Gesamtsystem geschaffen, auf dem die *SmarterWohnen*-Dienste sowohl in Form von Prozessen als auch als ECA-Regelsystem implementiert werden können. Für die Arbeitsabläufe kann somit die jeweils geeignete Implementierungsmethode evaluiert werden und allgemeine Aussagen über den Zusammenhang zwischen Arbeitsablauf einerseits und Implementierungsmethode andererseits abgeleitet werden.

2 Das Projekt SmarterWohnen

2.1 Überblick

Der Begriff *Intelligentes Wohnen* wurde vom Zentralverband Elektrotechnik- und Elektronikindustrie (ZVEI) eingeführt. Der Begriff fasst technische Lösungen im privaten Wohnbereich zusammen, durch die mittels Einsatz von technischen Systemen Komfort, Wirtschaftlichkeit, Flexibilität und Sicherheit geschaffen wird. Die Realisierung geschieht primär durch Vernetzung und zentrale Steuerung von Haustechnik wie beispielsweise Alarmanlagen, Heizungssystemen und Elektrohaushaltsgeräten.

Das innerhalb des Projektes *SmarterWohnen* eingesetzte Konzept *Smart Living* ist ein auf die Grundsätze des Intelligenten Wohnens aufsetzendes umfangreiches System zur Bereitstellung von smarten Diensten im Wohnbereich. Smarte Dienste sind im Vergleich zum Intelligenten Wohnen höherwertige Mehrwertdienste, die u. a. auf der sogenannten *Domotik*[1], also der intelligenten Haustechnik, aufsetzen. *Smart Living* ist ein aus dem Geschäftsfeld Informationslogistik hervorgegangenes Projekt des Fraunhofer-Institut für Software- und Systemtechnik (ISST) am Standort Dortmund. Dort ist es neben *Inhaus2*[2], *Smart-Wear* und *Smart Sport Solutions* eines von drei Projekten der Entwicklung so genannter smarter Objekte.

2.2 Motivation

In vielen alltäglichen Bereichen wie beispielsweise der Nutzung von Automobilen wird der Mensch mit rechnergestützen und informationstechnischen Komponenten konfrontiert, auf deren Nutzen nicht oder nur sehr schwer verzichtet werden kann. Beispielsweise der Einsatz einer elektronischen Zentralverriegelung ist in Automobilen mittlerweile zur Selbstverständlichkeit geworden. In den eigenen vier Wänden jedoch haben sich ähnliche

[1] integrierte Haustechnik; Wortschöpfung aus *domus* (lat. Haus) und Technik/Informatik
[2] vgl. [25]

Konzepte bisher nicht durchsetzen können. Der Bereich der Haustechnik wie z.B. elektrisch betriebene Jalousien oder Sicherheitssysteme gegen Einbruch und Diebstahl sind hauptsächlich gewerblichen und industriellen Objekten vorbehalten. Dennoch ist das Anwendungspotential für derartige Techniken im privaten Wohnraum groß und kann zur Realisierung einer Fülle von Services dienen und den Alltag erleichtern.

Insbesondere der demographische Wandel in Deutschland führt zu einer Veränderung der Anforderungen an den zur Verfügung stehenden Wohnraum. Die Bevölkerungszahl nimmt stetig ab und das Durchschnittsalter erhöht sich. Gemäß einer Studie des statistischen Bundesamtes werden im Jahre 2040 73 Personen im Rentenalter 100 Personen im erwerbsfähigen Alter gegenüberstehen. Die Austattung von Wohnraum mit entsprechender intelligenter Haustechnik wird somit nicht nur zur Notwendigkeit, sondern besitzt auch aus markttechnischen Gesichtspunkten eine besondere Attraktivität. Smarte Wohnräume bieten ihren Bewohnern nicht nur zusätzlichen Komfort, der den Lebensalltag erleichtert; zusätzlich können durch entsprechende Dienstleistungen Bedürfnisse in den Bereichen Sicherheit und Gesundheit befriedigt werden.

Insbesondere betreutes Wohnen und häusliche Pflege sind Anwendungsgebiete, die durch intelligente Domotik unterstützt werden können und auf diese Weise insgesamt die Attraktivität von Wohnräumen massiv erhöhen. Dies führt außerdem zur Steigerung des Marktwertes betroffener Wohnobjekte und ist somit auch aus wirtschaftlichen Aspekten für Investoren von Interesse. Somit ist die Entwicklung und Realisierung von Konzepten im Bereich der intelligenten Hausvernetzung in Kombination mit IT-gestützten Mehrwertdiensten ein zukunftsträchtiges und innovatives Geschäftsfeld der angewandten Forschung.

2.3 Realisierung

Das Fraunhofer ISST entwickelt und implementiert daher innerhalb des Pilotprojektes *SmarterWohnen* in Kooperation mit der Hattinger Wohnstätten eG (HWG) und weiteren Projektpartnern[3] Lösungen aus dem Forschungsbereich Smart Living. Es werden 185 von der HWG bereitgestellte Wohnungen einer Wohnsiedlung in der Hattinger Südstadt mit intelligenter Haustechnik ausgestattet, intelligente Mehrwertdienste entwickelt und geeignete Geschäftsmodelle realisiert und evaluiert. Kernkomponente dabei ist die vom ISST entwickelte Service-Plattform (ISST-interne Bezeichnung *IGM-Plattform*), auf der insbesondere die informationslogistischen Abläufe dieser Mehrwertdienste realisiert werden.

Aufsetzend auf die durch lokale Komponenten gebildete Hausinfrastruktur (vgl. Abschnitt 3.1) der Wohneinheiten werden informationslogistische Abläufe entwickelt und Bewohnern und Dienstleistern der Wohnanlagen zur Verfügung gestellt. Es entstehen somit komplexe höherschichtige Dienste aus den Anwendungsbereichen Komfort, Sicherheit und Gesundheit, die von den Bewohnern gegen Entgelt von der Hausverwaltung genutzt werden. Im folgenden Abschnitt werden diese drei Kernbereiche, aus denen Dienstleistungen erbracht werden, exemplarisch erläutert.

Komfort Durch die Existenz der zur Haustechnik gehörenden Kommunikationskomponenten wie der Bewohnerschnittstelle *Smart Living Manager* (vgl. Abschnitt 3.2) können so genannte Content-Dienste genutzt werden. Hierbei handelt es sich um die Bereitstellung von Informationen wie z. B. Fernsehprogramm, Bundesligaergebnisse oder

[3] Fraunhofer-Institut für mikroelektronische Schaltungen und Systeme (IMS), Berliner Institut für Sozialforschung (BIS), Trägerverein ZENIT e.V., Zentrum für Telematik im Gesundheitswesen GmbH

Energieverbrauchswerte, welche durch den Bewohner über das User-Interface abrufbar sind. Außerdem wird hierüber ein Menüservice[4] angeboten.

Sicherheit Durch die Erkennung von verschiedenen sicherheitsrelevanten Ereignissen wie Einbruch, Brand, austretendes Gas und Wasserschäden werden den Bewohnern komplexe Dienstleistungen aus dem Bereich Sicherheit angeboten. Dabei werden Gefahren durch die vorhandene Haustechnik detektiert und informationslogistische Geschäftsprozesse angestoßen, welche Bewohner und Dienstleister informieren und notwendige Abläufe steuern und kontrollieren.

Gesundheit Da das durch *SmarterWohnen* realisierte Angebot besonders für ältere Menschen attraktiv sein soll, werden außerdem Dienstleistungen aus dem Bereich Gesundheit angeboten. Die aus der Telemedizin stammende Messung, Erfassung und Bereitstellung von Vitalwerten in Zusammenarbeit mit dem „Dienstleister“ Hausarzt ist somit ein Beispiel für Mehrwertdienste aus dem Gesundheitsbereich.

Die hier nur grob beschriebenen smarten Dienste werden in Abschnitt 3 im Rahmen der Analysephase der zu Grunde liegenden *SmarterWohnen*-Prozesse näher erläutert.

Im folgenden Abschnitt 3 werden die für die Bereitstellung von smarten Diensten notwendigen Infrastrukturkomponenten erläutert. Insbesondere die Service-Plattform wird in Abschnitt 5 detailliert beschrieben.

[4] politische Bezeichnung für „Essen auf Rädern“

Teil II
Technischer Hintergrund

3 SmarterWohnen: Technisches Konzept

In diesem Kapitel wird das technische Konzept und die zu Grunde liegende Architektur des Projektes *SmarterWohnen* erläutert. Es werden in Abschnitt 3.1 die lokalen Komponenten der Hausinfrastruktur und in Kapitel 3.2 die Komponenten *Service-Gateway* und *Service-Plattform* beschrieben.

3.1 Lokale Komponenten und Vernetzung

Hausinfrastruktur Notwendige Voraussetzung zur Realisierung der in Abschnitt 2 beschriebenen smarten Dienste ist die Verfügbarkeit einer entsprechenden Hausinfrastruktur (Domotik). Daher ist eine Austattung des Objektes mit vernetzten *Sensoren* und *Aktuatoren* erforderlich. In Abhängigkeit von der Art der bereitzustellenden smarten Dienste sind geeignete technische Komponenten auszuwählen und zu installieren; desweiteren müssen unterschiedliche Vernetzungstechniken und -protokolle Berücksichtigung finden, die ggf. nebeneinander in einer heterogenen Umgebung koexistieren. Im Domotik-Umfeld kommen beispielsweise Steckdosensysteme, Lichtanlagen, Schließanlagen, Rauchmeldesysteme, Bewegungsmelder sowie Ortungssysteme und Kameras zum Einsatz.

Weitere Komponenten Smarte Komponenten aus den Bereichen der weißen und braunen Ware, also Waschmaschinen, Kühlschränke, Herde und Mikrowellen sowie Audio- und Video-Geräte gehen über die Definition von Domotik im deutschsprachigen Raum hinaus, sind allerdings zur Realisierung einiger Dienste erforderlich und stellen daher ggf. ebenfalls Komponenten der Haustechnik dar. Desweiteren ist der Einsatz von smarten Komponenten im Bereich der Energieversorgung wie beispielsweise Heizungs- und Lüftungssysteme erforderlich, sollen Services wie Verbrauchsmessung oder automatisierter Klimaregelung angeboten werden. Auch ist der Einsatz von medizinischen Komponenten notwendig, um Dienste aus dem Bereich Gesundheit und Telemedizin anbieten zu können. Die Spannweite von einsetzbaren lokalen Komponenten reicht von Personenwaagen über Blutdruck- und EKG-Messgeräten bis hin zu den sogenannten Wearables, also in die Kleidung integrierte eingebette Systeme.

Vernetzung Es besteht die Notwendigkeit, alle lokalen Komponenten der Haustechnik und Telemedizin mit Hilfe geeigneter Kommunikationstechniken zu vernetzen, damit auf diese durch das zentrale *Service-Gateway* zugegriffen werden kann. Dabei ist es nicht unbedingt erforderlich, dass alle Systeme untereinander vernetzt sind; jedoch muss das im folgenden Abschnitt 3.2 beschriebene Gateway über Zugriffsmöglichkeiten auf alle im Wohnobjekt vorhandenen smarten Komponenten verfügen.

Die baulichen Gegebenheiten im betroffenden Wohnobjekt grenzen teilweise die zur Verfügung stehenden Möglichkeiten im Bereich physikalischer und topologischer Techniken ein. So lassen sich die Systeme unterteilen in drahtlose Techniken wie WLAN[5], Wi-

[5] Wireless Local Area Network, Standard nach IEEE 802.11-Familie

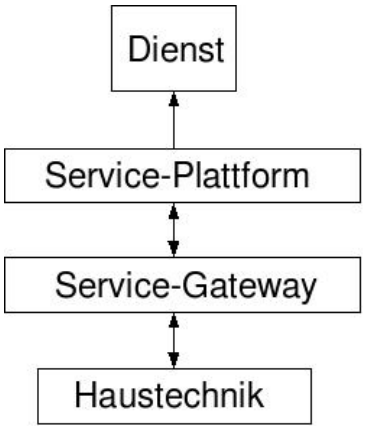


Abbildung 1: Architektur der technische Infrastruktur von *SmarterWohnen*

Max[6], Bluetooth[7] und Infrarot sowie drahtgebundene Lösungen wie EIB[8], PowerLine[9] und Ethernet. Wobei PowerLine auf eine in jedem Objekt vorhandene Stromversorgung aufsetzt und daher eine Sonderrolle spielt. Mit Hilfe des Standards LON[10] lassen sich sowohl drahtgebundene als auch drahtlose Lösungen realisieren, sodass er beiden genannten Kategorien zuzuordnen ist. Während bei der Austattung von Neubauten mit smarter Technik nahezu alle Vernetzungsmöglichkeiten offen stehen, sind bei nachträglicher Ausrüstung aufgrund verschiedener baulicher Gegebenheiten (z. B. Denkmalschutz) sowie unter betriebswirtschaftlichen Gesichtspunkten u. U. nur bestimmte Vernetzungsarten möglich; hierbei kommen meist drahtlose Techniken oder PowerLine zum Einsatz.

3.2 Service-Gateway und Service-Plattform

Die genannten Haustechniksysteme stellen die untere Schicht bei der Realisierung von smarten Diensten dar (vgl. Abbildung 1). Durch Vernetzung der Komponenten und der zentralen Steuerung von smarten Objekten einer Wohneinheit durch das *Service-Gateway* wird eine höher gelegene Abstraktionsschicht realisiert. Das *Service-Gateway* verfügt über die Kenntnis technischer Spezifikationen und Schnittstellen aller im Wohnobjekt vorhandenen Komponenten einschließlich der verwendeten Vernetzungsprotokolle. Es stellt nach Außen hin auf Seiten der *Service-Plattform* definierte Schnittstellen zur Steuerung der vorhandenen Domotik- und sonstiger lokalen Komponenten bereit. Somit greift die *Service-Plattform* über eine Abstraktionsebene gekapselt auf native Funktionen des smarten Wohnobjektes zu. Genaue Geräte-spezifische Kenntnisse über die Spezifikation der vorhandenen Komponenten ist daher auf Seiten der *Service-Plattform* nicht erforderlich.

Die *Service-Plattform* stellt weiterhin höherschichtige IT-Mehrwertdienste auf Basis der durch das *Service-Gateway* bereitgestellten Funktionalität zur Verfügung. Die bereitgestellten informationslogistischen Dienste aus den Bereichen Komfort, Sicherheit und Gesundheit werden auf der *Service-Plattform* implementiert. Dabei werden die Grundfunktionen der *Service-Gateways* sowie die Schnittstellen von Bewohnern und externen Dienstleistern an Informationsquellen und -senken verwendet. Die Kommunikation mit Bewohnern,

[6] Worldwide Interoperability for Microwave Access, Standard nach IEEE 802.16
[7] Drahtlose Funkvernetzung von Geräten über kurze Distanz, Standard nach IEEE 802.15.1
[8] European Installation Bus, Standard nach EN50090
[9] Datenübertragung über das Stromnetz
[10] Local Operating Network, Standard EN14908

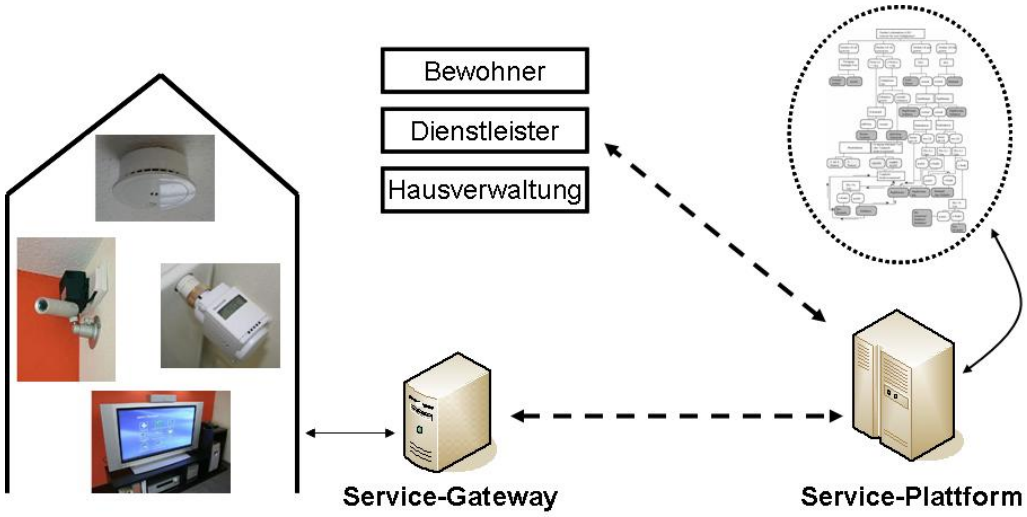


Abbildung 2: Schematische Darstellung der technische Infrastruktur von *SmarterWohnen*

Dienstleistern und Plattform-Administratoren erfolgt somit über unterschiedliche Kommunikationsmedien einschließlich der in Abschnitt 3.1 genannten Benutzerschnittstelle. Die verschiedenen Teilabläufe und durch das *Service-Gateway* gekapselten Funktionen werden auf der *Service-Plattform* zu smarten Diensten zusammengefasst und stehen den beteiligten Akteuren zur Verfügung.

Durch diese mehrschichtige Architektur wird gewährleistet, dass die vom ISST entwickelte Dienstleistungsplattform weitgehend unabhängig von lokal installierter Haustechnik entwickelt werden kann. Dies stellt nicht nur im Entwicklungsprozess, sondern im gesamten Lebenszyklus des Gesamtsystems einen entscheidenen Vorteil dar. Denn auf diese Weise kann auf Veränderungen im sich stetig weiterentwickelnden Bereich der Haustechnik, smarter Objekte sowie verwendeter Kommunikationsprotokolle auf Seiten des *Service-Gateways* reagiert werden, während die komplexere *Service-Plattform* durch ihre weitestgehend unabhängige Struktur davon unberührt bleibt. Außerdem kann die *Service-Plattform* in unterschiedlichen Lösungen und Umgebungen eingesetzt werden und ist in ihrer Gesamtheit nicht speziell auf ein bestimmtes Anwendungsszenario smarter Immobilienaustattung festgelegt.

Das Konzept ist in Abbildung 2 schematisch dargestellt. Auf die lokalen Komponenten auf der linken Seite der Abbildung greift die *Service-Plattform* über das *Service-Gateway* zu und bedient sich somit gekapselt der Haustechnikkomponenten. Weiterhin sind auf der Plattform smarte Dienste in Form von Prozessketten und Regelsystemen implementiert und werden von Bewohnern, Dienstleistern und der Hausverwaltung genutzt. Der Zugriff auf diese Dienste erfolgt dabei ausschließlich über die *Service-Plattform*. Das Gateway ist dabei für die Nutzer transparent und somit ein außerhalb der Wahrnehmung befindlicher Bestandteil der Infrastruktur.

Die IGM-Dienstleistungsplattform bzw. *Service-Plattform* wird momentan produktiv in einer eng auf das Smart Living-Konzept zugeschnittenen Implementierung betrieben. Diese wird nun schrittweise auf eine Instanz eines vom Fraunhofer ISST eigenentwickelten informationslogistischen Frameworks portiert werden, der *IGM-Plattform*. Dieses modular aus verschiedenen Subsystemen aufgebaute Gesamtsystem kommt ebenfalls bei der Olympiade im Jahr 2008 in Peking als *COMPASS* zum Einsatz. Diese Projektgruppe

wird ausschließlich mit letztgenannter Implementierung der *Service-Plattform* – in folgenden Abschnitten *IGM-Plattform* genannt – arbeiten. Eine detaillierte Beschreibung des Aufbaus und der Funktionsweise dieses Systems ist in Abschnitt 5 aufgeführt.

Schnittstellen und User-Interfaces Ein wichtiger Punkt, um die Akzeptanz von Bewohnern für smarte Umgebungen zu maximieren, ist der Einsatz von anwendungsfreundlichen, ergonomischen Schnittstellen zwischen System und Nutzer. Laut Angaben des Fraunhofer ISST waren 1999 96 % aller Haushalte in Deutschland mit einem Fernseher ausgestattet. Es liegt also nahe, dieses Medium als Hauptschnittstelle zur *Service-Plattform* zu verwenden, über die der Bewohner alle smarten Dienste steuern kann.

Die Schnittstelle zwischen Fernsehgerät und *Service-Plattform* wird dabei durch den *Smart Living Manager* realisiert. Dieses Gerät ähnelt im äußeren Erscheinungsbild einem Satelitenreceiver und ist über die interne Hausvernetzung mit der Haustechnik verbunden. Dabei stellt es eine Verbindung zur *Service-Plattform* her und bietet alle verfügbaren smarten Dienste kategorisiert über ein vom ISST entwickeltes Front-End an. Die Steuerung erfolgt menübasiert und ähnelt der von On-Screen-Menues bei TV-Zusatzgeräten wie Receivern oder Videorekordern. Über dieses Gerät steuert der Bewohner sämtliche verfügbaren Dienste, von weniger komplexen Services wie Beleuchtungseinstellungen und Heizungssteuerung bis hin zu IT-Mehrwertdiensten wie die Bedienung der Alarmanlage und Home-Delivery Services. Das *Service-Gateway* wird dabei transparent verwendet, d.h. der Bewohner nimmt diese zentrale Haustechnikkomponente nicht wahr, sondern kommuniziert direkt mit der *Service-Plattform* des ISST.

Für die Kommunikation mit externen Dienstleistern stehen verschiedene Schnittstellen zur Verfügung. Teilweise erfolgt diese über Web-Services zwischen der *Service-Plattform* und der jeweiligen IT-Infrastruktur der Dienstleister. Außerdem wird der Versand und Empfang von E-Mails eingesetzt, beispielsweise bei der Nutzung von Home-Delivery-Services. Welche Kommunikationsart für welchen IT-Mehrwertdienst verwendet wird ist sowohl von der Art des smarten Dienstes als auch von der IT-Infrastruktur des jeweiligen Dienstleisters abhängig.

4 ECA-Regelsysteme

Event-Condition-Action-Regelsysteme werden vorwiegend in aktiven Datenbanksystemen eingesetzt, um als verallgemeinerte Trigger die Integrität der verwalteten Datenbasis zu gewährleisten. Dabei bildet eine Menge von ECA-Regeln innerhalb einer Ausführungsumgebung ein ECA-Regelsystem.

Die Service-Plattform des SmarterWohnen-Projektes verfügt über eine Laufzeit-Umgebung für ECA-Regelsysteme, um ereignisorientierte Abläufe ausführen zu können; diese wird detailliert in Abschnitt 5.3.1 beschrieben. Um die notwendigen Grundlagen von ECA-Regelsystemen zusammenfassend darzustellen, gibt dieses Kapitel einen Überblick über verschiedene Aspekte solcher Systeme. In Abschnitt 4.1.1 wird zunächst der Aufbau von ECA-Regeln und deren Komponenten erläutert, während Abschnitt 4.1.2 die Ausführungssemantiken innerhalb einer Laufzeitumgebung von ECA-Regelsystemen beschreibt. Um Eigenschaften wie *Konfluenz* und *Terminierung* von ECA-Regelsystemen zu

überprüfen, werden in Abschnitt 4.1.3 Darstellungs- und Analysemethoden von Regelsystemen vorgestellt. Der Einsatz solcher Systeme in Workflow-Managementsystemen wird anhand des Prototyps $TriGS_{flow}$ exemplarisch vorgestellt. Dieses Kapitel basiert auf den Seminararbeiten [36] und [35] der Projektgruppe *Live* des Lehrstuhls V. Außerdem sei auf die darin zitierten weiterführenden Quellen [49], [1] sowie [23] verwiesen.

4.1 Grundlagen

4.1.1 Aufbau von ECA-Regeln

Ein ECA-Regelsystem besteht aus einer Menge $R = \{r_1, ..., r_n\}$ von Regeln r_i der Form $r_i = (E, C, A)$. Jede Regel wird demnach durch ein 3-Tupel (E, C, A) charakterisiert, wobei E eine Menge von Ereignissen ist, C eine optionale Bedingung und A eine Aktion. Dabei wird die Aktion A genau dann ausgeführt, wenn Ereignis E eintritt und zusätzlich Bedingung C erfüllt ist. Dieser Abschnitt beschreibt die Komponenten Ereignismenge, Bedingung und Aktion zur Darstellung von ECA-Regeln. Das Zusammenspiel von mehreren Regeln innerhalb eines Regelsystems R wird in den Abschnitten 4.1.2 und 4.1.3 näher beschrieben.

Ereignisse Ereignisse lassen sich grob in primitive und komplexe Ereignisse unterteilen. Primitive Ereignisse sind atomar und können daher nicht weiter in Teilereignisse zerlegt werden. Ein Beispiel für ein solches primitives Ereignis sind Zeitereignisse der Form „11. November 2007, 11:11 Uhr“. Das Erreichen dieses Zeitpunktes stellt das primitive Ereignis dar. Im Gegensatz dazu, stellen komplexe Ereignisse durch Operatoren verknüpfte Ereignisse dar, bei denen es sich sowohl um primitive als auch um komplexe Ereignisse handeln kann. Auf diese Weise lassen sich mit Hilfe von Booleschen Operatoren, Auswahloperatoren, Sequenzoperatoren, Zeit-Operatoren, Intervall- und Wiederholungsoperatoren komplexe Ereignisse bilden.

Das mit Hilfe des Booleschen Operators $\vee$ gebildete komplexe Ereignis $E_k = E_i \vee E_j$ tritt somit genau dann ein, wenn entweder E_i oder E_j eintritt. Ist eine Ereignismenge $E = E_1, ..., E_n$ sowie eine Zahl $n \in \mathbb{N}$ gegeben, dann wird mit Hilfe des Auswahloperators das komplexe Ereignis E_k genau dann ausgeführt, wenn n Ereignisse aus E eintreten. Durch Sequenz-Operatoren können komplexe Ereignisse $E_k =< E_i, E_j, E_l >$ gebildet werden, die dann erkannt werden, wenn Ereignisse in einer vorgegebenen Reihenfolge auftreten. Zeit-Operatoren bieten die Möglichkeit periodische Zeitereignisse[11] sowie relative Zeitereignisse[12] zu definieren. Soll ein komplexes Ereignis dann auftreten, sobald ein Ereignis E_j zeitlich nach E_i und vor E_l ausgelöst wurde, so kann dies mit Hilfe des Intervall-Operators ausgedrückt werden. Der Wiederholungsoperator ermöglicht hingegen die Definition eines komplexen Ereignisses E_k, welches dann auftritt, sobald ein Ereignis E_i n-mal aufgetreten ist. Weiterhin können Ereignisse zusätzlich Parameter enthalten, welche beispielsweise innerhalb des Bedingungsausdrucks einer Regel verwendet werden können.

Bedingungen Bedingungsausrücke von ECA-Regeln werden durch die Laufzeitumgebung ausgewertet und auf ihre Gültigkeit überprüft. Auch bei Bedingungen ist eine Unterscheidung zwischen primitiven und komplexen Bedingungen vorzunehmen. Primitive Bedingungen werden einerseits durch die Vergleichsoperatoren $<$, $>$, $\leq$, $\geq$ und $\neq$ sowie zwei zu vergleichenden Parametern gebildet. Andererseits können auch Funktionsaufrufe mit Booleschen Rückgabewerten verwendet werden. Komplexe Bedingungen hingegen

[11] z. B. alle 30 Sekunden
[12] z. B. 10 Sekunden nach Eintreten des Ereignisses E_i

werden durch Verknüpfung mit primitiven und komplexen Bedingungen mit Hilfe von Booleschen Operatoren wie z. B. $\wedge$, $\vee$, $\oplus$ und $\neg$ gebildet.

Aktionen Die Aktion einer ECA-Regel wird dann ausgelöst, sobald das spezifizierte Ereignis eintritt und der Bedingungsausdruck als *TRUE* ausgewertet wird. Primitive Aktionen stellen dabei die Ausführung von genau einer vom System bereitgestellten Aktion dar. Hingegen werden komplexe Aktionen als Sequenz mehrere Aktionen bezeichnet.

4.1.2 Ausführungssemantiken

Die Semantik einer Regelmenge R ist abhängig von der Ausführungsumgebung, die mit Hilfe der Regelmenge bei auftretenden Ereignissen abhängig vom Systemzustand Aktionen ausführen und den Systemzustand manipulieren kann. Durch diese Veränderung des Zustandes können weitere Ereignisse ausgelöst werden, deren Auftreten ihrerseits zur Ausführung von weiteren ECA-Regeln führen können. Wie das System die in 4.1.1 genannten Operatoren zur Definition von komplexen Ereignissen interpretiert und in welcher Reihenfolge Regeln ausgewertet werden, hängt maßgeblich von der Implementierung der Ausführungsumgebung ab.

Prioritäten Sind in einer Regelmenge mehrere Regeln vorhanden, die bei Auftreten eines Ereignisses ausgewertet werden müssen, so kann durch eine Zuordnung von Prioritäten deterministisch festgelegt werden, in welcher Reihenfolge diese Regeln ausgewertet und im Falle der erfüllten Bedingung der zugehörige Aktionsteil ausgeführt wird. Sei R eine Regelmenge und $P \subset \mathbb{N}$ eine total geordnete Prioritätsmenge, so werden durch eine injektive Abbildung $f : R \rightarrow P$ der Regelmenge absolute Prioritäten zugeordnet. Eine weitere Möglichkeit ist die Zuordnung von relativen Prioritäten, bei der für je zwei Regeln r_i und r_j eine Regel höher priorisiert wird als die andere.

Parameterkontexte Wie in Abschnitt 4.1.1 beschrieben, kann mit Hilfe der genannten Operatoren beispielsweise ein komplexes Ereignis E_k der Form „erst E_i, dann E_j“ gebildet werden. Bei einer gegebenen Ereignissequenz $< E_i, E_i^{\cdot}, E_j >$ von primitiven Ereignissen, die innerhalb der Ausführungsumgebung auftritt, werden zwei aufeinander folgende Ereignisse E_i gefolgt von E_j ausgelöst. Dabei unterscheiden sich in diesem Beispiel die Ereignisse E_i und $E_i^{\cdot}$ nur in Bezug auf die in ihnen enthaltenen Parameter, beide Ereignisse sind hingegen vom Typ i. Mit dem Auftreten des Ereignisses E_j wird zudem das komplexe Ereignis E_k ausgelöst, da zu diesem Zeitpunkt ein Ereignis vom E_i gefolgt von einem Ereignis E_j aufgetreten ist. Es stellt sich jedoch die Frage, in welchem Parameterkontext das komplexe Ereignis E_k ausgewertet werden soll. Es können einerseits die in den Ereignissen E_i und E_j enthaltenen Parameter herangezogen werden, aber auch die Parameter des Ereignispaares $E_i^{\cdot}$, E_j. Welcher Parameterkontext verwendet wird, ist abhängig von der Ausführungssemantik und wird durch die Implementierung des Ausführungsumgebung festgelegt.

Es werden vier Verfahresweisen zur Spezifikation des Parameterkontextes unterschieden. Während beispielsweise *recent* stets die Parameter der jüngsten Ereignisse verwendet, erfolgt bei *chronicle* die Parameterauswahl innerhalb der chronologischen Reihenfolge. Weiterhin stehen die Verfahrenweisen *continuous* und *cumulative* unterschieden [36].

4.1.3 Darstellung und Analyse von ECA-Regelmengen

Zur Visualisierung von ECA-Regelsystemen werden gerichtete Graphen verwendet, auf deren Basis das Regelsystem auch auf Eigenschaften wie Konfluenz und Terminierung

überprüft werden kann.

Triggering Graph Um Abhängigkeiten zwischen Regeln darzustellen, können *Triggering Graphs* verwendet werden. Ein Triggering Graph (V,E) besteht aus einer der Regelmenge entsprechenden Knotenmenge V, sowie einer Kantenmenge $E \subseteq V \times V$. Dabei ist das Tupel (r_i,r_j) genau dann in E enthalten, wenn durch die Verarbeitung der Regel r_i die Regel r_j ausgeführt werden kann, also die Ausführung der Aktionskomponente von E_i zur Auslösung des in E_j spezifizierten Ereignisses führt. Ob r_i die Regel r_j tatsächlich auslöst ist zusätzlich vom Systemzustand abhängig, da die Bedingung von r_i Voraussetzung zum Auslösen der Regel ist. Abbildung 3 zeigt einen Triggering Graphen.

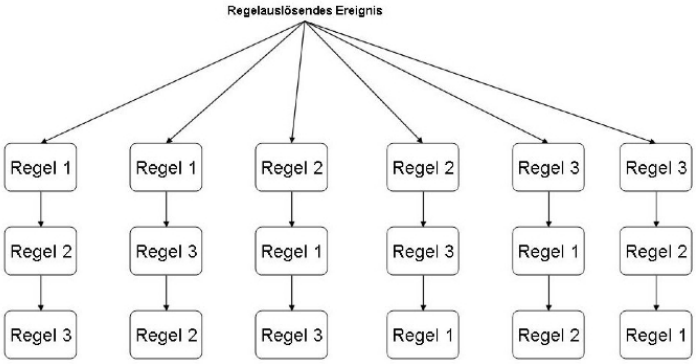


Abbildung 3: Triggering Graph [36]

Execution Graphs Eine zustandsbasierte Sichtweise auf die Interaktion von Regelsystemen in einem Gesamtsystem bieten Exceution Graphs. Ein Execution Graph (V,E) besteht aus einer Knotenmenge V sowie aus einer Kantenmengen E. Dabei enspricht V der Zustandsmenge Σ des regelausführenden Systems. Eine Kante $r_i = (\sigma,\sigma')$ ist genau dann in E enthalten, wenn die Ausführung von Regel r_i im Zustand σ zu einem Zustandswechsel zu σ' führt. Endzustände sind Knoten, mit einem Ausgangsgrad von 0. Mit Erreichen eines Endzustandes wird somit keine Regel mehr ausgelöst und das System terminiert. In Abbildung 4 ist ein Execution Graph dargestellt.

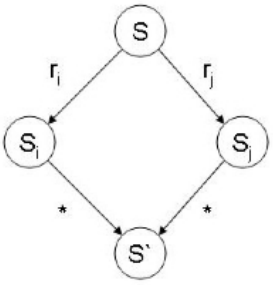


Abbildung 4: Execution Graph [36]

Terminierung Die in einem Triggering Graphen dargestellte Relation zwischen Regeln eines Regelsystems lässt die Analyse des Regelsystems auf die Eigenschaft der Terminierung zu. Ein Regelsystem terminiert, wenn Regeln sich unabhängig vom Anfangszustand

nicht unendlich oft gegenseitig aktivieren können. Wird ein Regelsystem R durch einen Triggering Graphen G repräsentiert, so terminiert R, wenn G azyklisch ist und keine unendlichen Wege enthält. Die Umkehrung dieser Implikation zur Erweiterung der Aussage zur Äquivialenz ist nicht gültig, da die Relation die Abhängigkeit von Regeln unabhängig vom Systemzustand modelliert. Würde für einen Triggering Graphen G mit Kantenmenge E gelten, dass $\{(r_i, r_j), (r_j, r_l), (r_l, r_i)\} \subseteq E$, so könnte für alle Zustände $\sigma \in \Sigma$‘ gelten, dass die Bedingung von r_l nicht erfüllt ist und somit keine Regelkaskade vorliegt. Das System würde folglich terminieren, obwohl G zyklisch ist. Außerdem kann durch diese statische Analyse mit Hilfe von Triggering Graphen ohne Betrachtung der Zustandsmengen nicht ausgeschlossen werden, dass ein vorhandener Zyklus nicht nach endlichen Durchläufen abbricht. Ein Triggering Graph mit Regelkaskaden und zwei Zyklen ist in Abbildung 5 dargestellt.

Um für ein gegebenes Regelsystem R sicherstellen zu können, dass es terminiert, können einerseits durch Modifikation von R alle Zyklen aus dem Triggering Graphen entfernt werden. Zudem muss gewährleistet sein, dass im Triggering Graph keine unendlichen Pfade existieren. Weiterhin wäre die Implementierung von globalen Zählern in der Ausführungsumgebung denkbar, durch welche die maximale Verarbeitungstiefe von Regelkaskaden limitiert werden kann. Die Regelausführung wird nach Erreichen eines vorab festgelegten Limits abgebrochen. Dieses Verfahren kommt in aktiven Datenbanksystemen zur Anwendung.

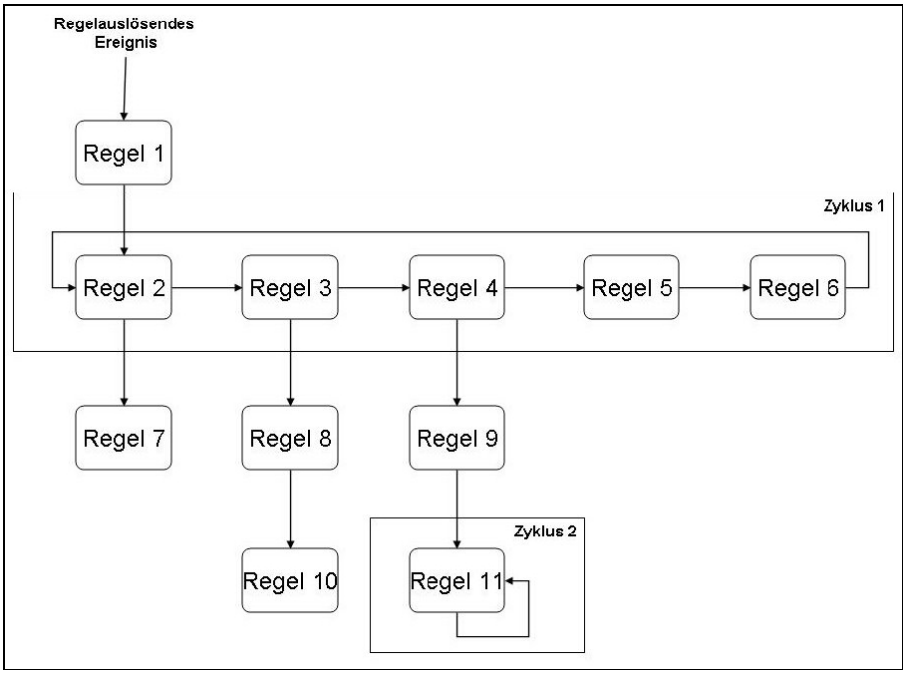


Abbildung 5: Regelkaskaden und Zyklen in ECA-Regelsystemen (Triggering Graph) [36]

Konfluenz Im vorherigen Abschnitt wurde bereits erwähnt, dass die Semantik der Regeln eines Regelsystems von der Spezifikation der Ausführungsumgebung abhängig ist. Werden zwei Regeln r_i und r_j durch ein Ereignis E_k gleichzeitig ausgelöst und die Ausführungsumgebung führt eine sequentielle Auswertung und Ausführung von Regeln durch, so kann durch die Vergabe von Prioritäten die Reihenfolge der Regelbehandlung eindeutig festgelegt werden. Findet keine solche Regelpriorisierung statt, so kann nicht vorhergesagt werden, in welcher Reihenfolge die Regeln ausgewertet werden und das System verhält sich

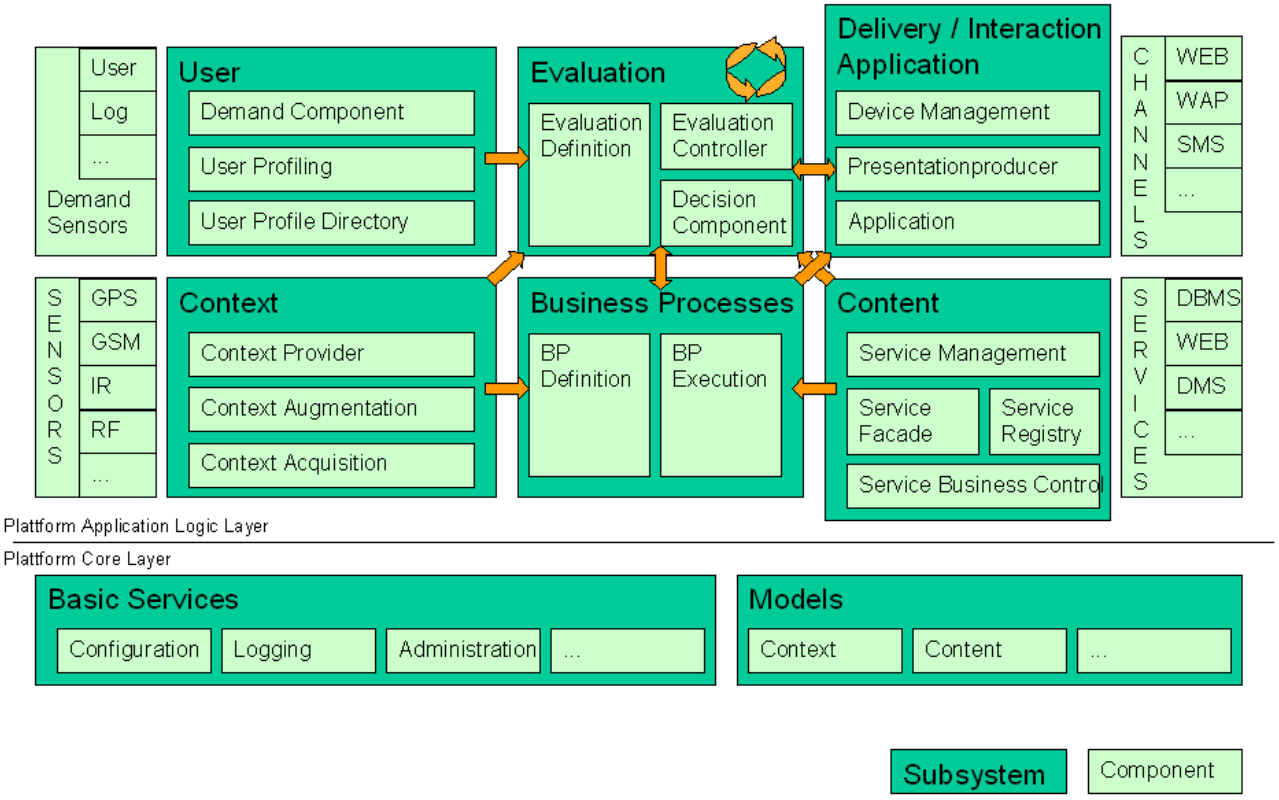


Abbildung 6: Architektur der *IGM-Plattform*

nicht deterministisch. Hat jedoch die Ausführungsreihenfolge von gleichzeitig feuernden Regeln keine Auswirkung auf den nach der Verarbeitung erreichten Systemzustand, d. h. es wird für alle möglichen Reihenfolgen gleichzeitig auszuführender Regeln stets derselbe Zustand σ erreicht, so ist das Regelsystem konfluent. Wird hingegen eine injektive Priorisierung der Regeln vorgenommen, so gibt es in keinem Zustand mehr als eine ausführbare Regel. Das System ist somit trivialerweise konfluent.

5 IGM-Plattform

Wie im Abschnitt 2 bereits erwähnt, soll das *SmarterWohnen*-Projekt auf die *IGM-Plattform* portiert werden. Im folgenden Kapitel wird nun eine Einführung in Funktionsweise und Aufbau der Plattform gegeben.

5.1 Struktur und Konzeption der Plattform

Die *IGM-Plattform* stellt eine informationslogistische Dienstleistungsplattform dar. Abbildung 6 zeigt die Architektur der Plattform. Wie in der Abbildung zu sehen, teilt sich die Funktionalität der Plattform auf sechs Subsysteme auf. Konzeptionell liegt der Plattform eine Aufteilung in zwei logische Teile zu Grunde, einen Ausführungsteil und einen Evaluierungsteil. Der Evaluierungsteil besteht aus den Subsystemen *User*, *Context*, *Service* und *Evaluation* und realisiert damit die informationslogistische Auswertung, wohingegen der Ausführungsteil, die Komponente *Delivery/Interaction/Application*, als eine Art Platzhalter für die Anwendung dient, die um informationslogistische Aspekte erweitert werden

soll. Jedes Subsystem realisiert somit einen gegenüber den anderen Subsystemen klar abgegrenzten Funktionsumfang. Eine Kommunikation zwischen den einzelnen Subsystemen ist durch definierte Schnittstellen möglich. Hierbei wird grundsätzlich zwischen synchroner und asynchroner Kommunikation unterschieden. Synchrone Kommunikation dient zumeist dem Abfragen und Ändern von Informationen, die durch Subsysteme bereitgestellt werden. Eine synchrone Kommunikation ist daher nur zwischen bestimmten Subsystemen möglich. Eine asynchrone Kommunikation hingegen ist zwischen allen Subsystemen möglich und wird zur Behandlung von Events genutzt. Events dienen den Subsystemen zur Mitteilung von bemerkten (realen) Ereignissen. Konzeptionell liegt hier ein Request-Event-Ansatz zu Grunde (vgl. [29], Kapitel 2). Events werden von einem Subsystem A an ein Subsystem B nur dann versendet, wenn Subsystem B diese Events im Voraus angefordert hat (Request). Die einzelnen Subsysteme stellen sich wie folgt dar: Das *User*-Subsystem dient zur Verwaltung von benutzerspezifischen Informationen, das *Kontext*-Subsystem zum Erkennen, Verwalten und Benutzen von Kontextinformationen. Zur informationslogistischen Auswertung von Ressourcen dient das *Evaluation*-Subsystem, zur Verwaltung und Bereitstellung von Diensten das *Service*-Subsystem und zur Definition und Ausführung von Geschäftsprozessen das *Business Processes*-Subsystem. Darüber hinaus existiert das *Delivery/Interaction/Application*-Subsystem zur Beschreibung der Anwendungslogik, durch das dem Benutzer der Zugang zur Anwendungsfunktionalität und den damit verbundenen Informationen möglich ist.

5.2 Technische Aspekte

Eine Implementierung der Plattform auf Basis des *Microsoft .Net 2.0-Frameworks* steht zur Verfügung. Zu Beginn der Projektgruppe wurde am ISST eine Migration auf die Version 3.0 evaluiert. Im Rahmen der Projektgruppe diente allerdings die Implementierung auf der Basis von *.Net 2.0* als Grundlage. Die einzelnen Subsysteme sind als *MS.Net Enterprise Services* realisiert. Synchrone Kommunikation zwischen den Subsystemen findet mit Hilfe von *DCOM-RPC* statt. Für die Behandlung von Ereignissen (asynchrone Kommunikation) existieren allgemeine Klassen, die generische Funktionen für die Wiederverwendung in den Subsystemen als Software-Bibliothek bereit stellen. Eine Implementierung liegt auf Basis von *Microsoft Message Queue (MSMQ)* vor.
Das Subsystem *Business Processes (BPM)* ist in Form einer Implementierung nicht vorhanden, es existieren aber konzeptionelle Beschreibungen und Anforderungen an die Schnittstellen.
Für alle weiteren Subysteme sind ebenfalls Schnittstellenspezifikationen vorhanden und es wird eine Implementierung vom ISST zur Verfügung gestellt.

Nachfolgend wird für die einzelnen Subsysteme eine nähere Beschreibung gegeben:

5.3 Subsysteme

5.3.1 Evaluation (EVA)

Das EVA-Subsystem Das Subsystem *Evaluation* stellt einen wesentlichen Bestandteil der Plattform dar, dessen Hauptaufgabe in der informationslogistischen Auswertung von Ressourcen besteht. Diese Auswertung erfolgt im Zusammenspiel mit den anderen Subsystemen der Plattform. Zum einen können Aktionen ausgeführt werden, wenn bestimmte Bedingungen erfüllt sind (ECA-Paradigma, PUSH-Ansatz), zum anderen besteht die Möglichkeit Ad-hoc Anfragen zum Ausführen von Aktionen zu stellen (PULL-Ansatz).

Ersteres wird im Rahmen der Plattform als *subscription* und letzteres als *inquiry* bezeichnet.
Durch ein permanentes Monitoring und den Empfang von Events, die durch die anderen Subsysteme gesendeten wurden, ist es dem Subsystem möglich, Informationsbedürfnisse oder Wünsche eines Benutzers nach bestimmten Diensten zu ermitteln und entsprechende Aktionen auszuführen, um diese Bedürfnisse zu befriedigen. Ereignisse und Bedingungen (EC) werden auf der Plattform durch *ec-jobs* ausgewertet. Hierzu liegt eine Spezifikation und Beispielimplementierung von *boolean ec-jobs* vor, die den Rahmen eines normalen *ec-jobs* um Funktionalitäten erweitern, die genutzt werden können um alternative Aktionen auszuführen.
Neben diesem permanenten Monitoring kann das Subsystem auch durch andere Subsysteme mit der Ad-hoc-Auswertung von Ressourcen „beauftragt“ werden. Für die damit verbundene Ad-hoc-Ausführung von Aktionen existiert das Konzept der *action-jobs*. Auch hierzu liegt eine Spezifikation mit einigen Beispielen für die Implementierung vor.

Repräsentation und Spezifikation von ECA-Regeln Der folgende Abschnitt beschreibt die Realisierung von ECA-Regeln auf der IGM-Plattform durch das EVA-Subsystem. Die Beschreibung der einzelnen IGM-spezifischen Komponenten sind der Quelle [53] entnommen.

Boolean ec-jobs und BooleanECAs Das Konzept der in Abschnitt 4 beschriebenen ECA-Regelsysteme wird in Form von *Boolean ec-jobs* auch von der IGM-Plattform unterstützt. Ausführungsumgebung für diese Regelsysteme ist das Evaluation-Subsystem. Ein *Boolean ec-job* fasst eine oder mehrere als *BooleanECA* bezeichnete ECA-Strukturen zusammen. Ein *BooleanECA* besteht aus einer Bedingungskomponente und einer Aktionskomponente. Weiterhin wird in jedem *BooleanECA* festgelegt, welche Ereignisse zur einer Aktivierung führen, d. h. zu einer Auswertung der Bedingungskomponente und ggf. der Ausführung einer Aktionskomponente. Abbildung 7 zeigt einen Boolean ec-job, der drei verschiedene BooleanECAs implementiert. Solche Ereignisse, die von einem oder von mehreren BooleanECAs zwecks Aktivierung verarbeitet werden sollen, werden global vom Boolean ec-job an der Ausführungsumgebung abonniert. Sobald ein ensprechendes Ereignis auftritt, wird dieses zum Boolean ec-job geleitet, der das Ereignis wiederum an alle implementierten BooleanECAs weiterreicht. Sofern es sich bei dem Ereignis um ein für das BooleanECA relevantes Ereignis handelt, wird die Auswertung der Bedingung, wie oben bereits erwähnt, durchgeführt.

Repräsentation von ECA-Regelsystemen Für ein wie in Abschnitt 4 genanntes Regelsystem $R = \{r_1, ..., r_n\}$, bestehend aus ECA-Regeln r_i der Form $r_i = (E, C, A)$, entspricht R der Menge aller BooleanECAs von allen in der Ausführungsumgebung definierten Boolean ec-jobs. Die Boolean ec-jobs können bei dieser Betrachtung ignoriert werden, da sie lediglich Funktionen zur Realisierung der BooleanECAs bereitstellen, selbst jedoch keine ECA-Regel im formalen Sinne darstellen. So werden beispielsweise Datenstrukturen und Persistenzmechanismen zur Verfügung gestellt. In den folgenden Abschnitten soll nun auf die Repräsentation des 3-Tupels (E, C, A) eingegangen werden, um die Realisierung der Komponenten Event, Condition und Action innerhalb des EVA-Subsystems zu analysieren.

Ereignisse Ereignisse sind standardisierte von anderen Subsystemen bereitgestellte Objekte zum Datenaustausch. Es lassen sich hierbei verschiedene Ereignistypen unterscheiden. Ereignisse verfügen über bestimmte Parameter zur weiteren Beschreibung des Ereignisses und der damit verbundenen Information. In jedem BooleanECA wird eine

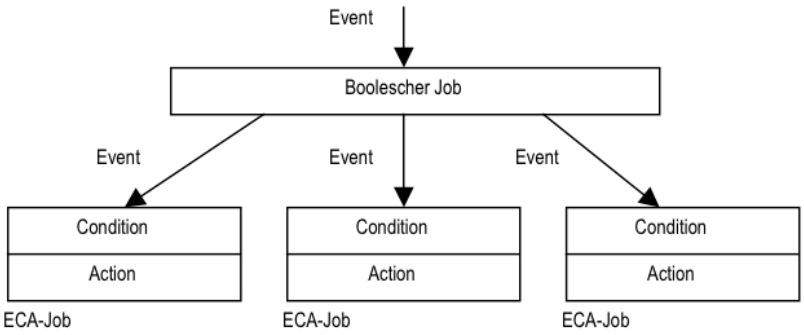


Abbildung 7: Boolean ec-job mit drei BooleanECAs[56]

Menge von Ereignistypen definiert, bei deren Auftreten die Auswertung der Bedingungskomponente angestoßen wird. Dabei wird diese Auswertung für jedes zutreffende Ereignis durchgeführt. Jedes dieser voneinander unabhängigen Ereignisse kann dabei als primitives Ereignis im Sinne von Abschnitt 4.1.1 angesehen werden, da eine Funktionalität zur Definition von komplexen Ereignissen mittels Operatoren wie beispielsweise dem Booleschen Operator oder dem Sequenzoperator nicht implementiert ist. Außerdem können primitive Zeitereignisse verwendet werden, z. B. „point of time: 16h“. Besonders hervorzuheben ist, dass Boolean ec-jobs und somit auch die darin gekapselten BooleanECAs implementationsbedingt ausschließlich Ereignisse von anderen Subsystemen empfangen können, nicht jedoch Ereignisse, die von Boolean ec-jobs selbst ausgelöst werden. Als Konsequenz kann daher festgehalten werden, dass ein solches Regelsystem durch einen Triggering-Graphen mit leerer Kantenmenge E abgebildet wird, da die durch BooleanECAs repräsentierten ECA-Regeln sich nicht direkt untereinander auslösen können.

Bedingungen In jedem BooleanECA können als Bedingungskomponente Boolesche Terme verwendet werden. Die ECA-Regeln verfügen somit sowohl über primitive als auch komplexe Bedingungen. Es werden die Booleschen Operatoren $\vee$ (*BooleanOrTerm*), $\wedge$ (*BooleanAndTerm*) und $\neg$ (*BooleanNotTerm*) zur Konstruktion von Booleschen Termen bereitgestellt. Als Operanden dieser Operationen können hierbei sowohl primitive Bedingungen als auch weitere Boolesche Terme verwendet werden.

Aktionen Als Aktionskomponente besitzt jedes BooleanECA die Methode `DoAction()`, in der die Menge der auszuführenden Aktionen implementiert wird. In der Repräsentation eines BooleanECAs als ECA-Regel werden somit sowohl primitive als auch komplexe Aktionen unterstützt.

Das EVA-Subsystem als Ausführungsumgebung

Parameterkontexte Da keine komplexen Ereignisse durch die Ausführungsumgebung EVA-Subsystem unterstützt werden, ist eine Analyse unterstützter Parameterkontext-Verfahren nicht sinnvoll. Jedes auftretende Ereignis wird mit seinen Parametern sofort ausgewertet und kann, da es sich nicht um ein komplexes zusammengesetztes Ereignis handelt, bei unterschiedlichen Auswertungsverfahren keinen unterschiedlichen Informationsgehalt aufweisen.

Priorisierung Das EVA-Subsystem unterstützt weder eine globale Priorisierung von Regeln, noch eine relative. Somit ist eine Reihenfolge bei der Abarbeitung von ECA-

Regeln nicht eindeutig definiert. Das EVA-Subsystem unterstützt jedoch eine quasi parallele Auswertung und Ausführung von Regeln. Dabei werden alle BooleanECAs eines Boolean ec-jobs zwar nacheinander bedient, jedoch stehen Funktionen bereit um den Zustand des Boolean ec-jobs einzufrieren und somit allen BooleanECAs eine identische Kopie dieses Zustands zum Zeitpunkt des Eintreffen des auslösenden Ereignisses zur Verfügung zu stellen. Auf diese Weise kann den ECA-Regeln eine Simultanität im Hinblick auf die Sicht des aktuellen Zustandes simuliert werden. Dieses Konzept kann jedoch insbesondere beim schreibenden Zugriff auf den Datenspeicher zu Race Conditions[13] führen und bedarf daher trotz der Abstraktion einer Validierung des Entwicklers. Außerdem ist der Wirkungsbereich dieses temporären Snapshots nur auf BooleanECAs eines Boolean ec-jobs beschränkt, d. h. BooleanECAs verschiedener Jobs können nicht „gleichzeitig“ ausgeführt werden. Für das durch die Menge aller Jobs realisierte ECA-Regelsystem steht somit kein konsistenter Mechanismus zur Verfügung, eine eindeutige Ausführungsreihenfolge der ECA-Regeln zu festzulegen und das Verhalten des Systems kann somit nicht deterministisch bestimmt werden.

Terminierung Wie bereits erwähnt, kann eine im EVA-Subsystem in Form eines BooleanECAs repräsentierte ECA-Regel aktiv keine Ereignisse erzeugen, um somit selbst weitere ECA-Regeln auszulösen. Somit wären keine Regelkaskaden und Zyklen innerhalb eines Triggering Graphen möglich und das System würde trivialerweise terminieren. Jedoch ist bei dieser Aussage zu beachten, dass innerhalb der Aktionskomponente durch Funktionsaufrufe anderer Subsysteme der globale Systemzustand verändert werden kann. Dies hat zur Folge, dass möglicherweise von diesen Subsystemen Ereignisse erzeugt werden, die ihrerseits ECA-Regeln des Regelsystems auslösen. Daher sind in einer abstrakteren Sichtweise die ECA-Regeln sehr wohl im Stande, gegenseitige Aufrufe zu erzeugen und somit möglicherweise unendliche Kaskaden auszulösen, obwohl dies nicht direkt aktiv durch die ECA-Regeln erfolgen kann. Somit ist also die Implementierung eines ECA-Regelsystems innerhalb des EVA-Subsystems möglich, dessen Regelmenge nicht terminiert. Ob die Eigenschaft der Terminierung letztendlich einem konkreten im EVA-Subsystem implementierten Regelsystem zugesprochen werden kann, muss im Einzelfall evaluiert werden und hängt von der Menge der BooleanECAs und den kapselnden Boolean ec-jobs ab.

Konfluenz Auch die Eigenschaft der Konfluenz kann nur für jedes konkret vorliegende ECA-Regelsystem gesondert festgestellt werden. Die Implementierung des EVA-Subsystems als Ausführungsumgebung lässt keine allgemeingültigen Schlüsse in Hinblick auf diese Eigenschaft zu. Beispielsweise können zwei ECA-Regeln r_1 und r_2 durch zwei Boolean ec-jobs b_1 und b_2 realisiert werden, die folgendes Verhalten aufweisen: r_1 wird durch das Ereignis E ausgelöst und schreibt Datum d_1 in die Speicherstelle x. Regel r_2 wird ebenfalls durch E ausgelöst, beschreibt x hingegen mit Datum d_2. Da keine Priorisierung der Regeln vorgenommen werden kann, kann das Auftreten des Ereignisses E zu zwei unterschiedlichen Zuständen führen: Wird zunächst Job b_1 ausgeführt, enthält x nach Abarbeitung beider Jobs das Datum d_2, bei umgekehrter Reihenfolge jedoch das Datum d_1. Somit ist der erreichte Zustand von der Reihenfolge der Regelausführung abhängig und das System insgesamt nicht konfluent.

5.3.2 Context

Das *Kontext-Subsystem* stellt die Informationen bereit, mit denen man die Situation einer Entität, z.B. User beschreiben kann. Die Komponenten des Systems bieten unter-

[13] Wettlaufsituation

schiedliche Möglichkeiten mit den Informationen zu arbeiten. Diese umfassen den Zugriff (synchron, asynchron), die Überwachung, Änderung und Auswertung der Daten. Jede Komponente stellt Interfaces bereit, über die man auf die Funktionalität der Komponente zugreifen kann. Die konkrete Implementierung der Interfaces ist zur Zeit noch nicht vollständig oder nur rudimentär umgesetzt.

Die Bereitstellung der Informationen folgt dem *Publish/Subscribe*-Ansatz. Jedes System, das Informationen von dem *Kontext-Subsystem* erhalten möchte, muss sich zuerst dafür anmelden bzw. registrieren. Anhand der Anmeldung oder *Subscription* ist dem *Kontext-Subsystem* bekannt, welche Daten überwacht und wo sie bereitgestellt werden müssen. Um sich abzumelden muss ein System seine *Subscription* wieder aufheben. Die aktuelle Umsetzung des *Kontext-Subsystems* benutzt eine MSSQL Datenbank um Daten zu speichern und zu überwachen.

5.3.3 User

Das *User-Subsystem* dient zur Verwaltung der Benutzerdaten. Es basiert auf einem LDAP-Verzeichnis, das, abhängig von der Anwendung, unterschiedliche Daten enthält. In der aktuellen Implementierung benutzt es Microsofts Active Directory. Das Subsystem enthält alle notwendigen Informationen über den User. Diese Informationen erhält es entweder über den Benutzer direkt oder über ein Profiling, welches das System durchführt.

Die Daten dienen zur Personalisierung der Anwendung, aber auch um die Bedürfnisse des Benutzers festzustellen. Das Subsystem ist so angelegt, dass es die Daten mehrerer Anwendungen zur Verfügung stellen kann. Gleichzeitig sollten die Anwendungen es dem Benutzer ermöglichen, volle Kontrolle über seine Daten zu behalten.

Die Daten werden mit Hilfe so genannter Profile abgefragt, die im Wesentlichen aus Schlüssel-Wert-Paaren bestehen. Anhand der im Profil übergebenen Schlüssel, meist die Attribut-Namen, stellt das Subsystem fest, welche Daten gewünscht sind und trägt diese in das Profil ein. Auf ähnliche Art und Weise werden auch die Benutzerdaten in das Subsystem geschrieben. Der Benutzer wird über eine eindeutige ID identifiziert, ebenso muss jeweils über eine eindeutige ID angegeben werden, welche Anwendung die Daten anfordert. Darüber hinaus stellt das Subsystem Methoden zur Verfügung, um die Struktur des zu Grunde liegenden LDAP-Verzeichnisses zu ergründen und die zur Verfügung stehenden Attribute zu erfragen.

5.3.4 Delivery/Interaction/Application (DIA)

DIA steht für *Delivery/Interaction/Application*, was auch direkt die Aufgaben dieses Subsystems umschreibt: Erstellung und Zustellung von Informationen für Benutzer, Bereitstellung von Methoden, um Rückmeldungen vom Benutzer verarbeiten zu können und die Anbindung anderer Anwendungen, die nicht direkt in die *IGM-Plattform* eingebunden werden sollen. Das Subsystem ist in der Lage, die Informationen selbst in unterschiedliche Formen wie reine Texte für SMS und E-Mail oder HTML für unterschiedliche Anzeigegeräte aufzubereiten, um sie über unterschiedliche Kommunikationswege wie SMS, E-Mail und Internet dem Benutzer zur Verfügung zu stellen. *DIA* implementiert alle „technischen“ Funktionen, die für die Kommunikation nötig sind; es stellt sicher, dass das Empfangsgerät verfügbar ist, die Übermittlung der Daten funktioniert hat und gibt bei möglichen Fehlern diese an die aufrufende Methode zurück.
Zur Zeit sind nur die Möglichkeiten, eine SMS und eine E-Mail zu verschicken, implementiert, im aktuellen Stand des *SmarterWohnen*-Projektes werden aber die Bereitstellung der Informationen und die Interaktion mit dem Benutzer noch nicht durch DIA realisiert.

5.3.5 Services

Das *Service-Subsystem* ist für die Verwaltung und Bereitstellung von informations- und inhaltsbezogenen Diensten verantwortlich, die von Drittanbietern bereitgestellt werden. Solche Dienste können über eine entsprechende Schnittstelle dynamisch im Subsystem registriert werden. Des Weiteren werden für die anderen Subsysteme Funktionalitäten zum Auffinden und Benutzen der Dienste angeboten. Zu diesem Zweck hält das Subsystem Metainformationen vor, die durch die Dienstanbieter bereit gestellt werden müssen und beispielsweise Informationen zum Inhalt, Zugriffsmethoden oder Kontexten beschreiben. Neben diesen Funktionalitäten zum Auffinden und Verwalten der angebotenen Dienste bietet das Subsystem auch die Infrastruktur für Zugriffe auf diese Dienste.

6 Prozesse

6.1 Überblick

Die in Kapitel 3 genannten IT-gestützten Mehrwertdienste des *SmarterWohnen*-Projektes stellen für sich komplexe Abläufe dar, welche insgesamt dem Zweck der Erbringung einer zuvor festgelegten Dienstleistung dienen. Die Realisierung und Bereitstellung einer solchen Dienstleistung kann als Geschäftsprozess aufgefasst werden. Im folgenden Abschnitt wird zunächst der Begriff Geschäftsprozess erläutert und im Anschluss Parallelen zu den *SmarterWohnen*-Diensten aufgezeigt. Weiterhin sollen einige Aspekte im Zusammenhang mit Geschäftsprozessen genannt werden.

Ein Geschäftsprozess stellt eine zeitliche Abfolge von Aktivitäten dar, welche inhaltlich abgeschlossen ist und somit des Erreichens eines vorbestimmten Ziels dient. Hierbei kann es sich einerseits um einen Produktionsprozess handeln, bei welchem die Erzeugung eines Produktes im Vordergrund steht; andererseits kann dieses Produkt auch als Erbringung und Bereitstellung einer Dienstleistung aufgefasst werden. [3]

Aus betriebswirtschaftlicher Sicht handelt es sich bei einem Geschäftsprozess insgesamt um eine Sammlung von durch Ereignisse verbundenen Aktivitäten, die einen Input, also eine Menge von Eingaben, verwenden, um einen aus Kundensicht höherwertigen Output zu erzeugen. Ein solcher Geschäftsprozess besteht somit aus einer Menge von Aktivitäten bzw. (Teil-) Aufgaben sowie einer Menge von Bedingungen, welche die Reihenfolge dieser Aktivitäten festlegen. Diese Ausführungsreihenfolge kann in verschiedenen Ausprägungen innerhalb eines Geschäftsprozesses auftreten. Mögliche Reihenfolgen sind die sequentielle, die parallele, die bedingte, die rekursive sowie die iterative Ausführung von Aufgaben. [30]

Die Instanzen, welche die im Geschäftsprozess definierten Aufgaben ausführen, werden Resourcen genannt. Hierbei handelt es sich um zur Verrichtung der Aufgaben geeignete Personen, produktionstechnische Anlagen (z. B. Maschinen), aber auch Resourcen aus informationstechnischer Sicht wie beispielsweise Softwarekomponenten. Weiterhin können diese Resourcen zu Organisationseinheiten zusammengefasst und somit in geeigneter Weise strukturiert werden. [30]

Bei der Entwicklung, Wartung und Pflege von Prozessen werden folgende Begrifflichkeiten voneinander abgegrenzt. Das *Prozessdesign* fasst Aspekte der Neugestaltung von Geschäftsprozessen zusammen, indem zunächst zu definieren ist, welche Ziele mit welchen

Mitteln mit welchen Methoden realisiert werden sollen. Demgegenüber steht die *Prozessverbesserung*, welche auf die Optimierung von bestehenden bereits entwickelten Geschäftsprozessen abzielt; hierbei stehen das *Business Process Reengineering (BPR)* sowie das *Continuous Process Improvement (CPI)* zur Verfügung. Während *BPR* ein fundamentales Redesign von Prozessen darstellt, bei dem bereits bestehende Geschäftsprozesse durch neu entwickelte ausgetauscht werden, steht bei *CPI* die kontinuierliche Verbesserung von Geschäftsprozessen und zu Grunde liegenden Strukturen im Vordergrund. [30]
Die Prozessglättung beschäftigt sich unter anderem mit der Behandlung, dem Beheben und dem Standardisieren von Ausnahmen. Es wird eine höhere Prozesssicherheit erreicht je geglätteter und standardisierter ein Prozess ist.

Es stehen zwei Methoden zur Entwicklung, Wartung und Pflege von Prozessen zur Verfügung. Auf der einen Seite die *Top-Down-Methode*, bei der von der groben zur feinen Sicht modelliert wird und auf der anderen Seite die *Bottom-Up-Methode*, bei der von der feinen Sicht zur groben Sicht modelliert wird.
Die *Top-Down-Methode* führt zuerst zu einem Überblick über das Problem ohne die Details zu berücksichtigen. Die einzelnen Teilbereiche werden sukzessive erweitert und detaillierter beschrieben, bis das System vollständig erfasst ist. Der Schwerpunkt liegt auf der Planung und dem Verständnis eines Problems. Diese Methode bietet sich zum *Business Process Reengineering* an.
Im Gegensatz dazu beginnt die *Bottom-Up-Methode* zuerst mit der Modellierung und Spezifizierung der Details um die Sicht schrittweise zu vergröbern, bis das System vollständig beschrieben ist. Diese Methode bietet sich beim *Continuous Process Improvement* an.

6.2 Typisierung der Prozessarten

Anhand von Prozessvariablen, welche als Eigenschaften von Geschäftsprozessen betrachtet werden können, kann eine Klassifizierung von Geschäftsprozessen zu bestimmten Typen vorgenommen werden. Zu diesen Prozessvariablen eines Geschäftsprozesses zählen dessen Komplexität, der Grad der Veränderlichkeit, der Detaillierungsgrad, der Grad der Arbeitsteilung sowie die Interprozessverflechtung. Je nach Ausprägung und Kombination dieser Prozessvariablen lassen sich insgesamt grob drei Prozessklassen unterscheiden: Der Routineprozess, der Regelprozess sowie der einmalige Prozess. [40]

Routineprozess Der Routineprozess ist dabei durch weitgehend klare Strukturen, leichte Planbarkeit, standardisierte Abläufe und einen hohen Grad an Arbeitsteilung gekennzeichnet. Im Bereich der Produktionswirtschaft liegt er häufig der Produktion von Massenfertigungen mit geringer Variantenvielfalt zu Grunde.

Regelprozess Regelprozesse besitzen genau wie Routineprozesse noch kontrollierbare Strukturen, weisen jedoch eine erhöhte Komplexität auf. Weiterhin bedarf er der häufigen individuellen Veränderung durch Mitarbeiter, allerdings bei vorhersagbarer Variation. Die im Gegensatz zur Massenfertigung komplexere Einzelfertigung eines Produktes unter Berücksichtigung von Sonderwünschen wäre ein Beispiel aus der Produktionswirtschaft für einen Regelprozess.

Einmaliger Prozess Einmalige Prozesse besitzen eine nicht zu überschauende Komplexität und müssen individuell bearbeitet werden. Prozessablauf und Kommunikationspartner sind nicht planbar und entziehen sich einer Automatisierung. Ein solcher Prozess wird oft von einzelnen Teams oder Mitarbeitern bearbeitet, die über Funktions- und

Abteilungsgrenzen hinweg kommunizieren. Die Erstellung von Individualsoftware ist ein Beispiel für einen Einmaligen Prozess.

6.3 Darstellungsformen von Geschäftsprozessen

Zur Darstellung von Geschäftsprozessen haben sich die erweiterten Ereignisgesteuerten Prozessketten (eEPK) durchgesetzt. Das Konzept wurde Scheer [46] vorgestellt und basiert auf dem Konzept der Petrinetze.

6.3.1 Petrinetze

Petrinetze sind gerichtete bipartite Graphen mit zwei Elementen: (1) Stellen oder places, grafisch als Kreise dargestellt und (2) Transitionen oder transitions, grafisch als Quadrate dargestellt. Die beiden Elemente werden durch gerichtete Pfeile miteinander verbunden. Es können nur Stellen mit Transitionen und Transitionen mit Stellen verbunden werden. Die Verbindung von gleichartigen Elementen ist nicht möglich.
Das Petrinetz ist statisch. Stellen können veränderliche Marken oder tokens enthalten. Marken werden durch das schalten oder firing von Transitionen geschaltet. Durch das Schalten zieht eine Transition eine Marke von jedem Vorgänger ab und führt eine Marke jedem Nachfolger zu. Nur eine aktive Transition kann schalten. Die Transition wird aktiviert wenn jeder Vorgänger wenigstens eine Marke enthält. Transitionen sind die einzigen aktiven Elemente eines Petrinetzes, Marken die einzigen dynamischen Elemente. Stellen sind als statische und nicht aktive Elemente die Representation von Zuständen, Bedingungen und Speichern, während Transitionen die Aktivitäten darstellen. Die Verteilung der Marken stellt den aktuellen Zustand des Netzes dar.

Die formalen Grundlagen der Petrinetze werden hier nicht weiter betrachtet. Das Routing in Petrinetzen kann auf verschiedene Arten erfolgen:

- sequentielles Routing
- paralleles Routing
- bedingtes Routing
- iteratives Routing
- rekursives Routing

Beim sequentiellen Routing werden die Aufgaben nacheinander ausgeführt. Dies stellt die einfachste Form dar. Beim parallelen Routing werden voneinander unabhängige Teilaufgaben nebenläufig ausgeführt. Das bedingte Routing führt Aufgaben nur aus, wenn die erforderlichen Bedingungen erfüllt sind. Eine wiederholte Ausführung, bzw ein modellierter Kreis bei dem Teilaufgaben wiederholt ausgeführt werden, wird als iteratives Routing bezeichnet. Das rekursive Routing ist dem iterative Routing ähnlich, aber bei jeder Iteration wird eine neue Marke erzeugt. Beispielhaft werden iteratives (Abb. 8) und paralleles (Abb. 9) Routing dargestellt.

6.3.2 Ereignisgesteuerte Prozessketten (EPK)

EPK sind grafische Zustandsdiagramme auf Basis von Petrinetzen zur Darstellung von Geschäftsprozessen und Arbeitsabläufen. Die Symbole zur grafischen Darstellung sind Abb. 10 dargestellt. EPK stellen Arbeitsprozesse in einer semiformalen Modellierungssprache grafisch dar. Dadurch können Abläufe systematisiert und parallelisiert werden. Dazu werden Objekte in gerichteten Graphen mit gerichteten Kanten in einer 1:1-Zuordnung verbunden.Logische Verknüpfungen weisen allerdings 1:N-Zuordnungen

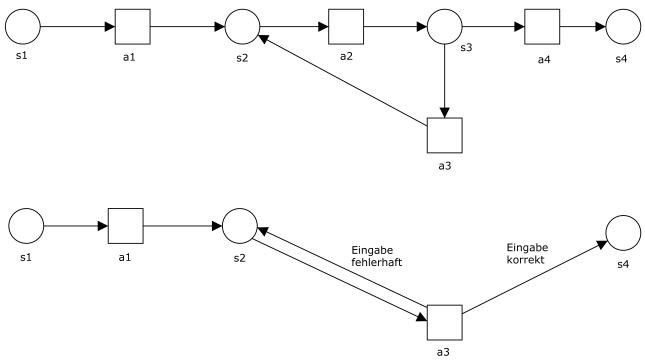


Abbildung 8: Iteratives Routing [*Quelle: [30]*]

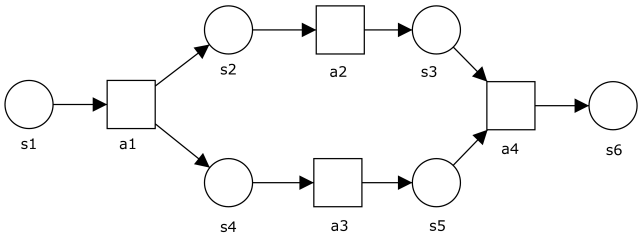


Abbildung 9: Paralleles Routing [*Quelle: [30]*]

auf. In einer solchen Prozesskette wechseln die Objekte sich in ihrer Bedeutung zwischen Ereignis und Funktion ab, d.h. sie bilden eine alternierende Folge, die zu einem bipartiten Graphen führt. Jede Funktion kann zusätzlich mit einem Informationsobjekt verbunden werden, aus dem Informationen geladen oder in das Informationen gespeichert werden.

Die Knoten in einem EPK beschreiben:

- Funktionen (dargestelltdurch abgerundete Rechtecke)
- Ereignisse oder Zustände (dargestellt durch Sechsecke)
- Verknüpfungsoperatoren oder Konnektoren (dargestellt durch Kreise) zur Verbindung von Funktionen und Zuständen

Funktionen und Zustände folgen immer abwechselnd aufeinander, wobei Funktionen von einem Zustand oder mehreren Zuständen ausgelöst werden und einen oder mehrere Zustände erzeugen. Umgekehrt kann ein Zustand von einer Funktion oder mehreren Funktionen ausgelöst werden und eine oder mehrere Funktionen auslösen. Mit den Verknüpfungsoperatoren werden alternativ die logischen Verknüpfungen **UND**($\wedge$), **inklusives ODER**($\vee$) und **exklusives ODER** (XOR) zwischen auslösenden und erzeugten Funktionen und/oder Zuständen zum Ausdruck gebracht. Wird der Ablauf durch einen Konnektor aufgeteilt, so muss der Ablauf durch den gleichen Konnektor wieder zusammengeführt werden. Ereignisse können keine Entscheidung treffen, deshalb kann auf ein Ereignis nur eine Konjunktion folgen. Ein EPK beginnt immer mit einem Ereignis und Endet mit einem Ereignis. Nur Funktionen können Zeiten und Kosten zugeordnet werden. Zur Reduktion der Komplexität können Teilprozesse definiert werden. Hierbei ist darauf zu achten, dass Teilprozesse nur auf ein Ereignis folgen können.[52] [48] [47]

Die erweiterte Ereignisgesteuerte Prozesskette erweitert das Modell um Elemente der Organisations-, Daten und Leistungsmodellierung. Für jedes dieser Elemente gibt es eine

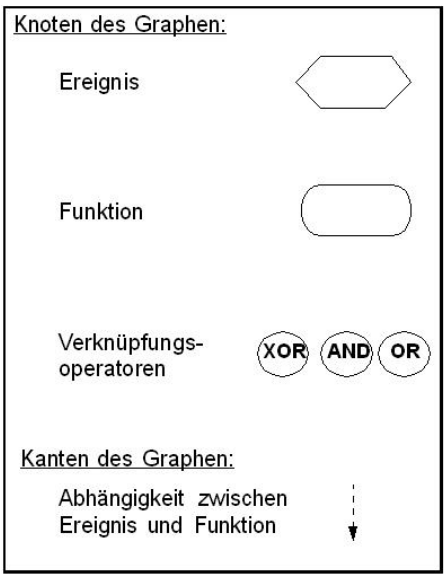


Abbildung 10: Grafische Darstellung von EPK

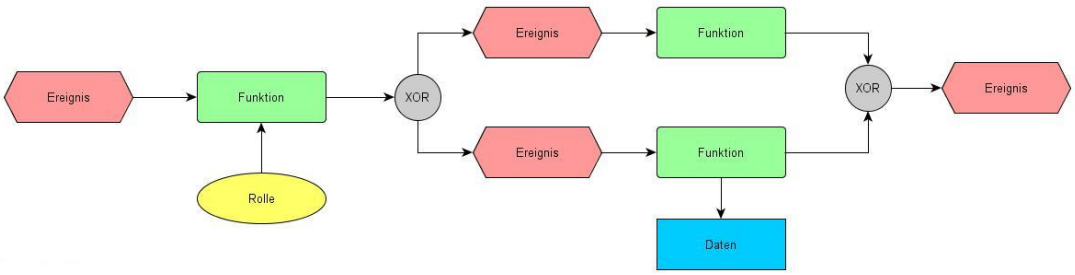


Abbildung 11: Beispiel für die grafische Darstellung von erweiterten Ereignisgesteuerten Prozessketten (eEPK).

gebräuchliche Darstellungsform (Organigramm, ERM, SERM, etc). Die eEPK führt als logisches und steuerndes Modell alle diese Sichten auf die Elemente in einem Prozess zusammen. Eine beispielhafte erweiterte Ereignisgesteuerte Prozesskette ist in Abbildung 11 dargestellt.

6.4 Geschäftsprozesse im Smarter Wohnen Umfeld

Dienstleistungen, die innerhalb des *SmarterWohnen*-Projektes erbracht werden, zeichnen sich durch eine strukturierte Folge von Teilaufgaben bzw. Aktivitäten aus, deren Gesamtheit auf die Erbringung eines Ziels ausgerichtet ist. Hierbei handelt es sich um die Bereitstellung von IT-Mehrwertdiensten unter Zusicherung bestimmter Qualitätsanforderungen, deren Einhaltung maßgeblich für den Erfolg der Erbringung einer vordefinierten Leistung ist. Weiterhin stehen als ausführende Resourcen einerseits Mitarbeiter von Hausverwaltung und beauftragten externen Dienstleistern bereit, andererseits handelt es sich dabei um Komponenten der Hausinfrastruktur, Funktionen des *Service-Gateways* sowie funktionale Einheiten der *Service-Plattform*. Die Ausführung von Aktivitäten kann abhängig von der Art des Dienstes sowohl sequentiell als auch nebenläufig erfolgen und wird durch externe und interne Ereignisse beeinflusst. Somit stellt zumindest der nicht ausnahmenbehaftete Verlauf einer im *SmarterWohnen*-Projekt bereitgestellten Dienstleistung

einen als Regelprozess typisierten Geschäftsprozess dar. Doch gerade die Berücksichtigung von Eskalationsstufen innerhalb des Prozessdesigns von kritischen Echtzeit-Prozessen, wie beispielsweise der Brandmeldung[14] als angebotene Dienstleistung, unterwerfen diese Geschäftsprozesse einer signifikanten Erhöhung ihrer Komplexität. Gerade dieser Aspekt ist bei der Formulierung von Aussagen in Bezug auf den Vergleich der Implementierung der *SmarterWohnen*-Dienste als Prozess im Gegensatz zu ECA-Regelsystemen (siehe hierzu Abschnitt 4) von maßgeblicher Bedeutung.

7 jABC

7.1 Einleitung

Zur Modellierung der Geschäftsprozesse wird das *Java Application Building Center* genutzt. Mit diesem Werkzeug lassen sich Prozesse aus einzelnen Komponenten zu Graphen zusammenbauen. Im Gegensatz zu anderen Modellierungskonzepten wie UML können jABC-Graphen direkt ausgeführt und in Programmcode übersetzt werden, so dass die modellierte Geschäftslogik stets mit dem ausgeführten Programm übereinstimmt.

7.2 Lightweight Process Coordination

Dem jABC liegt das Konzept des *Lightweight Process Coordination* (LPC [24]) zu Grunde. Dessen Idee ist es, eine grafische Koordinationsschicht zu dem bestehenden 3-Schichten-Modell hinzuzufügen (vgl. Abbildung 12). Mit Hilfe des jABC können Anwendungen ohne tiefgehende Programmierkenntnisse aus Komponenten (SIBs) grafisch modelliert und ausgeführt werden. Die Programme werden als hierarchische Graphen dargestellt. Durch die grafische Modellierung unterteilt sich die Anwendungsentwicklung in zwei Bereiche, die (grafische) Modellierung der Anwendungslogik durch einen Anwendungsexperten und die Programmierung der grundlegenden SIBs durch einen Java-Programmierer.

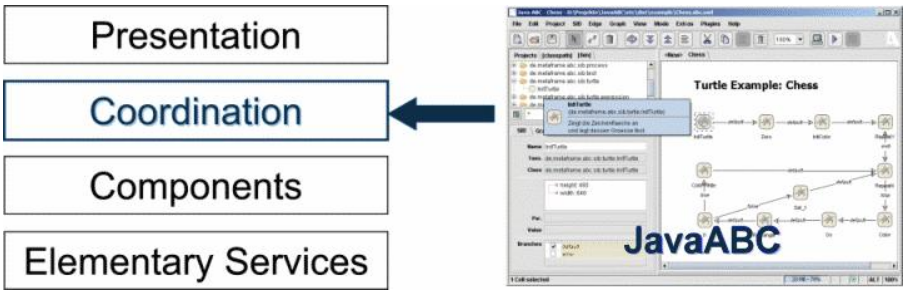


Abbildung 12: Die Koordinationsschicht [24]

7.3 Service Independent Building Block (SIB)

7.3.1 Definition

Service Independent Building Blocks (SIBs) sind wiederverwendbare Komponenten im jABC und werden als Knoten in dem SIB-Graphen abgebildet. Sie stellen die Implementierungsebene in der Anwendungsentwicklung mit dem jABC dar. Der Anwendungsexperte definiert (ohne Programmierkenntnisse) funktionale Anforderungen und ausgehende Kanten (Branches) an einen SIB, der dann von einem SIB-Experten unabhängig implementiert

[14] vgl. Abschnitt 8.1.4

werden kann. Je nach Anforderungen der Anwendung können einzelne SIBs sehr kleine Aufgaben übernehmen, um eine möglichst große Wiederverwendbarkeit zu erreichen. [24]

7.3.2 Aufbau

SIBs sind Java Klassen mit einigen besonderen Eigenschaften, die mit jeder Java Entwicklungsumgebung erstellt werden können, vorzugsweise mit Eclipse, da dieses eng ins jABC integriert ist. Jeder SIB wird durch eine `SIBClass`-Annotation (aus dem Paket `de.metaframe.common.sib.annotation`) gekennzeichnet; diese enthält einen weltweit eindeutigen String zur Identifizierung:

Listing 1: `SIBClass`-Annotation

```
@SIBClass("81d91b1c:2d393839313031353038:1158657043751")
public class ExampleSIB {
}
```

Parameter Die Parameter eines SIBs können vom Anwender verändert werden. Als Parameter gelten alle `public`-Variablen in einer SIB-Klasse. Sie sollten einen aussagekräftigen Namen tragen und müssen initialisiert werden. Gültige Variablentypen für Parameter sind `Boolean, Byte, Double, Float, Integer, Long, Short, String, ArrayList, LinkedList, Vector, Hashtable, HashSet, LinkedHashSet, TreeSet` und `TreeMap`. [24]

Branches Branches sind die Bezeichner ausgehender Kanten an SIBs, über die im SIB-Graphen SIBs miteinander verbunden werden. Neben den zwei Standard-Branches `default` und `error` können weitere in Form von String-Arrays definiert werden, entweder fest oder vom Benutzer veränderbar.
Feste Branches:

```
public final String[] BRANCHES = {"default", "error"};
```

Variable Branches:

```
public String[] branches= {"default", "error"};
```

Diese beiden Varianten können auch kombiniert verwendet werden. [24]

Onlinehilfe Das jABC Hilfesystem verwendet Tooltips zur Erklärung von SIBs. Diese Onlinehilfe wird über die Funktion `getToolTipText(String category, String param)` (siehe Quelltext) in der SIB-Klasse festgelegt. Die GUI unterscheidet die Tooltips in die drei Kategorien `TOOLTIP_SIB, TOOLTIP_BRANCH, TOOLTIP_FIELD`, für die unterschiedliche Onlinehilfen angezeigt werden können. [24]

Listing 2: Onlinehilfen

```
public String getToolTipText(String category, String param) {
  // Kategorie TOOLTIP_SIB
  if (category.equals(TOOLTIP_SIB))
    return "";
  // Kategorie TOOLTIP_BRANCH
  if (category.equals(TOOLTIP_BRANCH))
    return "";
  // Kategorie TOOLTIP_FIELD
  if (category.equals(TOOLTIP_FIELD))
    return "";
```

```
  return null;
}
```

Anstatt der Funktion `getToolTipText(String category, String param)` kann auch die Funktion `getDocumentation(String category, String param)` benutzt werden, die die gleiche Funktionalität bietet.

7.3.3 ProxySIB und GraphSIB

`GraphSIB` ermöglicht die hierarchische Einbettung von SIB-Graphen als SIB in andere Graphen. Das jABC nutzt `ProxySIB` als Platzhalter für SIBs, die nicht verfügbar sind. Damit bleiben alle Eigenschaften und Branches erhalten. Jeder SIB kann selbst ein SIB-Graph sein und aus mehreren SIBs bestehen, und jeder SIB-Graph kann als ein SIB in einen anderen Graphen eingefügt werden. Damit unterstützt das jABC sowohl den Top-Down- als auch den Bottom-Up-Ansatz. [24]

7.4 Plugins

Das jABC lässt sich durch Plugins im Funktionsumfang erweitern. Dieser modulare Gedanke ist stark in der Architektur verankert, so dass selbst wichtige Funktionen wie die Ausführungsebene (Tracer) als Plugin implementiert sind. Dies erlaubt eine weitgehende Freiheit und Erweiterbarkeit. Eine wichtige Funktion der Plugins ist es, dem SIB-Graphen eine Semantik zu verleihen.

7.4.1 Tracer

Das standardmäßig installierte Plugin Tracer dient der Ausführung von jABC-Modellen. Die steuernde Klasse `ExecutionController` nimmt eine Menge von SIB-Graphen mit den zugehörigen SIBs entgegen und führt diese ähnlich wie eine Entwicklungsumgebung aus. Der Vorgang kann schrittweise fortgesetzt oder durch Breakpoints bei der Erreichung eines bestimmten SIBs angehalten und beobachtet werden. Die eigentliche Ausführung (Abb. 13) wird von einem oder mehreren `ExecutionThreads` durchgeführt. Zur Vermittlung zwischen den verschiedenen Kontexten und zum Debugging dient die Klasse `ExecutionEnvironment`. Neben der Ausführung lokaler Graphen ist es auch möglich, Modelle aus anderen virtuellen Maschinen beziehungsweise auf anderen Hosts auszuführen und zu beobachten. [10]

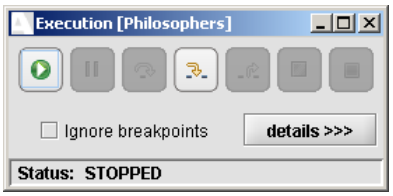


Abbildung 13: Tracer-Dialog

Ausführung von SIB-Graphen Um einen SIB-Graphen mit Tracer ausführen zu können, muss ein Start-SIB vorhanden sein und jedes auszuführende SIB ein dem `ExecutionController` bekanntes Interface implementieren, normalerweise `de.metaframe.abc.sib.Executable`, das die Funktion `trace` enthält. Ein Start-SIB darf keine eingehenden Kanten, muss aber mindestens eine ausgehende Kante besitzen. Nach der Ausführung dieses SIBs wird der Nachfolger-SIB an Hand des Branches (z.B. über die

Kante mit der Beschriftung `error` im Fehlerfall) bestimmt und die Ausführung an diesem SIB fortgeführt. [10]

ExecutionEnvironment Die trace-Methode erhält als Parameter eine ExecutionEnvironment. Diese ExecutionEnvironment bietet SIBs die Möglichkeit, untereinander über einen gemeinsamen Speicherbereich zu kommunizieren. Diese Umgebung stellt eine Map und passende Funktionen bereit, in der SIBs `Object`-Variablen mit der Funktion `put(String name, Object variable)` ablegen können, die von anderen SIBs über die Funktion `get(String name)` ausgelesen werden.

7.4.2 LocalChecker

LocalChecker prüft lokale Bedingungen von SIBs und zeigt gegebenenfalls auftretende Probleme in den vier Stufen Info, Warnung, Fehler und fataler Fehler im Plugin-Inspektor an. Zu überprüfende SIBs müssen von sich aus *LocalChecker* unterstützen. Der Zustand (`OK` oder einer der vier Problemzustände) eines SIBs wird direkt im SIB-Graphen als Icon angezeigt. Im Info-Fenster des Plugins findet man zusätzlich eine Beschreibung zu jedem geprüften SIB. [37]

7.4.3 GEAR

GEAR ist ein *Model Checking*-Werkzeug. *Model Checking* ermöglicht die automatische formale Verifikation von Graphen anhand von Formeln. Diese Formeln können in den Logik-Sprachen CTL und μ-Kalkül formuliert sein und bilden umgangssprachliche Bedingungen formal ab. Zur Beschreibung der Formeln nutzt *GEAR* das Plugin FormulaBuilder. *GEAR* ist vollständig in das jABC integriert, lässt sich aber auch als eigenständiges Programm nutzen. [45]

7.4.4 CodeGenerator

Das Plugin *CodeGenerator* ermöglicht die Übersetzung von SIB-Graphen in ausführbaren Code. Durch die Unterstützung verschiedener Generatoren ist das Plugin programmiersprachenunabhängig. Derzeit stellt es Generator-Klassen bereit, um SIB-Graphen in Java-Klassen, Java HTTP Servlets, Java-Klassen, die den SIB-Graphen rekonstruieren und ausführbare SIBs umzuwandeln. Letzteres ist nützlich, um Teilgraphen zu einem einzigen SIB zusammenzufassen.
Der *CodeGenerator* setzt voraus, dass jeder SIB im zu übersetzenden Graphen das Interface `CodeGeneratorBlock` und die damit verbundene Methode `execute` implementiert. Das Plugin unterstützt für jeden SIB-Graphen mehrere Generator-Konfigurationen, in denen der zu verwendene Generator und Eigenschaften wie der Paketname der erstellten Klasse, der Zieldateiname und eine Beschreibung gespeichert werden (Abb. 14). [33]

7.4.5 SIBCreator

Mit *SIBCreator* lassen sich zur Laufzeit des jABC SIB-Komponenten erstellen und zum SIB-Graphen hinzufügen. Dies kann vom Anwendungsexperten erfolgen und erfordert keine Programmierkenntnisse. Der Anwendungsexperte erstellt ein leeres SIB ohne Funktion, das später von einem SIB-Experten implementiert werden kann. Dazu werden im Plugin-Dialog von *SIBCreator* der Name des SIBs (inklusive Paketstruktur), die Parameter und die ausgehenden Branches definiert. Ein Parameter wird über eine Schaltfläche hinzugefügt und sein Typ aus einer Liste ausgewählt. Für die ausgehenden Kanten des SIBs legt der Anwender fest, ob diese `final` (also unveränderbar) oder vom Benutzer änderbar sein

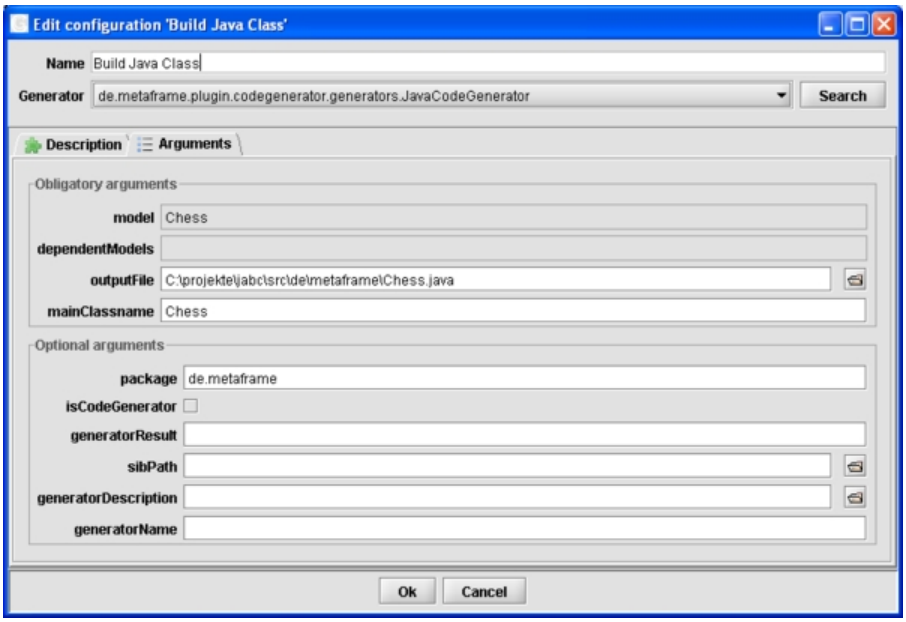


Abbildung 14: *CodeGenerator* Konfiguration

sollen. Optional kann auch die Onlinehilfe in Form der oben genannten Tooltips ergänzt werden. Das Plugin erstellt eine SIB-Klasse, die der Anwender direkt in seinen Graphen integrieren kann. [38]

7.4.6 TaxonomieEditor

Der *TaxonomieEditor* ermöglicht neben der vom Dateisystem gegebenen Ordnung der SIBs eine weitere „logische Sicht auf eine Menge von SIBs“ [9], die vom Benutzer verändert werden kann. Analog zum Dateisystem werden die SIBs in hierarchischen Ordnern verwaltet, können aber statt ihrem Dateinamen auch einen Alias tragen. Nach Aktivierung des Plugins kann im SIB-Browser die Ansicht zwischen Dateisystem und Taxonomie getauscht werden (Abb. 15). Mit dem *TaxonomieEditor* wird die Taxonomie grafisch be-

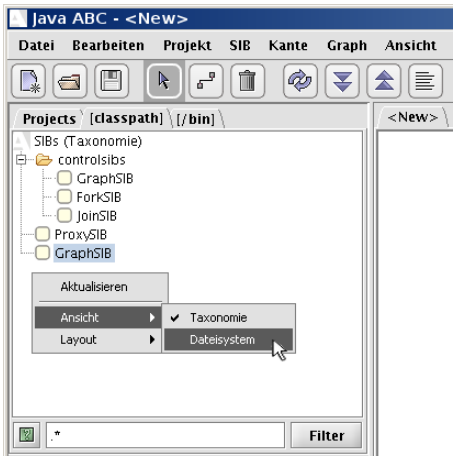


Abbildung 15: Taxonomie-Ansicht

arbeitet. In einer zweigeteilten Ansicht stellt er die SIBs im Dateisystem-Baum links und den Taxonomie-Baum rechts dar, der Benutzer kann in der Taxonomie sehr einfach Ordner erstellen und die SIBs sortieren und benennen, wie er es aus einem Dateisystem-Browser

gewohnt ist.
Jeder SIB-Pfad kann genau eine Taxonomie enthalten. Diese wird in einer XML-Datei gespeichert, die neben den Ordnern die SIB-Namen (beziehungsweise den vergebenen Alias) und die zugehörige Universal ID enthält. Die XML-Datei bekommt der Anwender aber nicht zu sehen, er benötigt nur den grafischen TaxonomieEditor. [9]

Teil III
Konzeption und Umsetzung

8 Fachliches Gesamtkonzept

8.1 Analyse der SmarterWohnen-Abläufe

Um die Abläufe der *SmarterWohnen*-Prozesse in eine Prozesssicht überführen und diese im *jABC* implementieren zu können, müssen diese zunächst analysiert werden. Die Vorgehensweise und die dabei angewendeten Analysemethoden werden in Abschnitt 8.1.2 erläutert. Im weiteren Verlauf wird unter Verwendung der Analyseergebnisse in Abschnitt 8.2 eine Anforderungsanalyse durchgeführt, in der Prozessteile als *SIBs* identifiziert werden.

8.1.1 Überblick

Die organisatorischen und funktionalen Abläufe verschiedener Dienstleistungen, die innerhalb des Projektes *SmarterWohnen* durch das Fraunhofer ISST in Kooperation mit internen und externen Dienstleistern erbracht werden, liegen im Rahmen der Projektdokumentation als Prozessketten vor. Als informationslogistische Komponente und zentrale Infrastrukturlösung auf oberster Ebene fungiert die *Service-Plattform*, auf der die genannten prozessorientierten Abläufe bereitgestellter Services implementiert sind. Sämtliche Abläufe sind auf Seite der *IGM-Plattform* durch ECA-Regeln umgesetzt. Das *EVA-Subsystem* setzt die Ereignisbehandlung und die Ablauflogik der *SmarterWohnen*-Dienste um. Die Daten und Funktionalitäten der *IGM-Plattform* stehen der Projektgruppe zur Umsetztung dieser Abläufe als Geschäftsprozesse im *jABC* zur Verfügung.

Die vom Fraunhofer ISST als Prozessabläufe durchgeführte Modellierung und Dokumentation der Dienstleistungen liegt in zwei verschiedenen Sichten vor: Während die *fachliche Sicht* global den Ablauf von Prozessen aller beteiligten Akteure beschreibt, also auch externe Prozessschritte, welche die *IGM-Plattform* nicht direkt betreffen, modelliert die *technische Sicht* IGM-spezifische Prozessabläufe detaillierter; hierbei werden Methodenaufrufe der Plattform und beteiligter Infrastrukturkomponenten verschiedener Akteure konkretisiert und externe Prozessschritte ausgeblendet. ([14], [15], [16], [17] und [18])

Die in der Dokumentation beschriebenen Prozesse lassen sich in *Event-basierte* und *Content-basierte* Prozesse kategorisieren. Event-basierte Prozesse und deren Prozessketten werden durch Events initiiert, wie sie beispielsweise von Sensoren zur Brand- und Einbruchserkennung erzeugt werden. Content-basierte Prozesse werden bei der Realisierung von Dienstleistungen zum Deployment von zuvor durch Benutzer abonnierten Informationen (Content) eingesetzt; die Bereitstellung und Aufbereitung von Informationen erfolgt dabei periodisch, während der Abruf der innerhalb einer Periode statischen Informationen durch den Bewohner aktiv über ein User-Interface erfolgt. Während Event-basierte Prozesse sowohl fachlich als auch technisch dokumentiert sind, existiert für Content-basierte Prozesse nur die technische Beschreibung.

Da Content-basierte Prozesse lediglich den Austausch von statischen Informationen zwischen *Service-Gateway* und *Service-Plattform* realisieren, sind sie nur unzureichend als komplexe Prozessketten zu realisieren. Daher verbleibt die Implementierung auf der

IGM-Plattform und wird nicht im *jABC* abgebildet. Es werden daher ausschließlich Event-basierte Prozesse im *jABC* implementiert. Zu den Content-basierten Diensten gehören z.B. der Menüservice, Kalenderdienste, Bundesligaergebnisse, Wetterinformationen oder Apothekeninformationen.

8.1.2 Beschreibung des analytischen Vorgehens

In den folgenden Abschnitten werden ausgewählte Beispiele Event- und Content-basierter Prozesse beschrieben. Als Grundlage zur Analyse dieser Prozesse dient die vom Fraunhofer ISST als Prozessdokumentation[15] bereitgestellte fachliche und technische Sicht der *SmarterWohnen*-Abläufe. Diese Informationen werden zusammengetragen und zunächst auf solche Aspekte reduziert, welche die reinen Prozessabläufe der *Service-Plattform* betreffen; auch die Interaktion mit den im Prozessverlauf involvierten Akteuren wird auf die technische Sichtweise beschränkt, sodass Schnittstellen und Prozessschritte, die außerhalb der Interaktionslinie der Plattform liegen, nicht betrachtet werden, da sie für die Implementierung der Prozesslogik nicht relevant sind. Auch die Kommunikation mit Akteuren wie z. B. mit Bewohnern und Dienstleistern wird auf die tatsächlich technisch zur Verfügung stehenden Kommunikationsmedien und User-Interfaces reduziert. Kommunikation erfolgt hierbei ausschließlich durch Nachrichten, die von IGM-Subsystemen über verschiedene Kommunikationsmedien versendet werden. Grundsätzlich wird in der initialen Analysephase ein Top-Down-Ansatz angewendet. Als graphische Darstellungsform der Prozesse werden erweiterte ereignisgesteuerte Prozessketten (eEPK) verwendet. Diese bieten über Methoden zur Darstellung der reinen Prozesslogik hinausgehende Eigenschaften wie Organisations-, Daten- und Leistungsmodellierung. Auf diese Weise werden die vom Fraunhofer ISST erhaltenen Dienstbeschreibungen in eine Prozesssicht überführt und mit Zusatzinformationen zu Ein- und Ausgabedaten verschiedener Prozessschritte versehen. Außerdem werden Informationsquellen und -senken wie IGM-Subsysteme und Datenbanken identifiziert und dokumentiert.

Um die gegenseitige Beeinflussung der *SmarterWohnen*-Prozesse untereinander bewerten und Aussagen über die Komplexität des Zusammenspiels der einzelnen Prozesse in einem Gesamtkontext formulieren zu können, werden für jede Prozesskette Vorbedingungen und Nachbedingungen analysiert. Während Vorbedingungen Zustände beschreiben, in denen sich das Gesamtsystem befinden muss, um einen Prozess starten zu können, stellen Nachbedingungen Zustandsbeschreibungen dar, in denen sich das System nach Ausführen eines Prozesses befinden muss. Bei Event-basierten Prozessen werden außerdem auslösende Ereignisse identifiziert, die zur Initiierung dieser Prozesse führen. Prozesse lösen Ereignisse aus und werden durch diese beeinflusst. Somit kann auf Basis der Identifizierung von auslösenden und ausgelösten Ereignissen aller beteiligten Prozesse eine Relation modelliert werden, welche die Abhängigkeiten der Prozesse hinreichend darstellt und bei der späteren Modellierung im *jABC* verwendet werden wird.
Ein Beispiel für die gegenseitige Beeinflussung zweier Prozesse sind die Methoden *Brandmeldung* und *Brandbeendigung*. Während der durch ein Brandereignis initiierte Prozess *MeldeBrand* alle notwendigen Schritte von der Benachrichtigung von Dienstleistern wie beispielsweise der Feuerwehr und Bewohnern bis zur Protokollierung vornimmt, beendet der Prozess *Brandbeendigung* einen offenen Brandnotfall. Soll nun während der Abarbeitung der *MeldeBrand*-Methode bzgl. einer Wohnung genau dieser Notfall durch den Aufruf der Methode *Brandbeendigung* beendet werden (beispielsweise bei einem Fehlalarm), so wird hierdurch die Bearbeitung des Prozesses *MeldeBrand* maßgeblich beeinflusst.

[15] vgl. [14], [15], [16], [17] und [18]

Sowohl die Modellierung von Prozessen im *jABC* als auch die Darstellungsform der erweiterten Ereignisgesteuerten Prozessketten ermöglicht die Hierarchisierung von (Teil-) Prozessen auf verschiedenen Detaillierungsebenen. Zwischen Prozessketten lässt sich somit eine baumartige Eltern-Kind-Relation identifizieren, welche in Form von Eltern- und Kindprozessketten darstellbar ist. Daher wurden für alle analysierten Prozesse aufrufende Elternprozesse sowie aufgerufene Kindprozesse dokumentiert. Die Analyse dieser Prozessstruktur lässt sich bei der Modellierung im *jABC* nutzen und ermöglicht somit eine hierarchisierte Darstellung und Schachtelung.

Weiterhin wurden für jeden analysierten Prozess wichtige Aspekte des Prozesslebenszyklus herausgestellt. Hierbei ist die gesamte Ablaufdauer von Prozessstart bis zur Prozessbeendigung von besonderer Bedeutung. Hierbei ist nur die Größenordnung der Ablaufdauer interessant. Diese sind bei der Planung und Implementierung von Erweiterungen zur Laufzeitumgebung der *jABC-Plattform* notwendig, da Methoden zur Aktivierung, Deaktivierung und Persistenz von Prozesstasks erarbeitet werden mussten, um eine möglichst performante Realisierung der Prozesssteuerung auf Orchestrierungsebene zur Laufzeit gewährleisten zu können.

Für jeden Prozess wurde außerdem eine ausformulierte Prozessbeschreibung angefertigt, um neben der graphischen formellen Darstellung in Form von Ereignisgesteuerten Prozessketten eine detaillierte Ablaufbeschreibung zu erhalten [41].

Um die Prozesse im *jABC* als weiteren Schritt dieser Projektgruppe implementieren zu können, war eine genaue Kenntnis über Ein- und Ausgabeparameter der einzelnen Prozessschritte notwendig. Daher wurden für alle untersuchten Prozesse verwandte Parameter identifiziert und - sofern in diesem Analyseschritt möglich - Quellen und Senken zugeordnet. Diese somit gewonnenen Informationen flossen u. a. in die in Abschnitt 9.1.3 entwickelte Parametertaxonomie ein.

Diese mehrschrittige Analyse wird in 8.1.4 anhand des Brandmeldeprozesses ausgiebig demonstriert. Die Abläufe bei allen weiteren Notfallmeldungen sind ähnlich. Die Plattformabläufe werden im Folgenden kurz dargestellt.

8.1.3 Rahmenbedingungen im SmarterWohnen Projekt

Im Allgemeinen wird bei den event-basierten Notfallmeldungen zuerst eine Prüfung der durch das *Service-Gateway* übergebenen Parameter durchgeführt, wobei aber nur auf Vollständigkeit und den korrekten Wertebereich geprüft wird. Die notwendigen Daten über die Wohnung, den Bewohner, die zuständigen Dienstleister und die Konfiguration des entsprechenden Dienstes werden aus der Datenbank geladen. Dienstleister, Bewohner, Angehörige oder Nachbarn werden dann per Telefon, SMS oder Email benachrichtigt. Die erbrachte Leistung wird in einem Abrechnungsprotokoll festgehalten und die durchgeführten Maßnahmen werden in dem Systemprotokoll gespeichert. Wird eine Alarmmeldung beendet, wird in einem ähnlichen Ablauf eine Benachrichtigung an die Betroffenen versandt. Die Alarmsituation kann sowohl durch den Bewohner in der Wohnung beendet werden, als auch durch den Dienstleister, nachdem ein Alarm als Fehlalarm erkannt wurde oder geeignete Maßnahmen ergriffen wurden. Außerdem ist das *Service-Gateway* prinzipiell auch in der Lage einen aktiven Alarm zu beenden, genauso wie die Administration des Systems einen Alarm, z.B. im Fehlerfall, beenden kann.

Ein Notfall wird durch die Reaktion eines Sensors auf seine direkte Umwelt ausgelöst. Die verbauten Brand-, Gas- und Wassersensoren lösen einen entsprechenden Alarm aus. Neben der direkten Reaktion in der Wohnung, wie z.B. ein akustischer Warnton, werden die Daten durch das *Service-Gateway* an die *IGM-Plattform* übermittelt. Hierbei wird die eindeutige Identifikation des *Service-Gateways* und der Wohnung übermittelt, aber auch die meldende Komponente und der Raum in der Notfall erkannt wurde. Die Benachrichtigung von Bewohnern, Angehörigen und Dienstleistern wird in 8.1.4 beschrieben.

Die verbauten Türkontakte, Bewegungsmelder und Glasbruchmelder können einen Einbruchsalarm auslösen. Hierbei muss aber berücksichtigt werden, ob der Bewohner zuhause ist und Türen und Fenster öffnet oder ob die Wohnung verlassen ist. Der Ablauf des Meldungsprozesses ist den oben beschriebenen ähnlich. Die Sensorik, die zur Einbruchsdetektion verbaut ist, wird aber auch in entgegengesetzter Richtung benutzt. Die Bewegungsmelder werden eingesetzt, um zu erkennen, ob sich ein Bewohner in seinem üblichen Tagesrhythmus in seiner Wohnung bewegt. Wird keine Bewegung in der Wohnung festgestellt, obwohl innerhalb eines bestimmten Zeitintervalls eine Bewegung festgestellt werden sollte, dann wird ein Aktivitätsalarm ausgelöst. Hierbei ist zu beachten, dass sich dieser Dienst nur bei einem sehr regelmäßigen Tages- und Nachtrhythmus nutzen lässt. In einer Wohnung mit Haustieren kann der Dienst nicht sinnvoll genutzt werden. Nachdem ein Aktivitätsalarm ausgelöst wurde, wird ein entsprechender Alarmplan abgearbeitet. Innerhalb dieses Alarmplans wird wohl normalerweise zuerst versucht den Bewohner zu erreichen und eine Reaktion von ihm zu erhalten, ob tatsächlich ein Notfall vorliegt, bevor ein externer Dienstleister, Angehörige oder Nachbarn benachrichtigt werden. Im Grundzug ähnelt der Alarmplan aber dem Ablauf der bereits genannten Notfälle.

Ist ein Bewohner innerhalb seiner Wohnung in eine Notlage geraten, so kann er über Notfallknöpfe Hilfe rufen. Die Notfallknöpfe könne fest eingebaut sein, z. B. im Badezimmer, oder tragbar, z. B. in einer Uhr sein. Letztere werden auch als elektronische Finger bezeichnet. Das *Service-Gateway* sendet diesen Serviceruf an die *IGM-Plattform* die abhängig von der Konfiguration Dienstleister, Nachbarn oder Angehörige benachrichtigt. Die Körperparameter (z.B. Gewicht, EKG, Blutdruck), die innerhalb der Wohnung aufgezeichnet werden, können nach der Übermittlung zur *IGM-Plattform* geprüft werden. Treten hier Unregelmäßigkeiten auf, wird ein Alarmplan ähnlich zu den bisher genannten ausgeführt. Einige Sensoren des SmarterWohnen-Projekts sind in Abb.16 dargestellt.

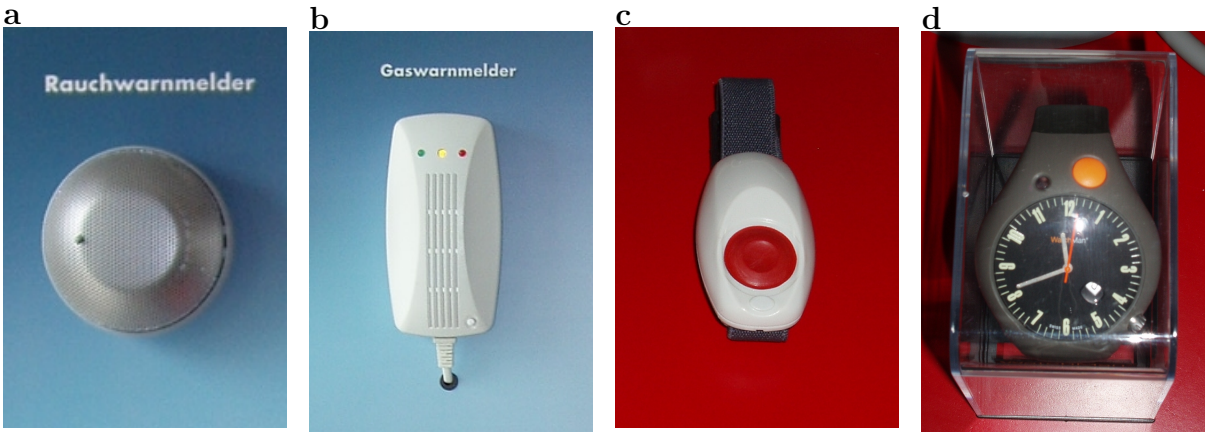


Abbildung 16: Verbaute Sensoren in der SmarterWohnen Musterwohnung. **a** Rauchmelder, Funkanbindung zum *Service-Gateway*; **b** Gaswarnmelder, Funkanbindung zum *Service-Gateway*; **c** elektrischer Finger; **d** Uhr mit Notfallknopf und Möglichkeit der Sprachausgabe

Prozessbezeichnung auf IGM	**Kategorie**	**Umsetzung als Prozess**
Brandmeldung	Event	ja
Brandbeendigung	Event	ja
Einbruchmeldung	Event	ja
Einbruchbeendigung	Event	ja
Aktivitätskontrolle	Event	ja
BeendeAktivitätskontrolle	Event	ja
Serviceruf	Event	ja
Gasleckagemeldung	Event	ja
BeendeGasleckagemeldung	Event	ja
Wasserleckagemeldung	Event	ja
BeendeWasserleckagemeldung	Event	ja
Dienstleisterbenachrichtigung	Event	ja
Bewohnerbenachrichtigung	Event	ja
Dienstaktivierung	Event	nein
Dienstkonfiguration	Event	nein
Energiewerteerfassung	Event	nein
Vitalwertübertragung	Event	nein
Monitoring der Komponenten	Event	nein
Monitoring des Gatewaystatus	Event	nein
Menüservice	Event	nein
Kontextänderung	Event	nein
Werteabruf durch Benutzer	Content	nein
Kalenderdienste	Content	nein
Contentdienste	Content	nein

Tabelle 1: Übersicht über die Prozesse von *SmarterWohnen*

Durch die klare Struktur und die Menge an Aktivitäten eignen sich die oben beschriebenen Notfallpläne gut zur Umsetzung als Geschäftsprozess. Alle Content basierten Dienste eignen sich weniger zur Umsetzung als Geschäftsprozess. Hier werden zumeist wenige Aktivitäten auf viele verschiedene Daten oder Rollen angewandt, so dass sich eine Umsetzung als ECA-Regelsystem anbietet.

Die content-basierten Dienste werden den Bewohnern über eine grafische Schnittstelle, z.B. dem Fernseher, dargestellt. Hier können auch die gemessenen Energieverbrauchswerte dargestellt werden. Die Übertragung der gemessenen Werte zur *IGM-Plattform*, die Aufbereitung und die Übertragung und Darstellung auf dem Fernseher ist auch ein komplexer Ablauf. Diese Art von Plattformabläufen ist dadurch charakterisiert, dass sie die verschiedenen Subsysteme ansprechen, die dann ihrerseits Aktivitäten ausführen. Diese Orchestrierung von Services wird durch die Plattform organisiert. Zu dieser Klasse von Abläufen zählt auch der Kalenderdienst, bei dem Termine online mit einem Dienstleister vereinbart werden können.

Die Übertragung von Wohnungsparametern und deren Überprüfung wird ebenfalls in verschiedenen Abläufen behandelt. Die Statusmeldungen von den verschiedenen Sensoren, sowie die des Servicegatways werden zur *IGM-Plattform* übertragen und in der Datenbank gespeichert. Weicht ein Wert von den Normalwerten ab, wird eine

Benachrichtigung ausgelöst. Der Status des "Verlassen-Szenarios", also das längere Verlassen der Wohnung, wird nur als Wert innerhalb der Datenbank gespeichert. Abläufe dieser Art sind nicht komplex genug um als Geschäftsprozess umgesetzt zu werden. Hier bietet sich der Einsatz von ECA-Regelsystemen unter Nutzung des EVA-Subsystems an.

Durch die Analyse der Plattformabläufe konnten die Abläufe herausgearbeitet werden bei denen es sich anbietet sie in Geschäftsprozesse umzusetzen. Die beschriebenen Erkenntnisse aus der Analyse werden in Tabelle 1 zusammengefasst.

8.1.4 Analyse anhand eines Musterprozesses: Prozess Brandmeldung

Anhand des Prozesses *Brandmeldung* soll das im vorherigen Abschnitt erläuterte analytische Vorgehen innerhalb der Untersuchung des IST-Zustandes des Gesamtsystems exemplarisch verdeutlicht werden. Hierbei ist diesem Prozess besondere Beachtung zu widmen, da die Prozesse *Einbruchmeldung*, *Gasleckagemeldung*, *Wasserleckagemeldung* sowie *Serviceruf* und *Aktivitätskontrolle* analog modelliert sind und diese Prozesse somit starke Ähnlichkeiten bezüglich des Ablaufes aufweisen.

Vorbedingungen Die Dienstleistung *Brandmeldung* muss zuvor durch den Bewohner als Dienst aktiviert bzw. abonniert worden sein. (Prozesse *Gesamtprozess* und *Dienstaktivierung*)

Nachbedingungen Nach vollständigem Ablauf des Prozesses wurde ein offener Notfall erzeugt. Dieser muss explizit durch den Bewohner oder den Dienstleister beendet werden.

Akteure Als Akteure innerhalb des Prozesses *Brandmeldung* fungieren

- *Service-Gateway*
- *IGM-Plattform*

Ereignisse

- Brand (von *Service-Gateway* detektiert und als Brand-Event gemeldet)

Parameter

- `wohnungsID` Bezeichner für die Wohnung, in der der Brand detektiert wird
- `Komponente` Bezeichner für die meldende Komponente der Hausinfrastruktur
- `Raum` Bezeichner für den Raum der Wohnung, in dem der Brand erkannt wird
- `Result` Erfolgsmeldung an das aufrufende *Service-Gateway*
- `returncode` (Subsystem-Methodenaufruf zum Nachrichtenversand)

Textuelle Beschreibung

Das *Service-Gateway* detektiert einen Brand und ruft daraufhin die Web-Methode *Brandmeldung* der *IGM-Plattform* auf; hierbei werden die Parameter `wohnungsID`, `Komponente` und `Raum` übergeben. Nun wird der Prozess von der *IGM-Plattform* durch Starten der aufgerufenen Web-Methode *Brandmeldung* vorangetrieben und das *Service-Gateway* über den Status des Aufrufs mittels des Rückgabeparameters `Result` informiert.

Es wird eine Notfallmeldung in der Datenbank protokolliert, die übergebenen Wohnungsdaten mit den zugehörigen Bewohnerdaten verknüpft und Dienstleisterdaten ermittelt. Nun wird der Dienstleister über den detektierten Brand informiert: Es wird eine entsprechende Nachricht erzeugt, diese abgesetzt, indem sie an das zuständige Subsystem weitergeleitet wird und der von diesem empfangene Statuscode (returncode) ausgewertet. Nun wird abhängig vom Erfolg des Nachrichtenversands entweder der Versand der Nachricht iterativ mittels anderer Kommunikationsmedien wiederholt (negatives Ergebnis beim Nachrichtenversand) oder im Falle eines positiven Ergebnisses der Versand protokolliert (Datenbank) und die Abrechnungsdaten erfasst (Abrechnungslog). Bei negativem Returncode des Kommunikations-Subsystems wird der Fehlversand protokolliert und weitere Kommunikationsmöglichkeiten evaluiert. Falls noch weitere Kommunikationsoptionen vorhanden sind, wird ein weiteres Kommunikationsmedium gewählt und der Versuch unternommen, dem Dienstleister die Nachricht auf diese Weise zuzustellen. Falls die Benachrichtigungsoption des Bewohners aktiviert ist (Bewohnerdaten), werden im Anschluss an die Benachrichtigung des Dienstleisters die Kommunikationsdaten des Bewohners ermittelt, eine Nachricht erzeugt und abgesetzt; außerdem wird der Versand protokolliert und das Abrechnungslog geschrieben.

Lebenszyklus

Dauer Es handelt es sich um einen Realzeitprozess, in dem die vom *Service-Gateway* detektierte Brandmeldung schnellstmöglich an den Dienstleister und ggf. den Bewohner gemeldet werden muss. Es gibt innerhalb des beschriebenen Prozesses keine Teilprozesse mit langer Prozessdauer.

Elternprozesse In der Dokumentation der Prozesse des ISST (vgl. [14]) wird der Prozess *Brandmeldung* durch den *Gesamtprozess* als zentrale Steuerungsinstanz aufgerufen. Diese Aufgabe übernimmt in der *jABC*-Implementierung der *ProcessManager* (siehe 9.3).

Kindprozesse Die Prozesse *Dienstleisterbenachrichtigung* und *Bewohnerbenachrichtigung* werden als Teilprozesse innerhalb von *Brandbeendigung* aufgerufen.

Konflikte und Abhängigkeiten

Der Ablauf des Prozesses *Brandmeldung* wird direkt beeinflusst durch den Prozess *Brandbeendigung*, sofern sich beide Prozesse auf dieselbe Wohnung beziehen. Es sind ferner Ausnahmen zu definieren, wie innerhalb von *Brandmeldung* in den einzelnen Prozessschritten auf die Beendigung des Notfalls zu reagieren ist; dies kann beispielsweise durch einen Rollback erfolgen. Eine gleichwertige Abhängigkeit besteht bezüglich des Prozesses *BeendeOffeneNotfaelle*.

Weiterhin ist der nebenläufige Aufruf der Prozesse Brandmeldung und Gasleckagemeldung in Bezug auf dieselbe Wohnung bzw. Gebäudeeinheit mit Abhängigkeiten und möglicherweise Konflikten behaftet. Bei gleichzeitigem Auftreten eines Brandes und eines Gasleitungslecks sind Dienstleister und Bewohner zu warnen und ggf. ein vom Standardfall abweichender Prozessverlauf durchzuführen. Dies ist bei der Modellierung durch das ISST nicht berücksichtigt worden, sollte jedoch in die Implementierung der Prozesse im *jABC* miteinfließen.

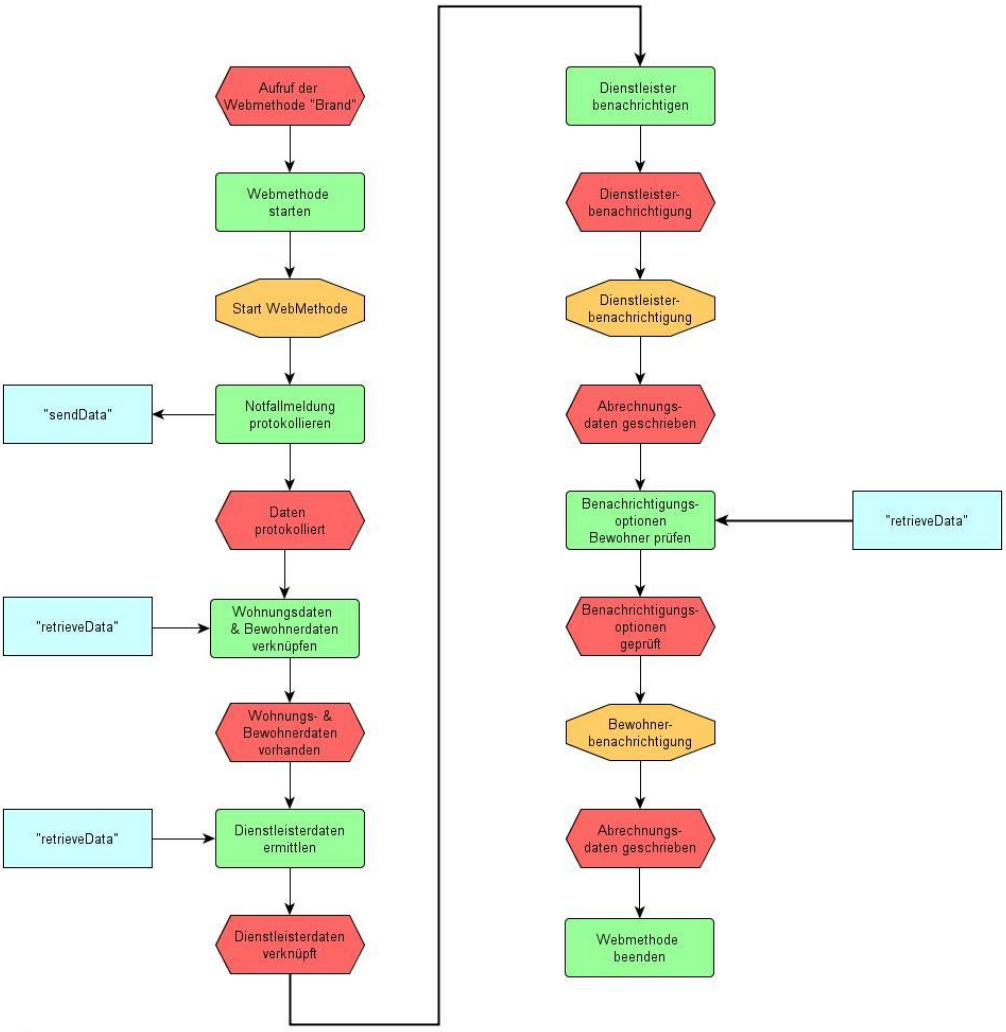


Abbildung 17: eEPK des Prozesses *Brandmeldung*

Falls während eines laufenden *Brandmeldung*-Prozesses der Dienst *Brandmeldung* für eine Wohneinheit durch den Prozess *DeaktiviereDienst* deaktiviert bzw. dessen Einstellungen durch den Prozess *Dienstkonfiguration* verändert werden, bestehen Abhängigkeiten zwischen den genannten Prozessen. Diese müssen in folgenden Analyseschritten spezifiziert und geeignete Ausnahmebehandlungen entwickelt werden.

Graphische Prozessdarstellung als eEPK

In Abbildung 17 ist der Prozess *Brandmeldung* als erweiterte ereignisgesteuerte Prozesskette dargestellt. Die als gelbes Achteck repräsentierten Unterprozesse *StartWebMethode*, *Dienstleisterbenachrichtigung* und *Bewohnerbenachrichtigung* sind Kindprozesse von *Brandmeldung* und somit in gesonderten eEPKs modelliert.

8.2 Identifizierung und Entwicklung von *SIBs* und *Events*

8.2.1 Überblick

Innerhalb der Analysephase wurden die *SmarterWohnen*-Prozesse in Geschäftsprozesse überführt. Zur graphischen Darstellung wurde die Methode der erweiterten ereignis-

gesteuerten Prozessketten (eEPK) gewählt. Diese Darstellungsform ermöglicht die Identifizierung von funktionalen Elementen, die als *SIBs* im *jABC* modelliert werden, sowie von *Ereignissen* und *Ein-* und *Ausgabeparametern*, die zwischen dem *jABC* und der *IGM-Plattform* ausgetauscht werden müssen.

Mit Hilfe dieser Modelle wurden nicht nur Anforderungen an die zu entwickelnden *SIBs* erarbeitet, sondern auch Aspekte an das Design der Schnittstelle zwischen *IGM-Plattform* und *jABC*. Zunächst mussten redundant wiederkehrende Teilprozesse der *SmarterWohnen*-Geschäftsprozesse zu *SIBs* zusammengefasst werden. Dies ermöglichte eine induktive hierarchische Abbildung der Prozesse im *jABC*. Diese *SIBs* auf unterster Hierarchisierungsstufe sind von sehr technischer Natur und wurden zu Einheiten zusammengefasst, die die fachliche Sicht wiedergeben.

Einige funktionale Eigenschaften ließen sich jedoch nicht als Prozess modellieren; daher mussten diese in der Schnittstelle zur *IGM-Plattform* implementiert werden.

Die Analyse der Lebenszyklen der Prozesse brachte zu Tage, dass diese sich durch Nebenläufigkeit und Wartezyklen auszeichnen, sodass innerhalb der Erweiterung des *jABC*-Systems zusätzliche Konzepte zur Ausführung, Kontrollflusssteuerung und Überwachung entwickelt werden mussten. Desweiteren erforderte das Starten und Beenden von Prozessen eine übergeordnete Verwaltung (vgl. Abschnitt 9.3.3 und *Elternprozesse* in Abschnitt 8.1.4).

8.2.2 Entwicklung von *SIBs*

Die zur Umsetzung der *SmarterWohnen*-Prozesse erforderlichen *SIBs* waren in erster Linie elementare *SIBs*, welche die Grundfunktionen auf atomarer Ebene abdecken. Aus diesen elementaren *SIBs* wurden im Anschluss komplexere, zusammengesetzte *SIBs* erzeugt, die die Komplexität für den Betrachter reduzieren und somit die fachliche Prozesssicht darstellten. Technische Details wurden auf diese Weise vor dem Modellierer der Geschäftsprozesse verborgen.

Als elementare Funktionen von *SIBs* ließen sich der lesende Zugriff auf Daten der IGM-seitigen Datenbanken sowie das Versenden von Nachrichten unter Verwendung des *DIA-Subsystems* identifizieren.

Die aus elementaren *SIBs* zusammengesetzten komplexeren *SIBs* verwendeten deren bereitgestellte gekapselte Funktionen und bildeten somit einheitliche Prozessschritte auf fachlicher Ebene ab. So wurden beispielsweise aus einer Kombination von elementaren *SIBs* Funktionen wie Bewohnerkontext- und Datenbankabfragen sowie Nachrichtenversand das zusammengesetzte *SIB Bewohnerbeachrichtigung*.

In einem weiteren Entwicklungsschritt wurden *SIBs* der oben genannten Arten identifiziert und auf Eingabeparameter, Rückgabeparameter, eingehende und ausgehende Kanten hin untersucht. Außerdem wurden die durch den jeweiligen *SIB* zu realisierende Funktion textuell beschrieben, sodass diese auf der *jABC*-Implementierungsebene codiert werden konnten. Auf Basis der ermittelten Ein- und Ausgabeparameter wurde im Anschluss an die Entwicklung der benötigten *SIBs* eine strukturierte Parametermenge gebildet (vgl. Abschnitt 9.1.3), welche eindeutig die über die als Webservice realisierte Kommunikationsschnittstelle zwischen dem *jABC* und der *IGM-Plattform* austauschbaren Daten adres-

SIB-Bezeichnung	Beschreibung
Init	Initialisiert einen Prozess
AboEvent	Abonniert ein bestimmtes Event, über dessen Auftreten der Prozess nun informiert wird
DeAboEvent	Deabonniert ein Event, keine Benachrichtigung mehr
SendEvent	Sendet ein Event an die *IGM-Plattform*
ProtocolIncident	Protokolliert aufgetretenden Notfall
ProtocolEndOfIncident	Protokolliert Beendigung des Notfalls
WriteBillingData	Schreibt Abrechnungsdaten in Abrechnungsdatenbank
retrieveData	Dient dazu verschiedene Datenbankanfragen an die *IGM-Plattform* zu richten. Welche Daten genau angefordert werden, wird über *DataTransferObjects* spezifiziert (vgl. Parametertaxonomie in Kapitel 9.1.3)
CreateMessage	Erstellt eine Benachrichtigung
SendMessage	Versendet eine Benachrichtigung über das *DIA-Subsystem* der *IGM-Plattform* (siehe Abschnitt 5.3.4)
NotificationControl	Überprüft, ob eine Benachrichtigung erfolgreich versandt wurde
Iterator	Iteriert z.B. über eine Anzahl von Dienstleister, falls mehrere für einen Notfall zuständig sind

Tabelle 2: Aus den *SmarterWohnen*-Geschäftsprozessen identifizierte *SIBs*

sierte. Auf diese Weise wurde eine abstrakte Sicht auf alle vorhandenen Ein- und Ausgabeparameter der *SmarterWohnen*-Prozesse geschaffen. Daher kann die Entwicklung der *SIBs*, der *jABC*-Laufzeitumgebung und der *IGM-Plattform*-Erweiterungen unabhängig voneinander vorangetrieben werden.

In Tabelle 2 sind alle *SIBs* aufgeführt, die sich aus den fachlichen Beschreibungen der *SmarterWohnen*-Prozesse ableiten ließen. Zur Verwaltung der auftretenden Events und für die Kommunikationsschnittstelle zwischen dem *jABC* und der *IGM-Plattform* waren weitere *SIBs* nötig. Deren Bezeichnungen und Funktionen können dem Kapitel 9.3 Prozessmanagement entnommen werden.

8.2.3 Identifizierung von Events

Zusätzlich zur Identifizierung und Entwicklung von *SIBs* war es notwendig, auf der *IGM-Plattform* und während des Ablaufs der *SmarterWohnen*-Prozesse in der *jABC*-Laufzeitumgebung auftretende Events zusammenfassend darzustellen; insbesondere bei der umfassenden Beschreibung von funktionalen Abhängigkeiten zwischen Prozessen wurde Events eine besondere Rolle zuteil, da sie die wichtigste Kommunikationsmöglichkeit von Prozessen untereinander darstellten.

Funktionale Abhängigkeiten Um funktionale Abhängigkeiten zwischenden Prozessen entdecken zu können, mussten alle auf der *Service-Plattform* und den *Service-Gateways* auslösbaren Events betrachtet werden. Beispiele für funktionale Abhängigkeiten sind Fälle, in denen eine durch ein *MeldeBrand*-Event angestoßener *Alarmmeldung*-Prozess noch nicht vollständig abgearbeitet wurde, dieser Alarm jedoch durch das Event *BeendeBrand*

vorzeitig beendet wurde.
Auch die Änderung der Konfiguration eines Dienstes während einer zeitgleich laufenden Instanz desselben konnte zu Nebeneffekten führen. Ferner ist zu beachten, dass einige Alarmmeldungen indirekt andere Alarme auslösen konnten. So wird die Feuerwehr, die wegen einer Brandmeldung ausrückt, in die Wohnung eindringen und somit eine Einbruchmeldung auslösen; durch das bei der Brandbekämpfung eingesetzte Löschwasser wird darüberhinaus eine Wasserleckagemeldung erzeugt.
Desweiteren sollte auch auf das kombinierte auftreten verschiedener Events reagiert werden. Ist für eine bestimmte Wohnung z.B. ein Brandprozess aktiv und tritt für die gleiche Wohnung das Event *Gasleckage* auf, so sollte auf Grund der hohen Explosionsgefahr nicht nur der Gasdienstleister sondern auch der Dienstleister, der gerade mit dem Löschen des Brandes beschäftigt ist, informiert werden. Zusätzlich könnten die Bewohner der Nachbarwohnungen über das erhöhte Gefahrenpotential informiert werden.

Kategorisierung von Events Alle von uns identifizierten, die *Service-Plattform* betreffenden, Events ließen sich in die folgenden Kategorien einordnen, *Alarms*, *Close Alarms* und *Service-Gateway*-Events. Wie die Bezeichnungen der Kategorien *Alarms* und *Close Alarms* bereits suggerieren, wurde der größte Teil der Events benötigt, um das Auftreten oder die Beendigung eines Alarms dem *jABC* zu signalisieren. Die in der Kategorie Sonstige zusammengefassten Events signalisieren zum einen eine direkte Interaktion der Benutzer, *Servicecall*, bzw. den Ausfall von Komponenten, *NotifyComponentProblem*. Wobei der *Servicecall* in unserer aktuellen Implementation nicht wieder zufinden ist. Auch auf den Ausfall einer *IGM*-Komponente konnte nur bedingt reagiert werden, da dem *jABC* keine Machanismen zur Verfügung stehen, um solch einen Ausfall zu kompensieren. Die einzige Möglichkeit wäre die, sämtliche betroffene Teilnehmer zu informieren, was natürlich nur möglich ist, wenn nicht die für die Benachrichtigung zuständige Komponente ausgefallen ist.

Events	**Beschreibung**
Alarms	
NotifyFire	Meldet einen Brand in einer Wohnung
NotifyBreakin	Meldet einen Einbruch in einer Wohnung
NotifyGasLeak	Meldet eine Gasleckage in einer Wohnung
NotifyWaterLeak	Meldet eine Wasserleckage in einer Wohnung
NotifyActivity	Der Bewohner befindet sich in seiner Wohnung hat sich über einen längeren Zeitraum nicht bewegt
NotifyBadVitalValues	Meldet schlechte Vitalwerte für einen Bewohner
Close Alarms	
CloseFireAlarm	Das Feuer wurde gelöscht (bzw Fehlalarm)
CloseBreakinAlarm	Keine Gefahr mehr durch Einbrecher
CloseWaterLeakAlarm	Wasserschaden behoben
CloseGasLeakAlarm	Kein austretendes Gas mehr
CloseActivityAlarm	Zustand des Bewohners wurde überprüft
CloseVitalValueAlarm	Vitalwerte des Bewohners wurden überprüft
Sonstige	
NotifyComponentProblem	Eine Komponente der IGM-Plattform meldet ein Problem
Servicecall	Der Bewohner fordert manuell Hilfe an

Tabelle 3: Kategorisierung von auftretenden Events

9 Technisches Gesamtkonzept

9.1 Kommunikation

Zu Beginn der Projektgruppe ergab sich als eine der maßgeblichen Fragestellungen, inwiefern das *jABC* und die *IGM-Plattform* an der Ausführung der Prozesse beteiligt sein sollen. Nach anfänglichen Überlegungen auf beiden Plattformen (Teil-)Prozesse auszuführen und eine entsprechende Verwaltung der Prozesse auf beiden Plattformen umzusetzen, wurde dieser Gedanke wieder verworfen und letztendlich entschieden die Ausführung und Modellierung der Prozesse im *jABC* umzusetzen. Die *IGM-Plattform* wird daher, sofern die Ausführung von Prozessen betroffen ist, nur für informationslogistische Aspekte eingesetzt. Dazu zählt vor allem das Bereitstellen von Kontextinformationen und Benutzerdaten, die Kommunikation mit dem Benutzer und das Auswerten von Events auf der Plattform. Aus dieser Entscheidung ergibt sich auch die Konsequenz, dass das *BPM-Subsystem* (vergleiche Kapitel 5) nur eine Proxyfunktionalität gegenüber dem *jABC* wahrnimmt. Dies wirft wiederum die folgenden Fragen auf:

- Welche Informationen müssen zwischen den Systemen ausgetauscht werden?
- Wie werden diese Informationen zwischen den beiden System ausgetauscht bzw. wie kommunizieren die beiden Systeme miteinander?

Für die Kommunikation zwischen den Plattformen sollte ein Ansatz mit loser Kopplung der Systeme gefunden werden. Dies machte die Entscheidung für eine *serviceorientierte Architektur* (SOA) einfach. Die Entscheidung fiel daher auf Webservices, die einen Quasi-Standard im Bereich SOA bilden. Dementsprechend mussten Operationen definiert werden, über die Informationen zwischen den Systemen ausgetauscht werden. Diese Operationen sollten möglichst generisch sein, damit bei eventuellen Erweiterungen oder neuen Anforderungen neuer Prozesse keine Änderung der Webservice-Schnittstelle notwendig wird. Darüber hinaus sollte die Webservice-Schnittstelle zunächst einmal abhängig von den auszutauschenden Informationen der bereits vorhandenen Prozesse modelliert werden. Zu diesem Zweck wurden die benötigten Informationen aus den Prozessen extrahiert und entsprechende Anforderungen an die Schnittstelle entwickelt. Als Ergebniss dieser Arbeit wurde eine Übersicht über die zu übertragenden Daten erstellt. Diese werden in den nachfolgenden Tabellen 4 und 5 dargestellt.

Auf Grundlage dieser Daten wurden WebService-Methoden definiert, die zum Austausch dieser Daten dienen sollen.

Zum einen musste es möglich sein, Events, die auf der *IGM-Plattform* auftreten, den Prozessen im *jABC* zur Verfügung zu stellen. Dafür dient auf der *jABC*-Schnittstelle die entsprechende Methode `sendEvent`. Umgekehrt kann es vorkommen, dass innerhalb eines Prozesses im *jABC* ein Event auftritt. Daher muss es ebenfalls möglich sein, Events mittels `sendEvent` von *jABC* and die *IGM-Plattform* zu senden. Bei weiteren Überlegungen ergab sich die Notwendigkeit, dass einige Events Kontextdaten enthalten müssen. Dies kann z.B. notwendig sein, wenn das Event *Brandmeldung* ausgelöst wird. Hier interessiert beispielsweise nicht nur, dass es brennt, sondern auch, in welchem Haus und welchem Raum es brennt. Um solche Informationen weiterleiten zu können, tragen Events immer Kontextinformationen.

In gewissen Situationen kann es notwendig sein, dass ein Prozess on-demand Kontext– oder Benutzerinformationen benötigt oder verändern muss. Zu diesem Zweck werden auf der Webservice-Schnittstelle auf der IGM-Plattform zwei Methoden `retrieveData` und `sendData` angeboten. Für beide Methoden ist es notwendig, jeweils einen Kontexttyp zu übergeben, da auf der *IGM-Plattform* unterschiedliche Kontextinformationen vorhanden sind. Für die Methode `sendData` ist es außerdem notwendig, eine Liste

WebMethode	Parameter
MeldeBrand	wohnungsID Komponente*String Raum*String
MeldeEinbruch	wohnungsID Komponente*String Raum*String
MeldeGasleckage	wohnungsID Komponente*String Raum*String
MeldeWasser-leckage	wohnungsID Komponente*String Raum*String
MeldeAktivität	wohnungsID Komponente*String Raum*String
Beende-MeldeBrand	brandID*String dienstleisterID*String gatewayID*String hinweis*String
Beende-MeldeEinbruch	einbruchID*String dienstleisterID*String gatewayID*String hinweis*String
Beende-MeldeGasleckage	gasID*String dienstleisterID*String gatewayID*String hinweis*String
Beende-Melde-Wasserleckage	wasserID*String dienstleisterID*String gatewayID*String hinweis*String

WebMethode	Parameter
leseNachrichten	keine
leseStatus	keine
leseKonfiguration	keine
MeldeEnergiewerte	seriennummer*String wert*String datum*DateTime energiewerttyp*int zusatz*String
MeldeVitalwerte	seriennummer*String wert*String datum*DateTime energiewerttyp*int zusatz*String
MeldeKomponente-Problem	WEID*String Komponente*String raum*String hinweis*String
MeldeStatusGateway	gatewayID*String
BeendeOffeneNotfälle	WEID*String komponente*String raum*String
MeldeContext	WEID*String contextenum

Tabelle 4: Übersicht über die von den Webmethoden erwarteten Parameter

Event	übergebene Parameter	Auslöser
Gateway Guard	gatewayID	Gateway Guard
NeukundenID	bewohnerID kundenID Anbieter	Menüservice

Tabelle 5: Sonstige auftretende Events

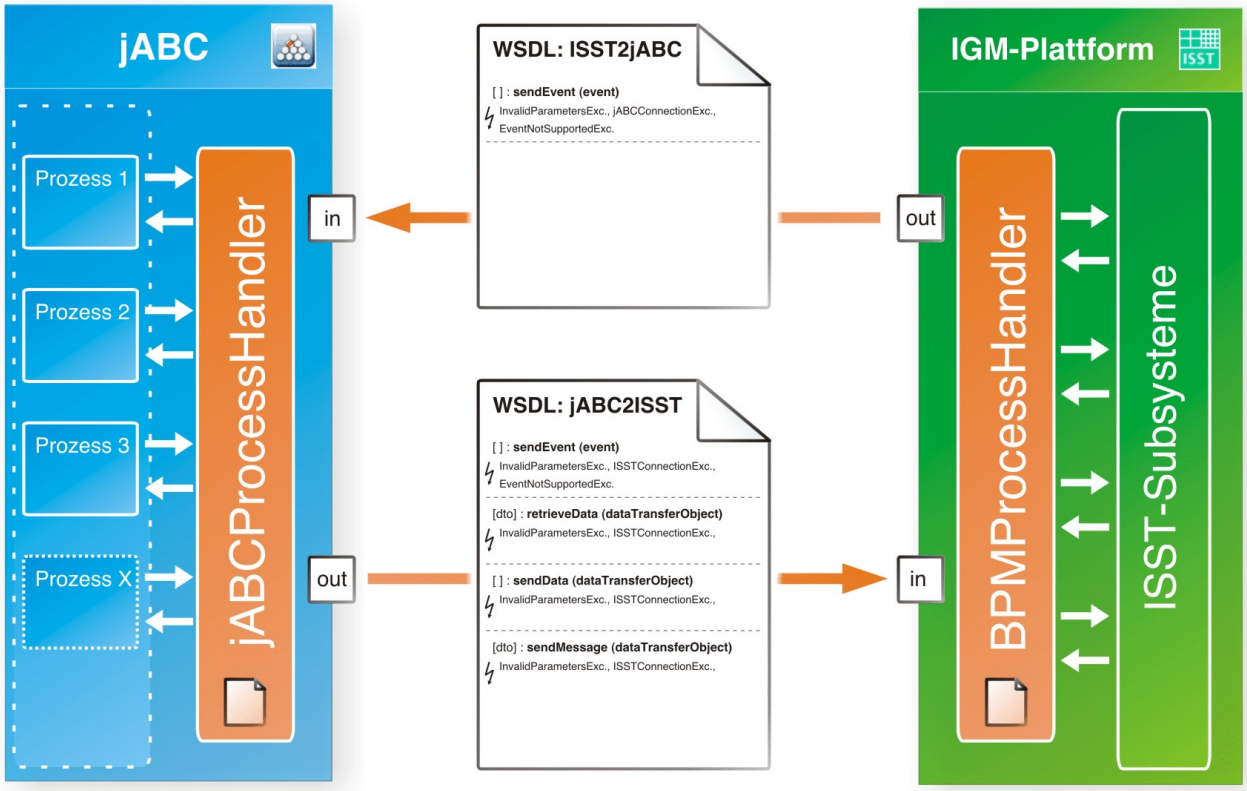


Abbildung 18: Übersicht Webservice-Schnittstellen

mit Schlüssel-Wert-Paaren zu übergeben, die die zu setzenden Kontextinformationen beinhalten. Die Methode `retrieveData` hingegen benötigt zwei weitere Listen. Beide beinhalten ebenfalls Schlüssel-Wert-Paare. Die eine Liste beinhaltet die Schlüssel, für die Informationen angefordert werden (Werte sind in diesem Fall leer) und Schlüssel mit Werten, die für die Bestimmung der angeforderten Werte benötigt werden, z.B. bei Anfrage einer E-Mail-Adresse die `userId`, um den Benutzer zu identifizieren, für den die E-Mail-Adresse angefragt wird. Die andere Liste beinhaltet entsprechend die Ergebnisse der Anfrage.

Für die Benachrichtigung von Bewohnern und Dienstleistern ist außerdem eine Methode vorgesehen, um Nachrichten zu verschicken. Prinzipiell gab es dafür auch die Überlegung, die Methode `sendData` zu nutzen. Allerdings fiel die Entscheidung eine weitere Methode `sendMessage` einzuführen, um den semantischen Unterschied deutlicher hervor zu heben. Für den Aufruf der Methode ist es ebenfalls nötig, einen Typ zu übergeben. Dieser dient z.B. der Unterscheidung zwischen verschiedenen Benachrichtigungsarten wie z.B. E-Mail oder SMS. Außerdem wird eine Liste mit Schlüssel-Wert-Paaren benötigt, die den Nachrichteninhalt, den Empfänger und evtl. nötige weitere Daten enthalten können.

Die Abbildung 18 zeigt eine schematische Übersicht der definierten Schnittstellen.

Damit das *jABC* eindeutig auf die Daten der *IGM-Plattform* zugreifen kann, wurde eine Taxonomie entwickelt. Diese ist in Abschnitt 9.1.3 beschrieben.

9.1.1 Technische Beschreibung der Datenpakete und Schnittstellen

Die nachfolgenden Abbildungen beschreiben die o.g. Methoden auf den beiden Schnittstellen.

ISST2jABC		
sendEvent		
input	parameters	sendEvent
output	parameters	sendEventResponse
InvalidParametersExceptionFault	detail	InvalidParameterExcpetion
EventNotSupportedExceptionFault	detail	EventNotSupportedException

Abbildung 19: WebService-Schnittstelle *jABC*

jABC2ISST		
sendEvent		
input	parameters	sendEvent
output	parameters	sendEventResponse
EventNotSupportedExceptionFault	detail	EventNotSupportedException
InvalidParametersExceptionFault	detail	InvalidParameterExcpetion
sendMessage		
input	parameters	sendMessage
output	parameters	sendMessageResponse
TypeNotSupportedExceptionFault	detail	TypeNotSupportedException
InvalidParametersExceptionFault	detail	InvalidParameterExcpetion
sendData		
input	parameters	sendData
output	parameters	sendDataResponse
TypeNotSupportedExceptionFault	detail	TypeNotSupportedException
InvalidParametersExceptionFault	detail	InvalidParameterExcpetion
retrieveData		
input	parameters	retrieveData
output	parameters	retrieveDataResponse
InvalidParametersExceptionFault	detail	InvalidParameterExcpetion
TypeNotSupportedExceptionFault	detail	TypeNotSupportedException

Abbildung 20: WebService-Schnittstelle *IGM-Plattform*

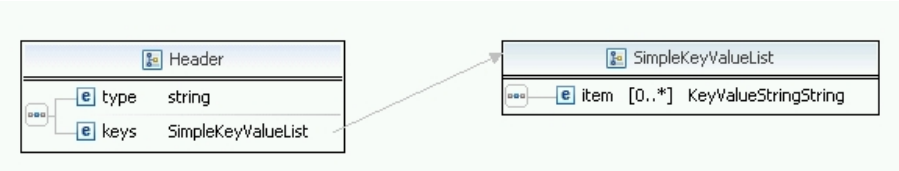


Abbildung 21: WebService-Datentyp `Header`

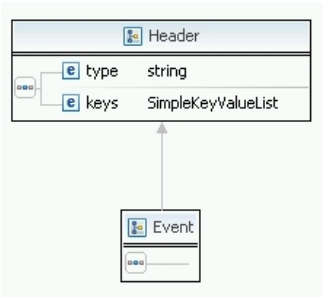


Abbildung 22: WebService-Datentyp `Event`

Die beiden Methoden `sendEvent` haben als Eingabeparameter jeweils einen Parameter vom Typ *Event*. Als Ausgabe-Parameter wird jeweils ein boolescher Wert zurückgegeben. Dieser soll angeben, ob der Event verarbeitet werden konnte. Dies ist unter anderem deswegen notwendig, da die technische Einschränkung besteht, dass nur bei synchroner Kommunikation die Möglichkeit besteht auch SOAP-Faults zu übertragen. Es wurden zwei SOAP-Faults definiert. Hierzu zählt zum einen `EventNotSupportedExceptionFault`. Dieser SOAP-Fault wird ausgelöst, wenn das empfangende System einen Eventtypen nicht unterstützt. Zum anderen wurde ein `InvalidParameterExceptionFault` definiert. Dieser wird ausgelöst, wenn ein Schlüssel-Wert-Paar des Events nicht der für den Eventtypen spezifizierten Liste von geforderten Schlüssel-Wert-Paaren entspricht. Die Methoden `sendMessage`, `sendData` und `retrieveData` nehmen jeweils einen Parameter vom Typ `DataTransferObject` entgegen. Aus o. g. Gründen geben die Methoden `sendData` und `sendMessage` jeweils einen booleschen Wert zurück. Der Rückgabewert der Methode `retrieveData` ist vom Typ `DataTransferObject` und beinhaltet die angeforderten Daten. Die für diese Methoden definierten SOAP-Faults entsprechen von ihrer Semantik her den bereits für die Methode `sendEvent` erwähnten SOAP-Faults. Da über diese Methoden allerdings keine Events ausgetauscht werden, wurde der SOAP-Fault `EventNotSupportedExceptionFault` entsprechend in `TypeNotSupportedExceptionFault` umbenannt.

Als Grundlage für die Datentypen `Event` und `DataTransferObject` dient der abstrakte Datentyp `Header`.

Die Bedeutung der einzelnen Elemente des Datentyps `Header` werden nachfolgend jeweils bei den konkreten Typen `Event` und `DataTransferObject` beschrieben. Der Datentyp `Event` erbt vom Datentyp `Header`. Er repräsentiert einen Event, der auf der *IGM-Plattform* oder im *jABC* ausgelöst wurde. Ein Event wird durch ein Element `type` (`Header.type`) vom Typ `string` klassifiziert. Des Weiteren enthält ein Event ein Element `keys` (`Header.keys`), das eine Liste von Schlüssel-Wert-Paaren (jeweils vom Typ `string`) repräsentiert, die die Kontextinformationen enthalten, die wiederum den Kontext beschreiben, in denen ein Event aufgetreten ist.

Wie der Datentyp `Event` erbt auch der Datentyp `DataTransferObject` vom abstrakten Datentyp `Header`. Das Element `type` (`Header.type`) erlaubt eine Klassifizierung eines übertragenen `DataTransferObjects` anhand der o. g. Taxonomie. Über das Element `keys`

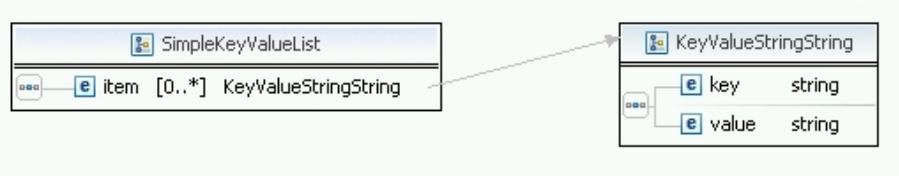


Abbildung 23: WebService-Datentyp `SimpleKeyValueList`

Abbildung 24: WebService-Datentyp `KeyValueStringString`

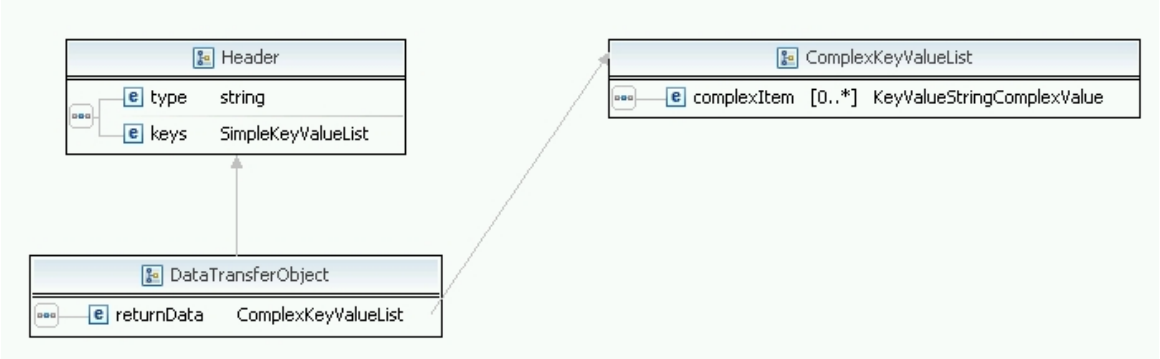


Abbildung 25: WebService-Datentyp `DataTransferObject`

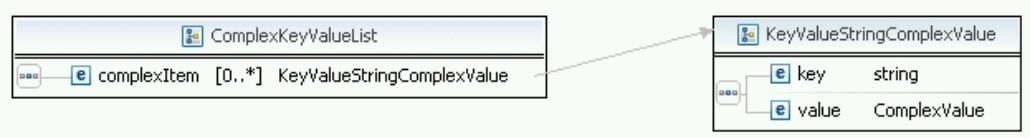


Abbildung 26: WebService-Datentyp `ComplexKeyValueList`

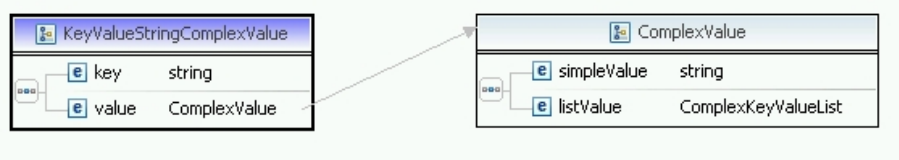


Abbildung 27: WebService-Datentyp `KeyValueStringComplexValue`

(`Header.keys`) wird der Kontext, in dem das `DataTransferObject` übertragen wird, näher bestimmt. Das Element `returnData` vom Typ `ComplexKeyValueList` kann weitere Schlüssel-Wert-Paare vom Typ `KeyValueStringComplexValue` enthalten. Diese dienen der Spezifikation von Daten, die vom *jABC* an die *IGM-Plattform* übermittelt oder von dieser abgerufen werden sollen. Wenn beispielsweise die E-Mail-Adresse eines Benutzers ermittelt werden soll, würde `type` einen Wert enthalten, der darauf hinweist, dass Benutzerinformationen benötigt werden. Unter `keys` wären bestimmte (Primär-)Schlüssel zur Identifikation des Benutzers hinterlegt, wie z.B. seine ID oder der Benutzername. Das Element `returnData` würde dann genau ein Schlüssel-Wert-Paar enthalten, bei dem der Schlüssel E-Mail lautet und der Wert leer ist. Somit ist auf der *IGM-Plattform* klar, dass die Benutzerinformation (`type`) E-Mail-Adresse (`returnData`) für den Benutzer mit einer bestimmten ID oder Benutzernamen (`keys`) benötigt wird. Für die Werte der Schlüssel-Wert-Paare wurde hier durch einen komplexeren Datentyp die Möglichkeit geschaffen, komplette Listen an Werten zu spezifizieren. Dies ist unter Umständen nötig, wenn z. B. Benachrichtigungskanäle eines Benutzers ermittelt werden sollen, dieser aber mehr als einen Kanal definiert hat. Ursprünglich sollte der Datentyp `ComplexValue` nur die Möglichkeit bieten, entweder einen String oder eine `ComplexKeyValueList` aufzunehmen. Dies ließe sich normalerweise leicht mit einem *XML-Schema-Element* `choice` abbilden. Leider sind in den *DataContracts* aus dem *Microsoft .Net 3.0 (WCF) Framework* keine `choice`-Elemente erlaubt (s. [4]).

9.1.2 Contracts

Es wurde bereits mehrfach erwähnt, dass Informationen zwischen dem *jABC* und der *IGM-Plattform* ausgetauscht werden müssen. Außerdem wurden die auszutauschenden Informationen definiert und entsprechende Datentypen und eine Taxonomie festgelegt. Es ist aber noch nicht geklärt, wie das Datenformat aussieht, in dem definiert wird, welche Daten zwischen den Systemen ausgetauscht werden. Klar ist, dass es Events und Kontextinformationen gibt, die ausgetauscht werden sollen. Um zu spezifizieren, welche Events das *BPM-Subystem* an das *jABC* senden soll, werden *Contracts* aufgesetzt. Gleichzeitig sichert

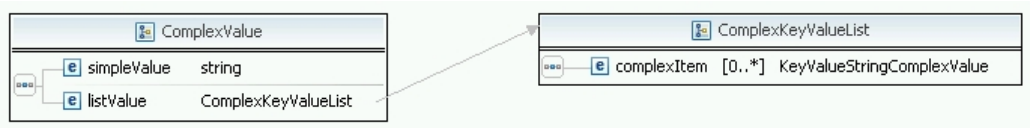


Abbildung 28: WebService-Datentyp `ComplexValue`

der *Contract* zu, dass das *jABC* auch in der Lage ist, ein solches Event zu verarbeiten. Ein weiterer *Contract* beschreibt die Rückrichtung. Aus Sicht des *BPM-Subystems* wird damit festgelegt, welche Events, die vom *jABC* gesendet werden, empfangen und verarbeitet werden können. Für das Senden und Empfangen von Kontextinformationen ist ebenfalls ein *Contract* vorgesehen. In diesem wird vor allem das Mapping der auf Webservice-Ebene verwendeten Schlüssel auf *Profile*[16] und Schlüssel der einzelnen Subsysteme spezifiziert. Für das Versenden von Nachrichten wird ebenfalls ein *Contract* spezifiziert, der die Nachrichtentypen und benötigte Informationen für den Versand dieser Nachrichten spezifiziert. Für die Definition dieser *Contracts* wurde kein spezielles Datenformat spezifiziert. Die *Contracts* stellen die Spezifikation der beiden Systeme dar und sind somit eine Vereinbarung zwischen den Personen, die die jeweilige Funktionalität in den beiden Systemen implementieren. Die im Abschnitt 9.1.3 definierte Parameterstruktur stellt eine Festlegung dieser *Contracts* dar.

Im Laufe der Arbeit der Projektgruppe stellte sich heraus, dass eine Implementierung bzw. eine Darstellung der *Contracts* in einem durch eine Maschine verifizierbaren Format sinnvoll wäre. Nachfolgend sollen daher kurz die Umsetzungen auf beiden Seiten, *jABC* und *IGM-Plattform*, beschrieben werden.

jABC-Plattform Auf der *jABC*-Seite wurden die *Contracts* mittels XML-Dateien umgesetzt. Für jedes *DataTransferObject (DTO)* liegt eine XML-Datei vor, in der definiert wird welche Informationen übertragen werden:

Listing 3: Beispiel-DTO: `GetUser`

```
<dto name="GetUser">
  <context>transactionID</context>
  <context>Domicile.domicileID</context>
  <retrieve>User.userID</retrieve>
  <retrieve>User.name</retrieve>
  <retrieve>User.forename</retrieve>
  <retrieve>User.gender</retrieve>
</dto>
```

Diese XML-Dateien sind im *jABC* beim Modellieren des Graphen als *SIB*-Parameter auswählbar. Dadurch kann der Anwendungsentwickler keine ungültigen *DTOs* verschicken. Nur beim Erstellen neuer *DTO*-Typen muss darauf geachtet werden, syntaktisch korrekte und semantisch gültige (d.h. auch auf der *IGM-Plattform* bekannte) *Contracts* zu erstellen. Dies könnte durch ein zu validierendes XML-Schema unterstützt werden, ist bislang aber noch nicht implementiert. Intern wird aus der XML-Datei ein `DTOType`-Objekt erstellt. `context`-Tags werden dabei aus dem Kontext ausgelesen und zusammen mit deren Wert als Schlüssel/Wert-Paar übertragen. `retrieve`-Tags werden mit einem leeren `String` als Wert übertragen. Dies sind die angeforderten Informationen, die das *jABC* von der *IGM-Plattform* benötigt.

IGM-Plattform Auf der IGM-Seite wurden die *Contracts* so implementiert, dass für alle eingehenden *Events* oder *DataTransferObjects* der für einen Typ zuständige *Handler* bzw. *Mapper* eine Überprüfung der Parameter vornimmt. Die Überprüfung testet dabei lediglich, ob alle, in der Parametertaxonomie definierten, Parameter vorhanden sind und ob nicht weitere Parameter mitgesendet wurden. Im Falle, dass die tatsächlichen Parameter nicht mit den definierten Parametern übereinstimmen, wird ein entsprechender SOAP-Fault ausgelöst. Eine Darstellung bzw. Formalisierung der *Contracts*, in einem durch eine

[16] vgl. Listing 20 auf Seite 122

Maschine verifizierbaren Format und vor allem auch zwischen den Parteien austauschbarem Format, ist somit momentan nicht vorhanden. Alle Überprüfungen liegen derzeit nur in Form einer festen Implementierung vor, die nicht generisch nutzbar ist. Allerdings besteht die Möglichkeit auf Basis von XML Schema[57] eine entsprechende Formalisierung vorzunehmen. So kann eine gültige Instanz eines Events vom Typ *Evacuation* folgendermaßen durch eine XML Schema Definition beschrieben werden:

Listing 4: beispielhafte XML Schema Definition für eine *Evacuation*-Event

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:tns="http://poets.unido.de/webservice/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://poets.unido.de/webservice/">
  <xsd:element name="sendEvent">
    <xsd:complexType>
      %\colorbox{green}{<xsd:sequence>}%
        <xsd:element name="message">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="type">
                <xsd:simpleType>
                  <xsd:restriction
                    base="xsd:string">
                    <xsd:enumeration
                      value="Evacuation" />
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:element>
              <xsd:element name="keys">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="item"
                      minOccurs="5" maxOccurs="5">
                      <xsd:complexType>
                        <xsd:sequence>
                          <xsd:element
                            name="key">
                            <xsd:simpleType>
                              <xsd:restriction
                                base="xsd:string">
                                <xsd:enumeration
                                  value="Domicile.domicileID">
                                </xsd:enumeration>
                                <xsd:enumeration
                                  value="Domicile.Emergency.
                                    emergencyType">
                                </xsd:enumeration>
                                <xsd:enumeration
                                  value="Domicile.Emergency.room
                                    ">
                                </xsd:enumeration>
                                <xsd:enumeration
                                  value="Domicile.Emergency.
```

```
                                    component">
                                </xsd:enumeration>
                                <xsd:enumeration
                                   value="transactionID">
                                </xsd:enumeration>
                              </xsd:restriction>
                            </xsd:simpleType>
                          </xsd:element>
                          <xsd:element
                            name="value" type="xsd:string" />
                        </xsd:sequence>
                      </xsd:complexType>
                    </xsd:element>
                  </xsd:sequence>
                </xsd:complexType>
                <xsd:key name="PK_item_key">
                  <xsd:selector xpath=".//item" />
                  <xsd:field xpath="key" />
                </xsd:key>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Um eine Validierung eines *Events* oder *DataTransferObjects* anhand einer solchen XML Schema Definition durchzuführen, muss die übertragene SOAP-Nachricht im XML-Format vorliegen. Deshalb erscheint eine Implementierung des Interfaces `System.ServiceModel.Dispatcher.IDispatchMessageInspector` sinnvoll, da hier auf die SOAP-Nachricht zugegriffen werden kann, noch bevor die Nachricht deserialisiert wurde. Das Interface sieht wie folgt aus:

Listing 5: `IDispatchMessageInspector`-Interface

```
public interface IDispatchMessageInspector
{

  object AfterReceiveRequest(ref Message request, IClientChannel
      channel, InstanceContext instanceContext);

  void BeforeSendReply(ref Message reply, object
     correlationState);

}
```

Für eine Implementierung, die sich der Möglichkeit der Validierung mittels XML Schema bedient, wird an dieser Stelle auf den Artikel “Message Inspectors”[5] aus der MSDN Library[6] verwiesen.

Um die Möglichkeit zu schaffen auf Operationsebene unterschiedliche XML Schemata zu laden, kann die Methode `ValidateMessageBody(...)` aus [5] erweitert werden und

zunächst durch die Eigenschaft `message.Headers.Action` die aufgerufene Operation ermittelt werden.

Um unterschiedliche XML Schemadefinitionen auf Basis des Typen eines *DataTransferObjects* oder *Events* zu laden, kann der Typ in der Methode `ValidateMessageBody(...)` folgendermaßen ermittelt werden:

Listing 6: Ermittlung des Typen eines Events

```
DataContractSerializer dcs = new DataContractSerializer(typeof(
    Event));
Event evt = (Event) dcs.ReadObject(message.
    GetReaderAtBodyContents());
string type = evt.Type;
```

Weitere Informationen zu einer möglichen Implementierung der *Contracts* finden sich auch im Konzeptpapier zur Validierung von Parametertypen[31].

9.1.3 Formalisierung einer Struktur zur abstrakten Parameter- und Eventdarstellung

Ein wichtiges Ziel dieser Projektgruppe ist es, die Systeme *IGM-Plattform* und das *jABC* zu einem Gesamtsystem zur Realisierung der *SmarterWohnen*-Geschäftsprozesse zu verbinden. Dabei spielen Informationen, die zwischen diesen heterogenen Systemen ausgetauscht werden, eine bedeutende Rolle. In der Analysephase wurden diese Informationen sowohl als Ein- und Ausgabeparameter von Methoden als auch als Events identifiziert. Um die Entwicklung möglichst unabhängig voneinander auf den beiden Systemen vorantreiben zu können, müssen Parameter und Events gleichermaßen strukturiert werden, um anschließend einen sie eindeutig identifizierenden Namensraum formalisieren zu können. Durch diese Abstraktion von tatsächlich auf den Systemen implementierten Namensräumen und typisierten Datenstrukturen können einerseits im Rahmen der Modellierung von Prozessen im *jABC* Parameter und Events präzise benannt und das Modell auf diese Weise konkretisiert werden; andererseits ist dieser Formalismus dazu geeignet, als gemeinsame Sprache innerhalb der in Abschnitt 9.1 beschriebenen Webservice-Schnittstelle zu dienen.
Der konzeptionelle Grundgedanke sieht eine Baumstruktur mit Zugriff auf Datenfelder über Schlüsselworte vor. Es gibt vier Äste in der Struktur, die die Bewohnerdaten (*User*), die Dienstleisterdaten (*ServiceProvider*), die Wohnungsdaten (*Domicile*) und die Daten der Nachrichten (*Message*) bereitstellen. Über die weiteren Verzweigungen wird auf die Datenfelder in den Ästen für Notfalldaten (*Emergency*) oder weitere verschachtelte Datenfelder zugegriffen. Für jeden Ast existiert ein Schlüsselattribut um einen Datensatz eindeutig zu identifizieren. Die Struktur wird in Abbildung 29 dargestellt.

Ein Bewohner kann also über seine eindeutige `UserID` identifiziert werden. Aus dem dadurch spezifizierten Datensatz können nun Informationen, wie Name, Vorname oder die eindeutige `DomicileID` gelesen oder geschrieben werden. Über die `DomicileID` wird die entsprechende Wohnung identifiziert. Hier kann auf weitere Daten wie Straße oder Postleitzahl zugegriffen werden. Je nachdem ob ein eintretender Notfall mit einem Bewohner (z.B. Herzstillstand) oder mit der Wohnung (z.B. Einbruch, Brand) assoziiert ist, wird über den Unterzweig *Emergency* im jeweiligen Ast der verantwortliche Dienstleister (`ServiceProviderID`) und die gewünschte Art der Bewohnerbenachrichtigung ausgelesen. Der Ast *ServiceProvider* enthält die notwendigen Dienstleisterdaten. Die mit einem *Event* übertragenen Daten sind auf die gleiche Weise spezifiziert. Dadurch stehen

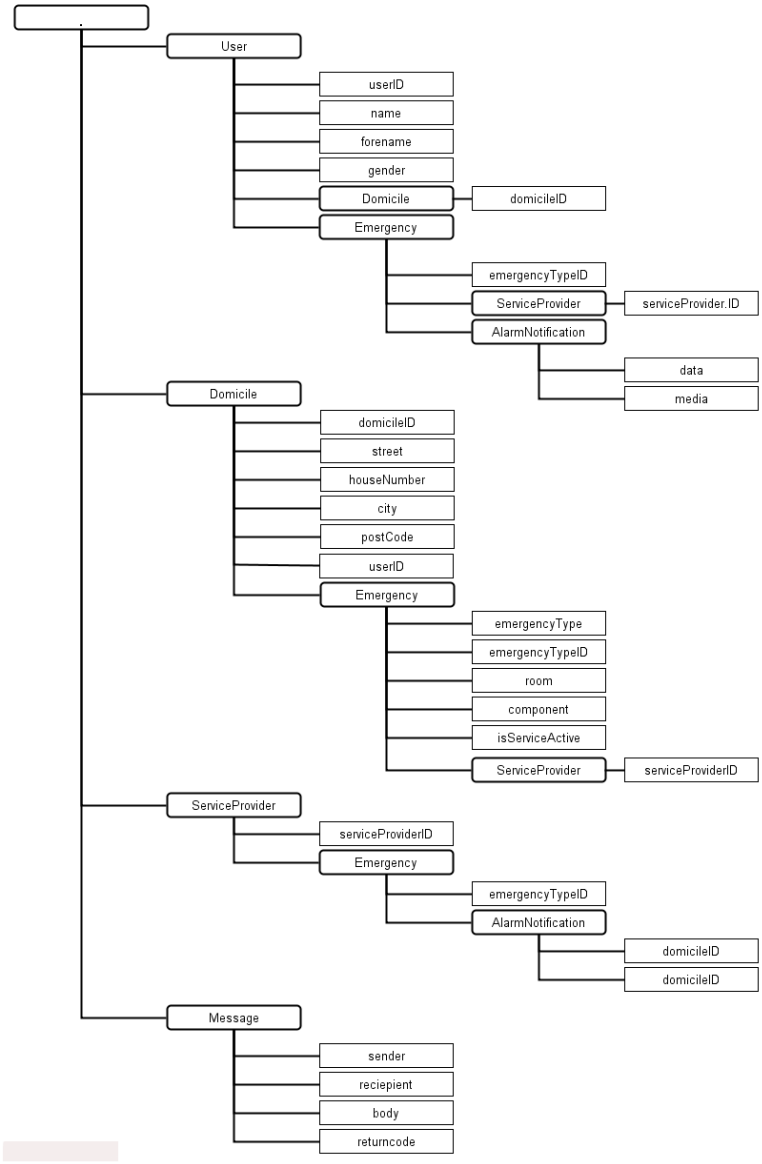


Abbildung 29: Parameter- und Eventdarstellung als Baumstruktur

sie im Prozesskontext sofort zur Verfügung.

Sowohl die *jABC*-Seite, als auch die Seite der *IGM-Plattform* müssen dieses Konzept implementieren. Auf diese Weise erhält man die nötige Freiheit, die Stärken beider Systeme auszunutzen und dabei nur eine minimale Datenmenge über den verbindenden Webservice zu übertragen. Konzeptionell lässt sich diese Datensprache erweitern, um auch andere Projekte als *SmarterWohnen* zu unterstützen. Hier muss allerdings notwendigerweise ein auf die entsprechenden Daten ausgerichtetes Modell etabliert werden (siehe 10). Die technischen Einzelheiten werden in Abschnitt 9.1 diskutiert.

Entwicklung von DataTransferObjects (DTOs)

Die Events, die über den Webservice übertragen werden, sind in Kapitel 9.1 dargestellt. Aus der dargestellten Parametertaxonomie lassen sich Datenpakete definieren. Die technischen Grundlagen wurden in Kapitel 9.1.1 diskutiert. Durch den Webservice stehen außer der Funktion *sendEvent* noch drei weitere Grundfunktionen zur Verfügung. Die

Funktion *retrieveData* wird benutzt um Daten von der *IGM-Plattform* auszulesen. Mittels *sendData* werden Daten zu der *IGM-Plattform* übertragen. *sendMessage* verschickt eine Nachricht über die *IGM-Plattform*. Sowohl beim Lesen als auch beim Schreiben von Daten muss darauf geachtet werden, dass genügend Schlüsselattribute übertragen werden um die Datenfelder ausreichend zu identifizieren. Der eindeutige Name des *DTO*'s ist auch ein Schlüsselattribut.

Um die Lesbarkeit zu Verbessern beginnen Bezeichner der Äste mit Großbuchstaben und die Bezeichner der Datenfelder in den Blättern mit kleinen Buchstaben.

Um die Kommunikation zwischen den beiden Systemen besser überwachen und nachvollziehen zu können wird in jedem Datenpaket eine eindeutige *transactionID* mitgesendet. Die *transactionID* wird von dem System erzeugt, das den initialen Event versendet. Die Datenpakete *IGMPlatform.Time* und *IGMPlatform.Status* wurden eingefügt um die technische Funktionalität des Gesamtsystems zu testen. Sie sind hier nur der Vollständigkeit wegen aufgeführt.

Bei der Entwicklung der *DTO*s wurde darauf geachtet, möglichst elementare Datenpakete zu finden. Dadurch werden einerseits mehr Funktionsaufrufe auf dem Webservice nötig, aber es wird eine größere Flexibilität erreicht. Die Funktion *sendMessage* ist der Funktion *sendData* zwar ähnlich, aber hier wurde die Trennung der beiden Methoden gewählt um das Verständnis für das System zu vereinfachen und den logischen und fachlichen Unterschied deutlich zu machen.

Für die Funktion *retrieveData* werden die Datenpakete in Tabelle 6 dargestellt. Die Datenfelder die gelesen werden sollen sind in der Tabelle mit "lesen" gekennzeichnet; die Datenfelder, die als Schlüssselattribute übertragen werden, sind als "Schlüssel" gekennzeichnet.

Für die Funktion *sendData* wurden zwei Datenpakete erstellt. Die *IGM-Plattform* unterstützt das Schreiben in die Datenbank nur bedingt, sodass zur Zeit nicht mehr Funktionalität umgesetzt werden konnte. Der Aufbau, die Schlüsselattribute und die zu lesenden Datenfelder sind in Tabelle 7 aufgeführt.

Die Funktion *sendMessage* benötigt ein eigenes Datenpaket für jede Methode mit der eine Nachricht verschickt werden soll. Zur Zeit unterstützt die *IGM-Plattform* nur das Versenden von E-Mails und von SMS (ShortMessageService). Der eindeutige Name des *DTO* reicht als Schlüsselattribut aus. Der Rückgabewert (`Message.returncode`) zeigt sofort an, ob die Nachricht erfolgreich übergeben werden konnte. Diese Information steht in dem Prozess zur Verfügung. Der Prozess kann mit dieser Information über den weiteren Prozessablauf entscheiden. Die Datenpakete sind in Tabelle 8 aufgeführt.

9.2 Subsystem: Erweiterung IGM-Plattform

9.2.1 Identifikation der Komponenten

Wie im Gesamtkonzept dargestellt, soll die *IGM-Plattform* hauptsächlich für informationslogistische Aspekte der Prozesse genutzt werden. Es werden also Komponenten gebraucht, die die gewünschten Kontext- und Benutzerdaten zur Verfügung stellen und die relevante Ereignisse an das *jABC* weiterleiten. Folgende Komponenten wurden identifiziert:

BPMProcessHandler Der *BPMProcesshandler* ist der Webservice, der die Anfragen vom *jABC* entgegennimmt und an das eigentliche *BPM-Subsystem* weiterleitet. Diese Komponente ist kein direkter Bestandteil des *BPM-Subsystems*, sondern eher eine Art vorgelagerte Kommunikationsschicht. Sie dient hauptsächlich dazu, die Anfragen vom *jABC* entgegen zu nehmen und je nach Operationsaufruf die empfangenen Daten aufzubereiten und an die jeweils zuständige Manager-Instanz im *BPM-Subsystem* weiter zu leiten.

JABCWSProxy Diese Komponente stellt einen Proxy für den *jABC*-Webservice (*jABC-Processhandler*) dar. Sie kapselt die eigentlichen Aufrufe des *jABC*-Webservices und kann von allen anderen Komponenten (vor allem von den Managern) zur Kommunikation mit dem *jABC* genutzt werden.

IGMEventManager Die Komponente kapselt sämtliche Funktionalitäten, die Ereignisse innerhalb der *IGM-Plattform* betreffen. Hierzu zählt insbesondere die Behandlung der auf der Plattform aufgetretenen Ereignisse. Diese können z.B. über den *JABCWSProxy* an das *jABC* weitergeleitet werden. Der *IGMEventManager* muss, um Ereignisse von anderen Subsystemen empfangen zu können, eine Subkomponente beinhalten, die die Ereignisse aus der MSMQ holt und verarbeitet. Außerdem müssen auf Grundlage der Contracts (s. Kapitel 9.1.2) entsprechende Subscriptions und Request an den anderen Subsystemen für Ereignisse gemacht werden.

WSEventManager Der *WSEventManager* übernimmt die Abarbeitung aller Ereignisse, die über den *BPMProcessHandler* vom *jABC* an das *BPM-Subsystem* gesendet werden. Je nach aufgetretenem Ereignis ist er dafür zuständig, vorher definierte Aufgaben jeglicher Art auszuführen. Darüber hinaus wird innerhalb des *WSEventManagers* eine Subkomponente innerhalb des *BPM-Subsystems* benötigt, die aufgetretene Ereignisse an die anderen Subsysteme über MSMQ weitergeben kann, sofern diese angefordert wurden (vgl. *IGMEventBroker*.

DataManager Der *DataManager* tritt als eine Art Proxy gegenüber dem *Context-* und dem *User-Subsystem* auf. Anfragen, die mittels `sendData` und `retrieveData` über den *BPMProcesshandler* an das *BPM-Subsystem* gesendet werden, werden an diesen weitergereicht und entsprechend verarbeitet. Er ist somit für die Ermittlung und Modifikation von Kontextinformationen verantwortlich.

CommunicationManager Der *CommunicationManager* tritt, ähnlich dem *DataManager*, als eine Art Proxy gegenüber dem *DIA-Subsystem* auf. Nachrichten, die mittels `sendMessage` über den *BPMProcessHandler* an das *BPM-Subsystem* gesendet werden, werden an diese Komponente weitergereicht und entsprechend verarbeitet.

IGMEventBroker Der *IGMEventBroker* dient als Schnittstelle für die asynchrone Kommunikation mit den anderen Subsystemen. Er bildet die Schnittstelle zur MSMQ und ist somit für das Empfangen und Versenden von Nachrichten bzw. Ereignisse von und zu den anderen Subsystemen verantwortlich. Auch die Behandlung und Verwaltung von Requests fällt in den Zuständigkeitsbereich dieser Komponente.

Erweiterungen Eventuell ist es notwendig, weitere Manager zu definieren. Über das *Service-Subsystem* bereitgestellte Dienste werden momentan nicht durch einen Manager gekapselt. Es besteht daher evtl. die Möglichkeit bzw. Notwendigkeit, dass es eine Art *ServiceManager* geben muss. Aufgrund der serviceorientierten Architektur des *BPM-Subsystems*, sollte dies aber auch nachträglich keine Probleme darstellen.

Abbildung 30 zeigt eine schematische Darstellung der Komponenten.

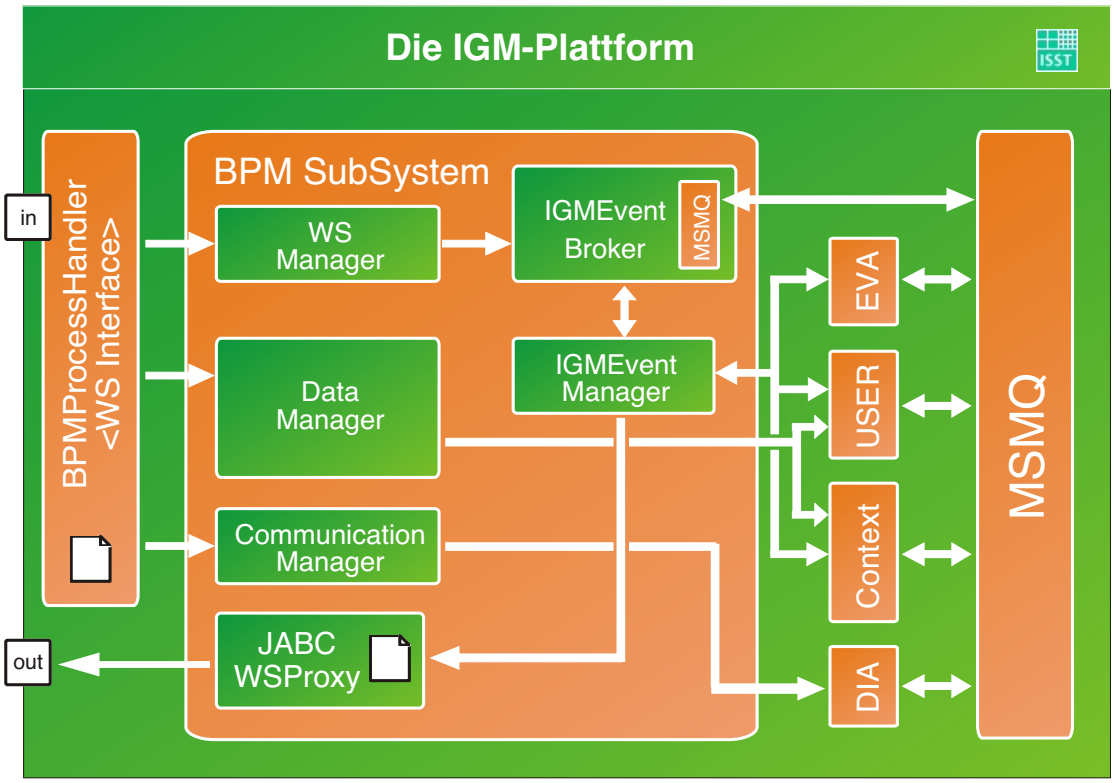


Abbildung 30: Schematische Darstellung der Komponenten

9.2.2 Detailbeschreibung der Komponenten

Nachfolgend sollen die in Kapitel 9.2.1 identifizierten Komponenten genauer beschrieben werden. Gleichzeitig wird das Zusammenspiel der Komponenten etwas genauer betrachtet. Die Abbildung 31 vermittelt einen ersten Überblick und ist mehr als eine Art Darstellung der Zusammenhänge zwischen den Komponenten anstatt eines Klassendiagramms zu verstehen.
Die nachfolgenden Schnittstellenbeschreibungen sind konzeptioneller Art. Bei einer Implementierung müssen Schnittstellen zur Verfügung gestellt werden, die die beschriebenen Funktionalitäten umsetzen. Eine detailierte Beschreibung der implementierten Schnittstellen findet sich in der API-Dokumentation zum *BPM-Subsystem*[42].

BPMProcessHandler Für die Kommunikation mit dem *jABC* werden die folgenden Methoden zur Verfügung gestellt:

- `:sendEvent(event)`
- `:sendData(type, inData)`
- `outData:retrieveData(type, inData)`

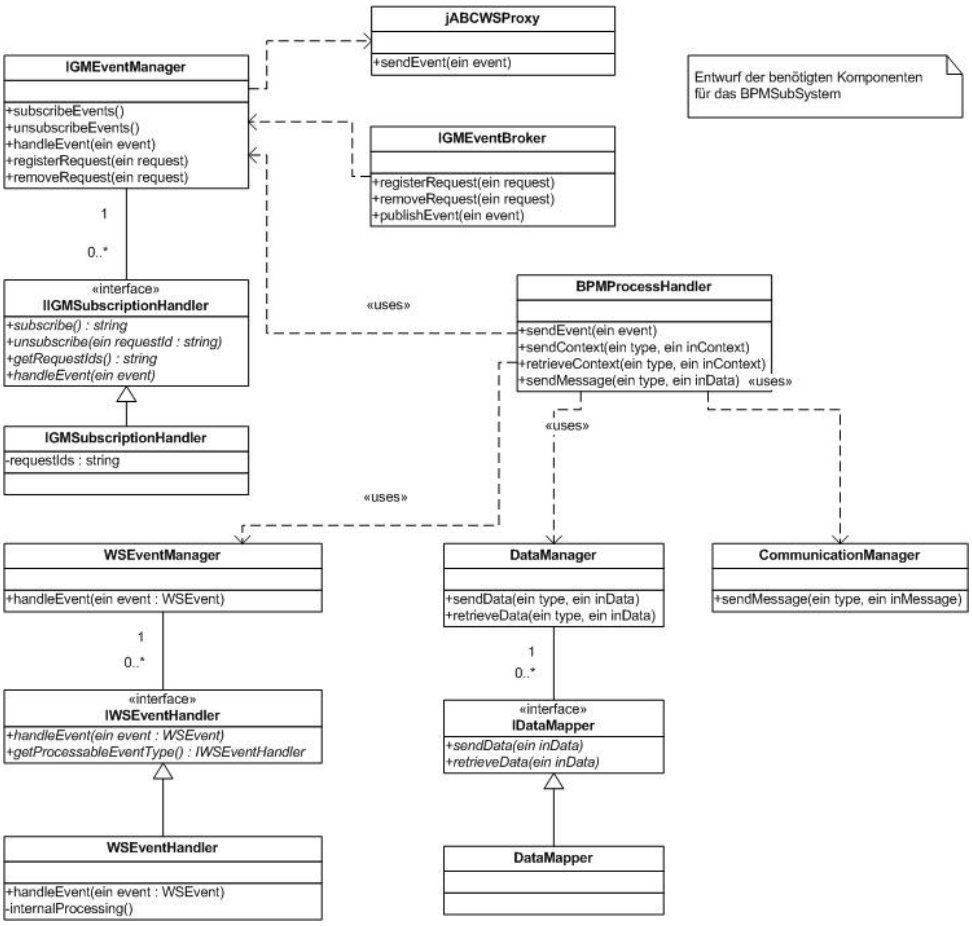


Abbildung 31: Komponenten Übersicht und Zusammenhänge

- `:sendMessage(type, inData)`

Die Bedeutung der Methoden und ihre Funktionalität werden nachfolgend erläutert:

sendEvent

- **Eingabeparameter**
  - `event` - Event, der vom *jABC* an das *BPM-Subsystem* gesendet werden soll
- **Rückgabewert** keine
- **Funktionalität** Die Methode dient dazu einen auf *jABC*-Seite ausgelöstes Ereignis an das *BPM-Subsystem* weiterzuleiten. Das Ereignis wird durch die Methode entgegengenommen und an den *WSEventManager* des *BPM-Subsystems* weitergeleitet.

sendData

- **Eingabeparameter**
  - `type` - Der Typ des zu ändernden Kontextes
  - `inData` - Die neuen Kontextdaten, die gesetzt werden sollen, in Form von Schlüssel-Wert-Paaren
- **Rückgabewert** keine

- **Funktionalität** Die Methode dient dazu, von *jABC*-Seite aus den Kontext auf der *IGM-Plattform* zu manipulieren. Der Parameter `type` beschreibt hierbei den Kontexttyp. Momentan sind zwei Kontexttypen vorgesehen, nämlich Benutzerinformationen (vorgehalten im *User-Subsystem*) und Kontextinformationen (vorgehalten im *Context-Subsystem*). Die entgegengenommenen Parameter werden an den *DataManager* des *BPM-Subsystems* zur Verarbeitung weitergegeben.

retrieveData

- **Eingabeparameter**
  - `type` - Der Typ der angeforderten Kontextinformationen
  - `inData` - Die angeforderten Kontextdaten und Kontextdaten zur Ermittlung des angeforderten Kontextes, in Form von Schlüssel-Wert-Paaren
- **Rückgabewert** Die angeforderten Kontextinformationen als Liste von Schlüssel-Wert-Paaren.
- **Funktionalität** Die Methode dient dazu, von *jABC*-Seite aus Kontextinformationen (Benutzer oder Kontext) aus der *IGM-Plattform* abzurufen. Die benötigten Kontextinformationen werden im Parameter `inData` als Schlüssel übergeben. Als Wert wird ein Leerstring übergeben. Desweiteren kann der Parameter `inData` Schlüssel enthalten, zu denen auch ein Wert angegeben wird. Diese Informationen dienen dem Auffinden der angefragten Kontextinformationen innerhalb der *IGM-Plattform*. Wenn z.B. eine E-Mail-Adresse angefragt wird, wird ein Schlüssel `eMail` ohne Wert angegeben und ein Schlüssel `userId` mit dem Wert der ID eines Benutzers. Somit kann anhand der `userId` die E-Mail-Adresse ermittelt werden.
  Angeforderte Kontextinformationen werden über den Rückgabewert an das *jABC* zurück gegeben.

sendMessage

- **Eingabeparameter**
  - `type` - Der Typ der zu versendenden Nachricht (E-Mail oder SMS)
  - `inData` - Die Nachricht und weitere für den Versand benötigte Daten, in Form von Schlüssel-Wert-Paaren
- **Rückgabewert** keine
- **Funktionalität** Die Methode dient dazu, Nachrichten an Benutzer oder Dienstleister zu senden. Die Nachricht ist als Schlüssel-Wert-Paar im Parameter `inData` vorhanden. Weitere Informationen wie die E-Mail-Adresse oder die Telefonnummer können ebenfalls mittels dieses Parameters übergeben werden. Der Parameter *type* gibt den Typ der Nachricht an (z.B. SMS oder E-Mail).

JABCWSProxy Der *JABCWSProxy* stellt eine Kapselung des Webservices des *jABC* dar. Alle Komponenten, die mit dem *jABC* kommunizieren müssen, sollen zu diesem Zweck diesen Proxy benutzen.

Die Klasse *JABCWSProxy* stellt die folgenden Methoden zur Verfügung:

- `:sendEvent(event)`

Die Bedeutung der Methoden und ihre Funktionalität werden nachfolgend erläutert:

sendEvent

- **Eingabeparameter**
  - `event` - Ereignis, das vom *BPM-Subsystem* an das *jABC* gesendet werden soll
- **Rückgabewert** keine
- **Funktionalität** Die Methode dient dazu ein auf der*IGM-Plattform* ausgelöstes Ereignis an das *jABC* weiterzuleiten.

WSEventManager Die Komponente *WSEventManager* übernimmt die Abarbeitung aller Ereignisse, die über den *BPMProcesshandler* vom *jABC* an das *BPM-Subsystem* gesendet werden. Zu diesem Zwecke stellt der *WSEventmanager* die folgende Methoden zur Verfügung:

- `:handleEvent(event)`

Die Bedeutung der Methoden und ihre Funktionalität werden nachfolgend erläutert:

handleEvent

- **Eingabeparameter**
  - `event` - Ereignis, das vom *jABC* an das *BPM-Subsystem* gesendet wurde
- **Rückgabewert** keine
- **Funktionalität** Die Methode dient dazu ein vom *jABC* gesendetes Ereignis abzuarbeiten. Zu diesem Zweck wird anhand des Ereignistyps ein entsprechender *WSEventHandler* ermittelt (Methode `IWSEventHandler.getProcessableEventType()` und die Abarbeitung des Ereignises an diesen mittels der Methode `handleEvent` übertragen.
  Für den Fall, dass andere Subsysteme der *IGM-Plattform* Ereignisse abonniert haben, wird das Ereignis mittels der Methode `publishEvent` an den *IGMEventBroker* weitergereicht.

IGMEventManager Die Komponente *IGMEventManager* kapselt sämtliche Funktionalitäten, die Ereignisse innerhalb der *IGM-Plattform* betreffen. Auf Basis der *Contracts* ist der *IGMEventmanager* beim Starten des *BPM-Subsystems* dafür zuständig, die konfigurierten Requests an die einzelnen Subsysteme zu machen, um von diesen generierte Ereignisse zu abonnieren. Zu diesem Zweck wird die Methode `subscribeEvents` bereit gestellt. Die Methode lädt für jeden konfigurierten Ereignistyp einen entsprechenden *IGMSubscriptionHandler*, der wiederum die eigentlichen Requests an den Subsystemen registriert (Methode `subscribe`). Beim Stoppen des *BPM-Subsystems* müssen diese Abonnements wieder zurück genommen werden. Hierzu dient die Methode `unsubscribeEvents` am *IGMEventmanager* bzw. `unsubscribe` an dem zuständigen *IGMSuscriptionHandler*. Sobald ein Ereignis auf der *IGM-Plattform* auftritt, wird dieser über den *IGMEventbroker* an den *IGMEventmanager* weitergeleitet. Dazu stellt dieser die Methode `handleEvent` bereit. Diese Methode ermittelt den zuständigen *IGMSubscriptionHandler* und überlässt diesem die Abarbeitung des Ereignises.

Um o.g. Funktionaliät bereit zu stellen, bietet die Klasse *IGMEventManger* folgenden Methoden an:

- `:subscribeEvents`
- `:unsubscribeEvents`
- `:handleEvent(event)`

- `:registerRequest(request)`
- `:removeRequest(request)`

Die Methoden werden nachfolgend detaillierter beschrieben:

subscribeEvents

- **Eingabeparameter** keine
- **Rückgabewert** keine
- **Funktionaliät** Die Methode instanziiert für jeden konfigurierten Ereignistyp eine *IGMSubscriptionHandler*-Instanz und ruft an dieser die Methode `subscribe` auf, um die entsprechenden Ereignisse an den Subsystemen zu abonnieren.

unsubscribeEvents

- **Eingabeparameter** keine
- **Rückgabewert** keine
- **Funktionalität** Die Methode ruft an jeder *IGMSubscriptionHandler*-Instanz die Methode `unsubscribe` auf, um die durch Request abonnierten Ereignisse wieder abzubestellen.

handleEvent

- **Eingabeparameter**
  - `event` - Ereignis, das auf der *IGM-Plattform* generiert wurde
- **Rückgabewert** keine
- **Funktionalität** Die Methode dient dazu ein über den *IGMEventbroker* empfangenes Ereignis von der *IGM-Plattform* zu verarbeiten. Die Methode sucht anhand der im Event vorhandenen Request-ID den entsprechenden *IGMSubscriptionHandler* und übergibt diesem mittels der Methode `handleEvent` das Ereignis zur Verarbeitung. Zumeist wird dieser dann das Ereignis an das *jABC* mit Hilfe des *JABCWSProxy* weiterleiten.

registerRequest

- **Eingabeparameter**
  - `request` - Request eines anderen Subsystems zum Abonnieren von Ereignissen
- **Rückgabewert** keine
- **Funktionalität** Die Methode nimmt ein Request eines anderen Subsystems zum Abonnieren von Ereignissen entgegen und gibt dieses zur Abarbeitung an den *IGMEventbroker* mittels `registerRequest` weiter.

removeRequest

- **Eingabeparameter**
  - `request` - Request eines anderen Subsystems zum Deabonnieren von Ereignissen
- **Rückgabewert** keine
- **Funktionalität** Die Methode nimmt ein Request eines anderen Subsystems zum Deabonnieren von Ereignissen entgegen und gibt dieses an den *IGMEventbroker* mittels `removeRequest` weiter.

DataManager Die *DataManager* Komponente kapselt den Zugriff auf das *User-* und *Context-Subsystem*. Die Aufgabe der Komponente besteht in der Transformation der fachlichen Kontext- und Benutzerdaten-Anfragen seitens der *jABC-Plattform* in technische Kontext- und Benutzerdaten-Anfragen der *IGM-Plattform*. Die Abbildung des Typs, in Form eines `type`-Parameters, einer fachlichen auf eine technische Anfrage und zurück, wird durch `IDataMapper` übernommen. Folgenden Methoden der Klasse `DataManager` beschreiben den Zugriff:

sendData

- **Funktionalität** Die Methode *sendData* wurde im Vergleich zum ursprünglichen Ansatz in der Funktionalität drastisch eingeschränkt. Die *IGM-Plattform* sieht nicht vor, dass man die *allgemeinen* Daten, wie z.B. Benutzer-, Wohnungs- oder Servicedienstleisterdaten über diese direkt manipulieren oder erfassen kann. Daher gibt es insgesamt auch nur zwei DTOs, die die Daten über die Logging-Funktion in die Datenbank schreiben.
- **Eingabeparameter**
  - `inData` - Die Daten in `inData` sind als Schlüssel-Wert-Paare formatiert, daher wird diese Formatierung für den String übernommen, wobei die einzelne Paare durch Komma und einen Zeilenumbruch getrennt sind.
- **Rückgabewert**
  - `true` - wenn die Schreiboperation erfolgreich war.
  - `false` - sonst

retrieveData

- **Funktionalität** Der generische Ansatz aus dem Zwischenbericht wurde verworfen. Der Grund dafür war hauptsächlich die eingeschränkten Möglichkeiten in der Datenbereitstellung der *IGM-Plattform*. Der aktuelle, statische Ansatz sieht vor, dass das eingehende `inData`-Objekt einen eigenen Typen definiert, Attribute `Type`, eine Liste von Schlüsseln enthält anhand dessen man den gewünschten Datensatz identifizieren kann. Ein Schlüssel ist immer notwindig, weil die *IGM-Plattform*, im speziellen -User und Service-Subsystem, keinen unbeschränkten Datenzugriff erlauben. Weiterhin muss das `inData`-Objekt eine Liste der `ReturnData` enthalten. Diese Liste ist nicht notwendig, weil die Rückgabewerte implizit gegeben sind, aber sie ist hilfreich in der Entwicklungsphase für das Auffinden von Tippfehlern und Missverständnissen bei Parameternamen.
  Die Spezifikationen der DTOs kann man im Dokument [7] nachlesen.
- **Eingabeparameter**
  - `inData` - das Objekt muss seinen DTO Typen definieren, eine Liste, mit Länge größer 0, von Schlüsseln enthalten, sowie eine Liste von ReturnData-Schlüsseln (Beschreibung s. o.).
- **Rückgabewert** Es werden die angefragten Daten zurückgeliefert, die anhand der eingegangenen Daten identifiziert werden konnten. Im Fall, dass die gesuchten Daten nicht vorhanden sind, wird das eingegangene `De.UniDo.Poets.Isst.IGM.BpmSubSystem.Common.Contracts.Data.DataTransferObject` Objekt zurück gegeben. Sollte im System ein unbekannter Fehler auftreten so ist die Rückgabe `null`. Die Rückgabe `null` für den Fehlerfall wurde aus zwei Gründen gewählt, der fehlenden Dokumentation bei IGM-Plattform und weil

die Notwendigkeit der Definition weiterer SOAP-Faults für den Webservice damit umgangen wird.

CommunicationManager Die *CommunicationManager* Komponente kapselt den Zugriff auf das *DIA-Subsystem*. Die Komponente arbeitet *jABC*-Nachrichten für das *DIA-Subsystem* auf.

sendMessage

- **Eingabeparameter**
  - `type` - fachliche ID des Message-Parameters. Das kann z. B. ein *SMS*- oder ein *E-Mail*-Typ sein.
  - `inMessage` - die eigentliche Nachricht.
- **Rückgabewert** Ein boolescher Wert. `true`, wenn die Nachricht erfolgreich verschickt werden konnte, sonst `false`.

IGMEventBroker Der *IGMEventbroker* dient als Schnittstelle für die asynchrone Kommunikation mit den anderen Subsystemen. Er überwacht die MSMQ und empfängt über diese einkommende Ereignisse. Jedes eingehende Ereignis wird an den *IGMEventhandler* mittels der Methode `handleEvent` weitergereicht und die Abarbeitung von diesem übernommen. Desweiteren wird eine Methode `publishEvent` bereit gestellt, über die Ereignisse an andere Subsysteme versendet werden können. Um festzustellen, welche Ereignisse an welche Subsysteme zugestellt werden sollen, wird eine Liste von Requests verwaltet. Über die Methoden `registerRequest` und `removeRequest` ist es den anderen Subsystemen möglich, über Requests entsprechende Ereignisse zu abonnieren oder Abonements zurück zu nehmen.

Die *IGMEventBroker* stellt die folgenden Methoden zur Verfügung:

- `:publishEvent(event)`
- `:registerRequest(request)`
- `:removeRequest(request)`

Die Methoden werden nachfolgend detaillierter beschrieben:

publishEvent

- **Funktionalität** Die Methode nimmt einen Ereignis entgegen, um diesen an andere Subsysteme asynchron mittels MSMQ zu versenden. Dabei wird geprüft, welche Subsysteme ein Abonnement auf Ereignisse des Ereignistyps haben, und das Ereignis nur an diese Subsysteme übermittelt.
- **Eingabeparameter**
  - `event` - ein Ereignis, das an andere Subsysteme versendet werden soll
- **Rückgabewert** keine

registerRequest

- **Funktionalität** Über diese Methode ist es anderen Subsystemen möglich, Ereignisse zu abonnieren. Die Methode wird vom *IGMEventmanager* aufgerufen.
- **Eingabeparameter**
  - `request` - ein Request eines anderen Subsystems zum Abonnieren von Ereignissen
- **Rückgabewert** keine

removeRequest

- **Funktionalität** Über diese Methode ist es anderen Subsystemen möglich, Abonnements auf Ereignisse zu kündigen. Die Methode wird vom *IGMEventmanager* aufgerufen.
- **Eingabeparameter**
  - `request` - ein Request eines anderen Subsystems um ein Ereignis-Abonnement zu kündigen
- **Rückgabewert** keine

9.3 Prozessmanagement: Erweiterung jABC

9.3.1 Einleitung

Wie im Gesamtkonzept bereits erläutert, soll *jABC* dazu benutzt werden, Prozesse zu modellieren und auszuführen. Diese sollen dann über den in Kapitel 9.1 beschriebenen Webservice mit der *IGM-Plattform* kommunizieren. Die *IGM-Plattform* leitet dabei alle fachlichen Events an die Prozesse weiter, die wiederum zur Erfüllung ihrer Aufgabe Events werfen, Daten anfordern und senden und Aktionen wie z.B. das Schicken einer E-Mail an einen übergebenen Empfänger auslösen können.
Die *jABCProzessPlattform* verwaltet nun diese Prozesse und ihre Kommunikation mit der *IGM-Plattform*. Sie startet die Prozesse, weist ihnen die über den Webservice erhaltenen Events und Daten zu und pausiert sie, sofern der Prozess es zulässt, um Ressourcen zu sparen.
Die folgenden Abschnitte sollen nun die Komponenten der *jABCProzessPlattform* vorstellen und deren Arbeitsweise erläutern.

9.3.2 Identifikation der Komponenten

Die Abbildung 32 gibt einen Überblick über die Kernkomponenten der *jABCProzessPlattform*. Im Folgenden werden sie einzeln vorgestellt.

Der jABCProcessHandler Die Klasse `jABCProcessHandler` stellt die Webservice-Schnittstelle zur Verfügung. Sie wandelt dabei interne Aufrufe in Webservice-Methoden um. Sie stellt somit Service-Methoden zur Verfügung, damit Prozesse Daten anfordern bzw. senden, Events erzeugen oder auch Aktionen auf der *IGM-Plattform* auslösen können. Des Weiteren leitet sie alle hereinkommenden Events an den `ProcessManager` weiter.

Der ProcessManager Die Klasse `ProcessManager` stellt das Herz der *jABCPlattform* dar. Sie verwaltet alle laufenden Prozesse auf der *jABCPlattform*. Dazu stellt sie Service-Methoden zum Starten und Pausieren von Prozessen bereit, ermöglicht Prozessen das Abonnement von Events und leitet die vom `jABCProcessHandler` übergebenen Events an die entsprechenden Prozesse weiter, die diese zuvor abonniert haben.
Das Behandeln verschiedener Eventtypen wird im Kapitel 9.3.3 erläutert.

Der Process Ein `Process` kapselt eine *jABC*-Prozess Instanz auf der *jABCProzessPlattform*. Er bietet dem `ProcessManager` Service-Methoden zur aktiven Beeinflussung des Prozesses an (z.B. Eventbenachrichtigung). Er ist eindeutig über eine vom `ProcessManager` vergebene *ProcessID* gekennzeichnet.

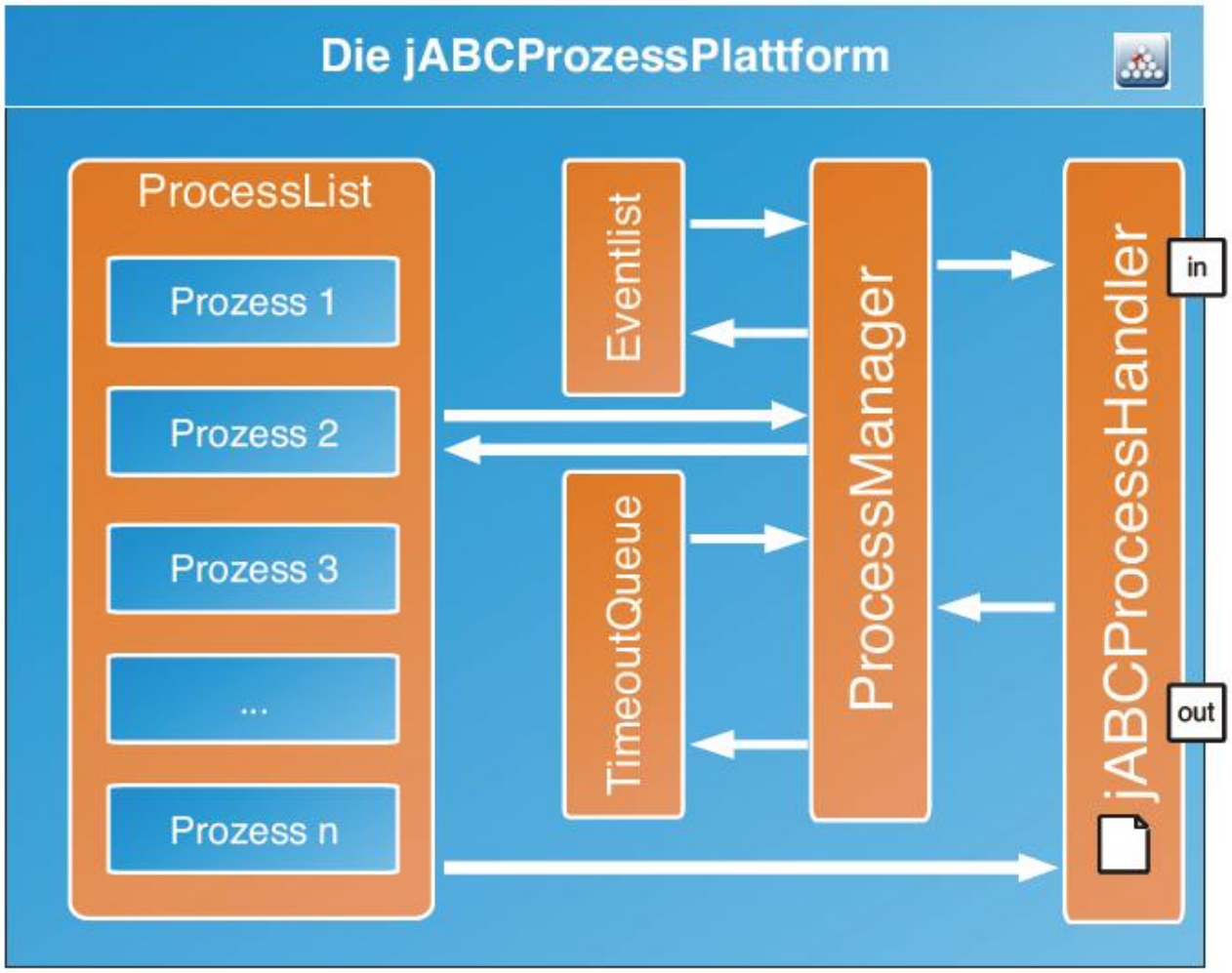


Abbildung 32: Übersicht *jABCProzessPlattform*

Die ProcessList Die `ProcessList` beinhaltet eine Liste aller auf der *jABCProzessPlattform* laufenden Prozesse.

Die EventList Die `EventList` dient der Abonnementverwaltung von Events für Prozesse. Ein Prozess kann während seiner Ausführung Events, die in einem bestimmten Kontext auftreten, abonnieren und und deren Abonnement löschen. Nur falls ein Prozess ein Event abonniert hat, wird er vom `ProcessManager` benachrichtigt, wenn ein solches Event auftritt.

Die TimeoutQueue Die `TimeoutQueue` verwaltet alle pausierten Prozesse der *jABCProzessPlattform*. Ein Prozess kann sich selber pausieren, wenn er z.B. auf ein Event wartet. Daraufhin wird seine Abarbeitung angehalten und er wird erst wieder fortgesetzt, wenn ein solches Event aufgetreten oder eine maximale Wartezeit (Timeout) erreicht wurde.

9.3.3 Die Funktionsweise der Komponenten im Einzelnen

Nach dieser kurzen Übersicht über die Kernkomponenten der *jABCProzessPlattform*, stellt sich nun die Frage, wie diese Komponenten zueinander in Beziehung stehen. Am Besten

zeigt man dies, indem man beschreibt, welche Aufgaben welche Komponenten bei der Verwaltung von Prozessen und Events übernehmen.

Die Prozessverwaltung Da das jABC-Framework lediglich einen Prozess und seine Teilprozesse verwalten kann, war es erforderlich, dieses um eine Prozessverwaltung zu erweitern. Diese durch den `ProcessManager` erbrachten Erweiterungen erlauben das Starten, Pausieren, Forsetzen sowie die Beendigung von Prozessen. Außerdem ermöglicht diese Lösung die externe Nutzung dieser Funktionen durch *SIBs*.

Wie wird ein Prozess gestartet? Da die *jABCProzessPlattform* nur durch Events von außen beeinflusst werden kann, kann das Starten eines Prozesses also nur aus einem anderen Prozess heraus erfolgen. Das heißt, während der Ausführung eines Prozesses wird ein *SIB*ausgeführt, welcher den Start eines neuen Prozesses bei dem `ProcessManager` anfordert. Dieser erzeugt eine neue `Process`-Instanz des gewünschten Prozesstyps (Modell des `ExecutionControllers` des `Process`-Objektes wird mit dem gewünschten Modelltyp initialisiert), initialisiert diesen mit dem übergebenen Kontext, fügt diesen der `ProcessList` hinzu und startet seine Ausführung.
Die einzige Ausnahme stellt der Gesamtprozess der *jABCProzessPlattform* dar. Dieser wird initial bei der Instanziierung des `ProcessManagers` gestartet. Er dient nun dazu, über den Webservice kommende Events zu verarbeiten und dabei neue Prozesse zu starten.

Wie kann ein Prozess pausiert werden? Ein Prozess kann natürlich nur dann pausiert werden, wenn er bereits gestartet wurde. In der Regel fordert der ausführende Prozess selber seine Pausierung in einem *SIB* an. Dies kann sinnvoll sein, wenn der Prozess erst mit seiner Ausführung fortfahren soll, wenn ein bestimmtes Event eingetreten ist. In diesem Falle müsste der Prozess vor seiner Pausierung das entsprechende Event beim `ProcessManager` als *HIGH-Level-Event* abonnieren (siehe Abschnitt 9.3.3: Eventlevel), so dass der `ProcessManager` beim Auftreten dieses Events den entsprechenden pausierten Prozess auch wirklich wieder aufweckt.

Wie und wann wird ein Prozess fortgesetzt? Voraussetzung für die Fortsetzung eines Prozesses ist, dass der entsprechende Prozess gerade pausiert. Im Gegensatz zum Pausieren, wird diese Aktion jedoch nicht aus einem Prozess heraus angefordert, sondern von Komponenten der *jABCProzessPlattform*.
Ein pausierter Prozess wird dann wieder fortgesetzt, wenn ein zuvor von ihm abonniertes *HIGH-Level-Event* durch den `ProcessManager` evaluiert wird oder der Prozess seine in der `TimeoutQueue` festgehaltene maximale Wartezeit erreicht hat.

Muss ein Prozess überhaupt von *außen* beendet werden können? Da Prozesse in *jABC* so lange laufen, bis sie zu einer Exit-Kante gelangen, also das *GraphSIB* abgearbeitet wurde, ist es grundsätzlich nicht notwendig Prozesse stoppen oder beenden zu können. Wenn die Ausführung eines Prozesses beendet ist, bleibt jedoch der Prozess selbst in der Liste der aktiven Prozesse, kann aber aufgrund seines internen Status nicht mehr beeinflusst werden. Deshalb besitzt der `ProcessManager` den `ProcessStatusListener` aller laufenden Prozesse, so dass er, sobald ein Prozess den Status gestoppt erreicht, diesen aus der `ProcessList` und `EventList` entfernen kann.

Das Eventmanagement Neben der Verwaltung von Prozessen ist es die Hauptaufgabe der *jABCProzessPlattform*, von der *IGM-Plattform* über den Webservice verschickte

Events den betreffenden Prozessen zuzuordnen und sie zu verarbeiten.
Somit ergeben sich zwangsläufig folgende Probleme:

- Wie entscheidet man, welcher Prozess welches Event abonniert hat?
- Wie kann dieser Prozess ermittelt werden?
- Inwiefern beeinflusst ein Event die weitere Ausführung eines Prozesses?

Die Lösung für diese Fragestellungen liefert der `EventAboService` in Form der `EventList`. Diese besteht aus zwei Listen, eine Liste der Events, die abonniert wurden, sowie eine Prozessliste, deren Elemente Events abonniert haben. Auf diese Weise kann man für ein auftretendes Event entscheiden, welcher Prozess auf dieses Event hört und für diesen dann die Event-Evaluierung starten; außerdem kann hierdurch die Liste aller abonnierten Events eines bestimmten Prozesses bestimmt werden. Diese beiden Listen werden durch die `EventList` synchron gehalten.

Abonnieren eines Eventtyps Für das Abonnieren eines Eventtyps steht ein *SIB* zu Verfügung, welcher die Service-Methode des `ProcessManager` aufruft. Diese fügt dann der `EventList` das Event-Prozess-Paar hinzu.

Abonnement eines Eventtyps beenden Für das Beenden eines Abonnements steht ebenfalls ein *SIB* zu Verfügung, welcher die Service-Methode des `ProcessManager` aufruft. Diese Methode entfernt anschließend das Event-Prozess-Paar aus der `EventList`.

Evaluieren eines Events Die Evaluierung eines Events gliedert sich in folgende Teilabschnitte:

1. Ermitteln der Liste von Prozessen, die auf das zu evaluierende Event hören
2. Eventverarbeitung für jeden Prozess abhängig vom abonnierten `EventLevel` (siehe nächster Abschnitt)

Weitere Einzelheiten können dem Sequenzdiagramm in Abbildung 33 entnommen werden.

Definition von Event-Level Auf der Implementierungsebene auf Seiten der Prozessverwaltung des *jABC* wird eine dreistufige Priorisierung von Events vorgenommen. Sie dient dazu, des Effekt jedes möglichen auftretenen Events festzulegen, den es auf Prozesse abhängig von deren aktuellen Zuständen ausübt. Bei Prozessen werden die Zustände aktiv und inaktiv definiert. Aktive Prozesse sind aktuell von der Prozessverwaltung ausgeführte Prozesse, während inaktive Prozesse sich in einem schlafenden Zustand befinden, bis sie durch ein von ihnen abonniertes Event wieder aufgeweckt werden. Beispielsweise innerhalb des Prozesses *Brand* (vgl. Abschnitt 9.4) wird nach erfolgreicher Benachrichtigung von Dienstleister und Bewohnern ein Zustand *WAIT* erreicht, in der dem laufende Prozess pausiert und das Event *CloseFireAlarm* abonniert.
Es werden drei Priorisierungsstufen unterschieden: *LOW*, *HIGH* und *CRITICAL*. Während als *LOW* priorisierte Events lediglich in den Kontext der *jABC*-Laufzeitumgebung geschrieben werden und nicht in der Lage sind zu veranlassen, dass inaktive Prozesse durch sie aufgeweckt werden, werden bei einem Event der Priorität HIGH alle inaktiven Prozesse reaktiviert, die dieses Event abonniert haben. Es ist somit erforderlich, dass beispielsweise das o. g. Event *CloseFireAlarm* die Priorität *HIGH* besitzen muss, um den pausierenden Prozess Brand aufwecken zu können. Die höchste Priorisierungsstufe stellt hingegen *CRITICAL* dar. Events diese Priorität erfordern in aktiven und inaktiven Prozessen die Ausführung einer Ausnahmenbehandlung, da durch dieses Ereignis Maßnahmen getroffen

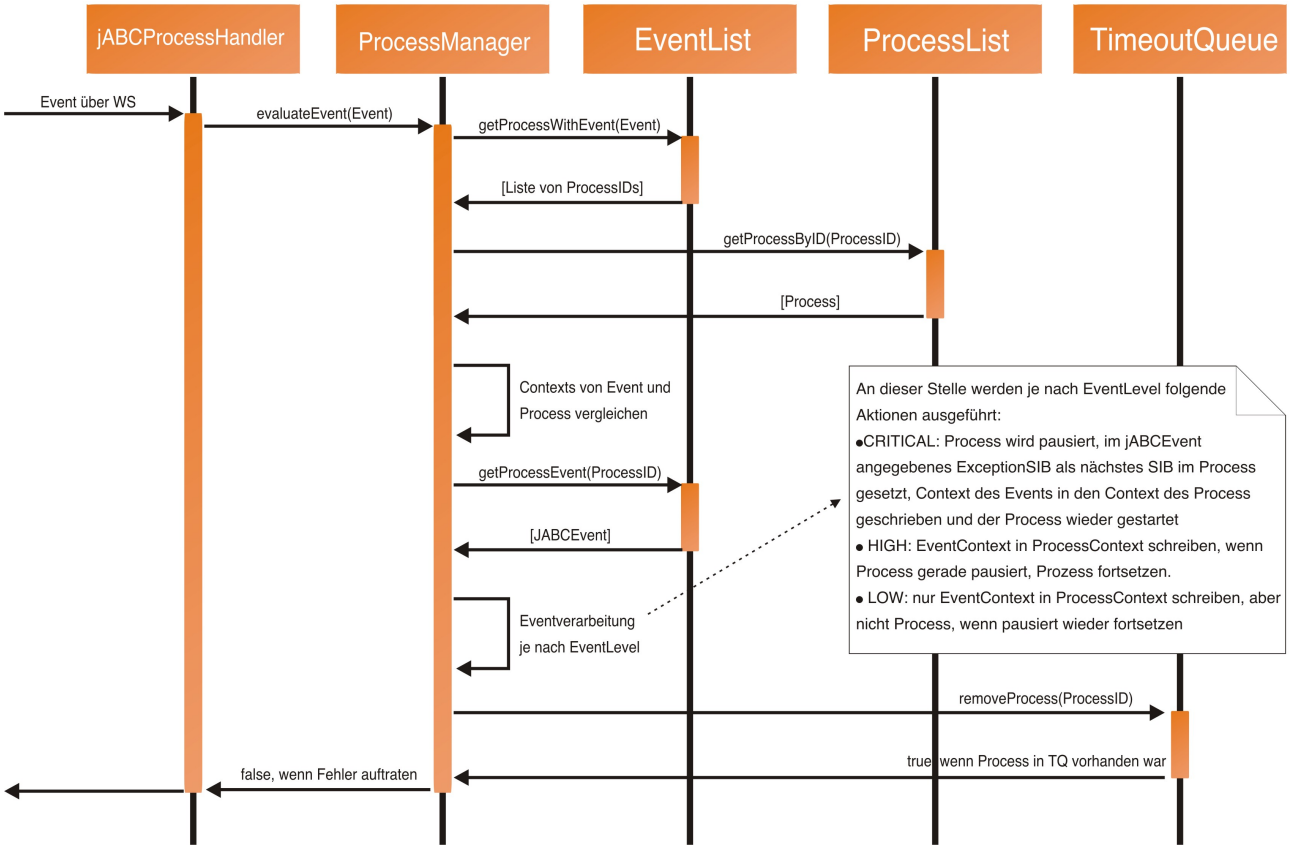


Abbildung 33: Sequenzdiagramm: Evaluierung von Events

werden müssen, die in dem Standard-Prozessverlauf nicht vorgesehen sind. Sie veranlassen somit einen Sprung innerhalb des Prozessverlaufs. Im o. g. Beispiel des Brandprozesses ist das Event *NotifiyGasLeak* als *CRITICAL* einzustufen, da es direkte Auswirkungen auf den Prozess ausübt, die es erfordern zusätzlich zum normalen Prozessverlauf alternative Prozessschritte durchzuführen. Eventuell muss der Dienstleister zusätzlich über die Gasleckagemeldung informiert werden, da bei der Brandbekämpfung austretendes Gas ein erhöhtes Sicherheitsrisiko darstellt.

9.3.4 Realisierung eines Gesamtprozesses

Motivation Die *jABCProzessPlattform* mit den in den vorherigen Abschnitten beschriebenen Komponenten und Funktionen stellt eine Verwaltungskomponente und Ausführungsumgebung für *SmarterWohnen*-Abläufe als *jABC*-Prozesse dar. Die eigentliche Modellierung dieser in Kapitel 8.1 fachlich spezifizierten Abläufe wird im folgendem Kapitel 9.4 beschrieben. Jedes dieser Prozessmodelle realisiert somit einen bestimmten Prozesstyp wie z. B. den Prozess *Brandmeldung*. Es handelt sich hierbei jedoch zunächst nur um statische Prozessbeschreibungen, welche zwar die für die jeweiligen Anwendungsfälle zu realisierende Prozesslogik beschreiben, allerdings keine tatsächlich laufende Prozessinstanzen darstellen.

Die *jABCProzessPlattform* implementiert Mechnismen zur Eventevaluierung, Prozessinstanziierung und -ausführung sowie zu deren Durchführung benötigte Datenstrukturen. Sie beinhaltet jedoch keine Komponente zur Realisierung einer steuernden Koordinierungsschicht, um die fachlich vorgegebene Ausführungssemantik einer eventbasierten Instanziierung von Prozessen bereitzustellen. Dabei ist die Implementierung eines nach

fachlichen Vorgaben definierten Mappings zwischen Event- und Prozesstypen von besonderer Bedeutung, da hierdurch das von Events induzierte reaktive Verhalten festgelegt wird. Es legt somit für jeden relevanten Eventtyp den zu instanziierenden Prozesstyp fest. Wann eine Prozessinstanz eines bestimmten Prozesstyps instanziiert und im Anschluss daran ausgeführt wird, ist daher abhängig von Events, die von der *IGM-Plattform* an die *jABCProzessPlattform* versendet werden.

Mit der eventbasierten Instanziierung eines solchen Prozesses beginnt der Lebenszyklus einer Prozessinstanz, wird abhängig von der Modellierung des Prozesstyps fortgeführt und endet, sobald der Prozess einen Endzustand erreicht. Dabei unterscheiden sich die verschiedenen Instanzen eines Prozesstyps in Hinblick auf die innerhalb des instanziierenden Events definierten Kontextinformationen. Sind hingegen mehrere Events bezüglich des Eventtyps und der enthaltenen Kontextinformationen identisch, so veranlasst jedes dieser Events eine Instanziierung gleichen Prozesstyps mit identischen Kontextinformationen. Eine eindeutige Identifizierung dieser Instanzen ist dabei stets durch die `ProcessId` des ProcessManagers gegeben.

Der Gesamtprozess Um eine solche den oben genannten Anforderungen genügende Komponente zu erhalten, wird nicht die *jABCProzessPlattform* um diese benötigten Funktionen erweitert, sondern die Bereitstellung dieser Funktionen durch einen Prozess realisiert. Denn das Mapping von Event- zu Prozesstypen stellt die Umsetzung einer fachlichen Anforderung an das Gesamtsystem dar und muss somit genau wie die Prozessmodellierung der *SmarterWohnen*-Abläufe die Möglichkeit der dynamischen Veränderung durch den Modellierer bereitstellen. Auf diese Weise wird nun auch in der Prozessausführungsumgebung eine strikte Trennung zwischen fachlicher und technischer Schicht herbeigeführt.

Dieser als Gesamtprozess oder *Big Process* bezeichnete Wurzelprozess wird direkt nach dem Start der *jABCProzessPlattform* instanziiert und ausgeführt. Es existiert dabei im funktionsfähigen Zustand der Plattform immer genau eine laufende Instanz dieses Prozesses. Er besteht aus den drei Phasen Initialisierung, Eventabonnement und Eventverarbeitung, die zum Prozessstart durchlaufen werden. In der Initialisierungsphase werden alle vorhandenen Prozesstypen eingelesen und in der Eventabonnementphase alle relevanten Eventtypen an der *jABCProzessPlattform* abonniert. Auf diese Weise werden alle von der *IGM-Plattform* versendeten Events an den Gesamtprozess weitergeleitet, damit dieser nun in der Eventverarbeitungsphase in Abhängigkeit vom jeweiligen Eventtyp die Instanziierung und Ausführung von Prozessen durchführen kann. Dabei werden die für diese Aufgabe benötigten Funktionen der *jABCProzessPlattform* in Form von speziellen *SIBs* bereitgestellt, sodass beispielsweise Methoden wie die Prozessinstanziierung oder die Eventabonnierung in der Modellsicht des *jABC* zur Verfügung stehen und zur Spezifikation des Gesamtprozesses genutzt werden können. Es folgt nun eine Beschreibung der zwei Hauptphasen Eventabonnement und Eventverarbeitung.

Eventabonnement In Abbildung 34 ist die Eventabonnementphase als *jABC*-Teilprozess dargestellt. Die Abonnierung eines Eventtypen erfolgt dabei jeweils durch ein spezielle *SIB*, welches den `ProcessManager` veranlasst, den Gesamtprozess als Abonnent dieses Eventtyps in die `EventList`-Datenstruktur aufzunehmen. Somit werden alle Events dieses Typs an den Gesamtprozess weitergeleitet. Die Eventabonnementphase wird beim Start des Gesamtprozesses einmalig durchlaufen.<

Abbildung 34: Die Phase *Eventabonnement* des Gesamtprozesses

Eventverarbeitung Die Phase Eventverarbeitung ist die Hauptphase des Gesamtprozesses, in der sich dieser den größten Teil seines Lebenszyklus befindet. In Abbildung 35 ist diese Phase als *jABC*-Modell dargestellt. Zentrale Komponente dieser Phase ist das *SIB WaitingForEvents*, welches einen Wartezustand repräsentiert, in welchem der Gesamtprozess auf die Zustellung von Events wartet. Bei Eintreffen eines Events wird in Abhängigkeit des Eventtyps eine Instanz des im o. g. Mapping festgelegten Prozesstyps erzeugt und gestartet. Die dazu benötigte Funktionalität zur Instanziierung wird durch Prozesstyp-spezifische *SIBs* realisiert, die von der *jABCProzessPlattform* bereitgestellte Schnittstellen zur Durchführung verwenden. Sobald dieser Arbeitsschritt abgeschlossen ist, kehrt der Gesamtprozess wieder in den Wartezustand zurück, um weitere Events zu verarbeiten. Die in diesem Zyklus gestartete Prozessinstanz wird genau wie der Gesamtprozess selbst von der *jABCProzessPlattform* bis zu seiner Terminierung ausgeführt. Die in diesen Prozessen je nach Prozessimplementierung vom zu realisierenden Anwendungsfall abhängige Abonnierung und Verarbeitung prozessspezifischer Events erfolgt dabei unabhängig vom Gesamtprozess und wird in den im *jABC* erstellten Prozessmodellen direkt implementiert. Die Prozessmodellierung der *SmarterWohnen*-Abläufe wird im folgenden Abschnitt 9.4 beschrieben.

9.4 Implementierung der Prozesse im jABC

Erste Schritte der Implementierung Ausgehend von der in Kapitel 8.1 beschriebenen Analysephase der *SmarterWohnen*-Prozesse und der im Rahmen einer Anforderungsanalyse identifizierten *SIBs* (siehe Abschnitt 8.2) wurden daraus konkrete *SIBs* sowie einige *SmarterWohnen*-Prozesse im *jABC* implementiert.

SIB-Entwicklung Bei der Implementierung dieses Prozesses im *jABC* werden zunächst *Proxy-SIBs*[17] eingesetzt; diese verfügen über keinen Programmcode, sondern dienen lediglich im Rahmen einer ersten Implementierung als Platzhalter für im Anschluss daran zu entwickelnde *SIBs*. Außerdem soll in diesem Prozess im Unterschied zur ursprünglichen Modellierung durch das Fraunhofer ISST die Benachrichtigung von Bewohner und Dienstleister nebenläufig erfolgen. Daher wird der Prozessfluss durch ein *Fork-SIB* in

[17] vgl. Abschnitt 7

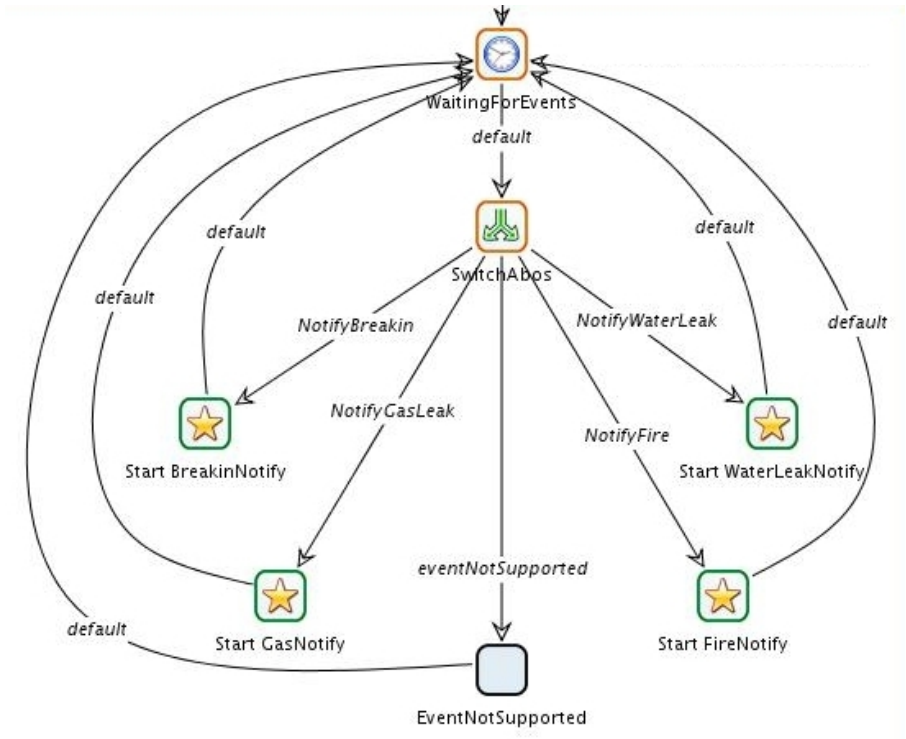


Abbildung 35: Die Phase *Eventverarbeitung* des Gesamtprozesses

zwei parallele Prozessketten aufgespalten und nach ausgeführter Benachrichtigung wieder durch ein *Join-SIB* zusammengeführt. Diese Modifikation dient einem nahezu gleichzeitigen Informieren von Dienstleister und Bewohner.

Kontextzugriff Jedes ausgeführte *SIB* bekommt in der `Trace`-Methode den aktuellen Kontext übergeben. Der Zugriff darauf erfolgt ähnlich wie bei einer `HashMap`, über `get` und `put`.

9.4.1 Implementierte SIBs

AboEvent 

Mit dem SIB *AboEvent* können Eventtypen abonniert werden, die für den Prozess relevant sind. Der Prozess reagiert solange auf diesen Eventtypen, bis das Abonnement mittels *DeAboEvent*-SIB beendet wird.

Parameter

- `eventLevel`: Legt das Level des Events fest.
- `eventType`: Identifikator des zu abonnierenden Events.
- `exceptionSIB`: Name des SIBs, an dem die Ausführung bei einem kritischen Event fortgesetzt wird.

CreateMessage 

Erstellt eine Benachrichtigungsnachricht und legt diese im Kontext ab.

Parameter

- `messageBox`: Art der Meldung, die erstellt werden soll.
- `notificationObject`: Name des Kontextschlüssels, unter dem das zu benachrichtigende Objekt abgelegt ist.
- `notifyBox`: Auswahl, ob Bewohner oder Dienstleister benachrichtigt werden soll.

DeAboEvent 

Mit dem SIB *DeAboEvent* kann das Abonnement von Eventtypen gekündigt werden.

Parameter

- `eventType`: Identifikator des Events, dessen Abonnement beendet werden soll.

Init 

Das *Init*-SIB initialisiert den Prozess und legt dessen Kontext an.

Parameter

- `initialContext`: `HashMap` mit initialen Parametern, die dem Prozess übergeben werden sollen.
- `processType`: Der Prozesstyp, zum Beispiel *Brandmeldung*.

Iterator 

Mit dem *Iterator*-SIB kann über eine im Kontext liegende `HashMap` iteriert werden. Dazu legt es in jedem Iterationsschritt jeweils ein Objekt aus der `HashMap` in den Kontext.

Parameter

- `iterationMap`: Name (Kontextschlüssel) der `HashMap`, über die iteriert werden soll.
- `iterationObject`: Name, unter dem die einzelnen Objekte abgelegt werden sollen.

Kanten

- `finish`: Wird gewählt, wenn die `HashMap` keine weiteren Elemente mehr enthält.
- `iterate`: Wird während der Iteration gewählt.

NotificationControl 

Das *NotificationControl*-SIB entscheidet an Hand des von der IGM-Plattform übergebenen `ReturnCode`, ob die gesendete Benachrichtigung erfolgreich übertragen wurde. Die Bedingungen, wann der Vorgang als erfolgreich gilt, legt die IGM-Plattform fest. Hier wird nur der boolesche `Returncode` ausgewertet.

Kanten

- `message`: Wird gewählt, wenn eine weitere Benachrichtigung erforderlich ist.
- `success`: Wird gewählt, wenn die Benachrichtigung erfolgreich war.

RetrieveData 

Mit dem *RetrieveData*-SIB kann ein DTO mittels `retrieveData` übertragen werden, um Daten anzufordern. Die empfangenen Daten werden in den Kontext geschrieben.

Parameter

- `dtoSelect`: Zu übertragendes DTO.

SendMessage 

Überträgt eine erstellte Benachrichtigung mittels `sendMessage` an die IGM-Plattform.

SendEvent 

Das *SendEvent*-SIB erstellt ein Event und schickt es mittels `sendEventToISST` an die IGM-Plattform.

Parameter

- `eventname`: Name des zu erstellenden Events.
- `keys`: `HashMap` mit Kontext-Schlüsseln, die ausgelesen und dem Event hinzugefügt werden.

WaitingForEvents 

Das *WaitingForEvents*-SIB wartet auf abonnierte Events und setzt die Ausführung erst dann fort.

Parameter

- `timeOut`: Maximale Wartezeit, nach der die Ausführung auch ohne Event fortgesetzt wird.

Generalisierung Um komplexe Prozesse übersichtlicher zu gestalten, können Teile des Prozesses in eigene Graphen ausgelagert und als *GraphSIB* eingebunden werden, was besonders bei sich wiederholenden Aufgaben von Vorteil ist. Außerdem können abgeschlossene Einheiten zusammengefasst und dem Anwendungsentwickler als Blackbox zur Verfügung gestellt werden. Dieser muss sich dann nicht mehr um die technischen Details des Vorgangs kümmern. Wir haben dies exemplarisch an Hand der Bewohnerbenachrichtigung implementiert.

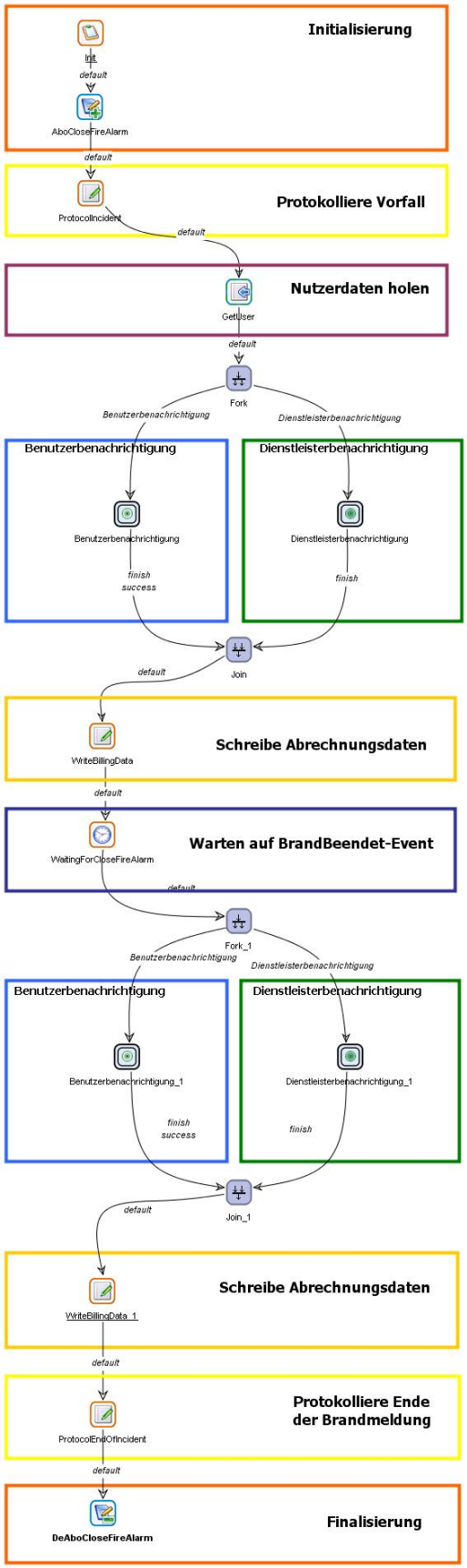


Abbildung 36: Brandprozess

9.4.2 Implementierte Prozesse

In diesem Abschnitt wird die Implementation des *Brandprozesses* aus dem Abschnitt 9.4.1 exemplarisch dargestellt. Die Prozesse *Wasserleckage, Gasleckage* und *Einbruch* sind ähnlich aufgebaut. Die Prozesse *Serviceruf* und *kritischeVitalwerte* unterscheiden sich insofern von den übrigen Prozessen, als dass bei ihrer Ausführung lediglich der entsprechende Dienstleister benachrichtigt wird und nicht der Bewohner, da dieser den Alarm (in)direkt ausgelöst hat.

Brandprozess Zunächst wird der Prozess durch das entsprechende SIB initialisiert. Danach abonniert er den *CloseFireAlarm*-Event für sich. d.h. der Prozess wird nun informiert, wenn eine Beendigung des Brandes für die Wohnung, für welche er zuständig ist, ausgelöst wird. Nach dem Protokollieren des Vorfalls werden die Bewohnerdaten ermittelt, die zum Benachrichtigen des Bewohners und des Dienstleisters erforderlich sind. Hierzu wird im *RetrieveData*-SIB die DTO `GetUser` ausgewählt.
Die folgenden Bearbeitungsschritte, Benachrichtigung des Bewohners und Dienstleisters über den ausgebrochenen Brand, werden nun parallel durchgeführt. Eine detaillierte Beschreibung dieser Graph-SIBs ist weiter oben zu finden. Nach den Benachrichtigungen werden die entsprechenden Abrechnungsdaten durch das *WriteBillingData*-SIB archiviert. Ist dieser Teil des Prozesses abgeschlossen, wird der Prozess in einen Wartezustand versetzt. Die Ausführung wird erst fortgesetzt, wenn der Prozess das oben bereits abonnierte *CloseFireAlarm*-Event erreicht.
Trifft das Event ein, so werden die Bewohner und Dienstleister analog zur obigen Beschreibung über das Ende des Brandes informiert. Erneut werden Abrechnungsdaten und das Ende des Brandes protokolliert. Danach wird das Abonnement des *CloseFireAlarm*-Events gekündigt und der Prozess terminiert.

Ausnahmebehandlung Da die Kombination aus einer Gasleckage und einem Brand verheerende Folgen haben kann, ist es wünschenswert, auf eine solche Kombination von Events zu reagieren. Diese Funktion muss sowohl bei *Brand-* als auch beim *Wasserleckage*-Prozess modelliert werden. Hierzu wird zusätzlich das entsprechende Alarm-Event abonniert (der *Brand*-Prozess abonniert also *NotifyGasLeak*) und geprüft, ob dieses Event bereits aufgetreten ist und ein entsprechender Prozess gestartet wurde. Ist dies der Fall, so wird eine Ausnahmebehandlung, welche zuvor in dem Graph-SIB modelliert wurde (z.B. die Benachrichtigung aller Bewohner dieses Hauses), durchgeführt, anderenfalls wird der normale Alarmprozess durchlaufen.

Farbliche Abgrenzung Um funktional unterschiedliche Bereiche eines Prozesses voneinander abzugrenzen, werden Rahmen benutzt. Dabei wurde sich auf folgende farbliche Konvention geeinigt:

Grün/Blau Versenden von Nachrichten über die IGM-Plattform (inkl. Abfrage der Kommunikationskanäle, Erstellen einer Nachricht und eigentliches Versenden)

Dunkelblau Auf ein Event warten

Lila Daten von IGM erfragen

Gelb Protokollieren

Orange Initialisierung und Terminierung eines Prozesses

9.5 Monitoringmechanismen

Durch den kompositionalen Aufbau des Gesamtsystems ergab sich die Notwendigkeit, ein Tool zu entwickeln, mit dessen Hilfe die Abläufe und der Zustand des Systems überwacht

werden können.
Hierzu wurde ein Monitoringtool implementiert, dass zum einen die Zustände der einzelnen Systeme anhand ihrer Log-Einträge zusammenführt und zum anderen ihre Kommunikation anhand von Transaktionen aufzeigt.
Eine detailliertere Analyse der beiden Plattformen kann durch ihre Log-Dateien vorgenommen werden.

9.5.1 Aufbau des graphischen User-Interfaces

Bei dem Monitoringtool zur Überwachung und Überprüfung der *jABCProzessPlattform* und der *IGM-Plattform*, handelt es sich um eine Webapplikation, die auf demselben Server läuft, den auch die jABC-Plattform nutzt. Zur Erstellung wurde das Google-Web-Tool-Kit [22] verwendet. Die Monitoring-Anwendung wurde als Java-Code geschrieben. Das GWT transformiert diesen dann in Javascript. Ergebnis ist eine Web-Applikation, die in jedem Browser dargestellt werden kann, aber trotzdem das Gefühl einer Desktop-Applikation vermittelt. Auf diese Weise kann der Status der beiden Plattformen von jedem Ort aus in Echtzeit überprüft werden.
Das Monitoringtool lässt sich unter `http://pulicis.cs.uni-dortmund.de:8080/JABCPlatform/Monitor.html` aufrufen. Es besteht aus den folgenden drei Ansichten:

Logs Das *Logs-Panel* stellt die Log-Einträge der beiden Plattformen nebeneinander dar; auf der linken Seite die Einträge der *jABCProzessplattform*, auf der rechten Seite die Einträge der *IGM-Plattform*.
Jeder Eintrag besteht aus dem Zeitpunkt, an dem dieser von der jeweiligen Plattform erzeugt wurde, dem *Debug-Level* und der Nachricht.

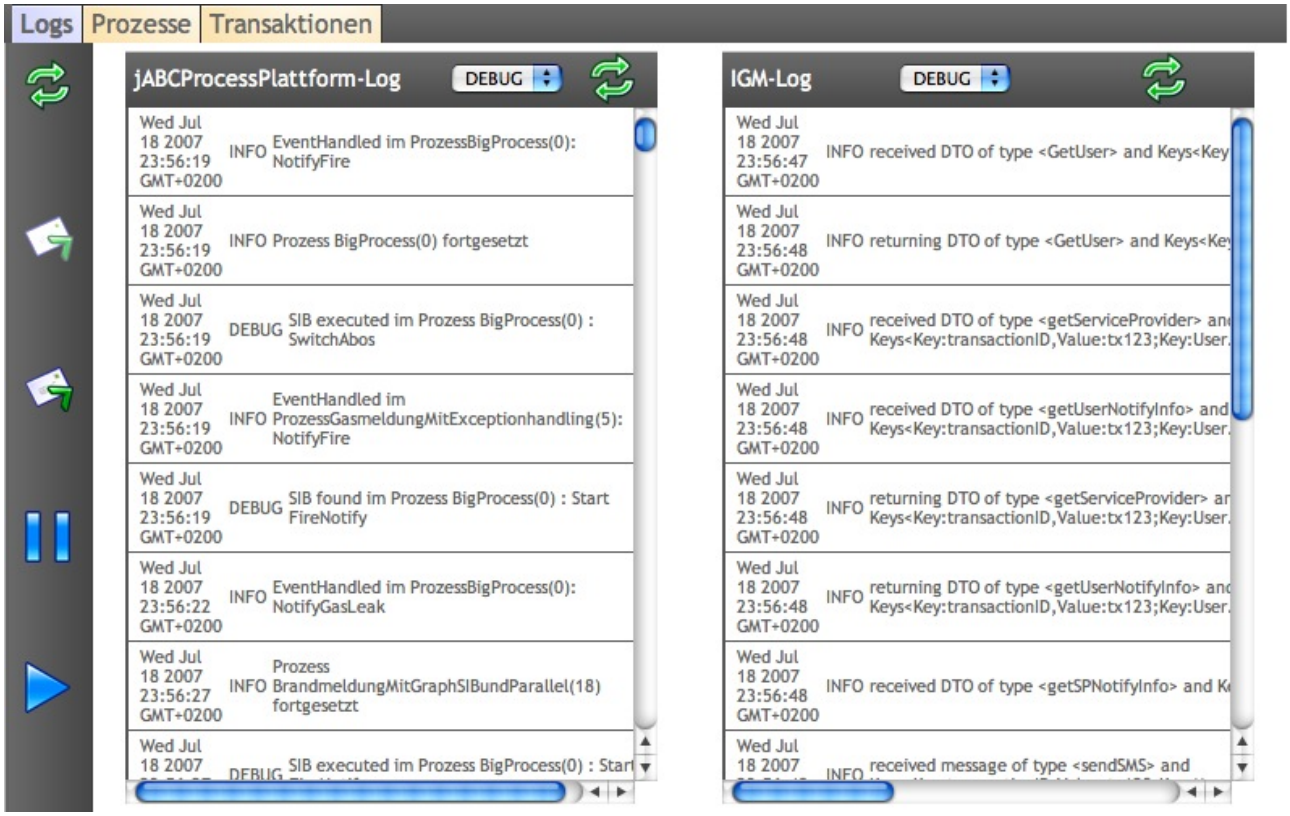


Abbildung 37: Das *Logs-Panel*

Das *Debug-Level* kann, anlehnend an die Log4J-Notation [12], die Werte *TRACE*, *DEBUG*, *INFO*, *WARN* oder *ERROR* annehmen. Die Bedeutung der einzelnen Level kann somit ebenfalls der *Log4J*-Dokumentation entnommen werden.
Die Einträge der beiden Logs können über das jeweilige Drop-Down-Menü im Kopf der Liste nach dem *Debug-Level* gefiltert werden, wobei die fachlich übergeordneten *Debug-Level* mit eingeschlossen werden. Auf diese Weise können die Log-Einträge auf die *wichtigen* Meldungen beschränkt werden.
Ein Beispiel:
Wenn nach dem *Debug-Level DEBUG* gefiltert wird, werden alle Log-Einträge mit den *Debug-Leveln DEBUG*, *INFO*, *WARN* und *ERROR*, aber nicht *TRACE*, angezeigt.
Die *Logs* der Plattformen können entweder getrennt voneinander, manuell über den *Aktualisierungsbutton* im Kopf der jeweiligen Liste, oder automatisch (alle 2 Sekunden) über den *Auto-Aktualisierungsbutton* in der Toolbar aktualisiert werden. So kann nahezu in Echtzeit das Verhalten der beiden Plattformen verfolgt werden.

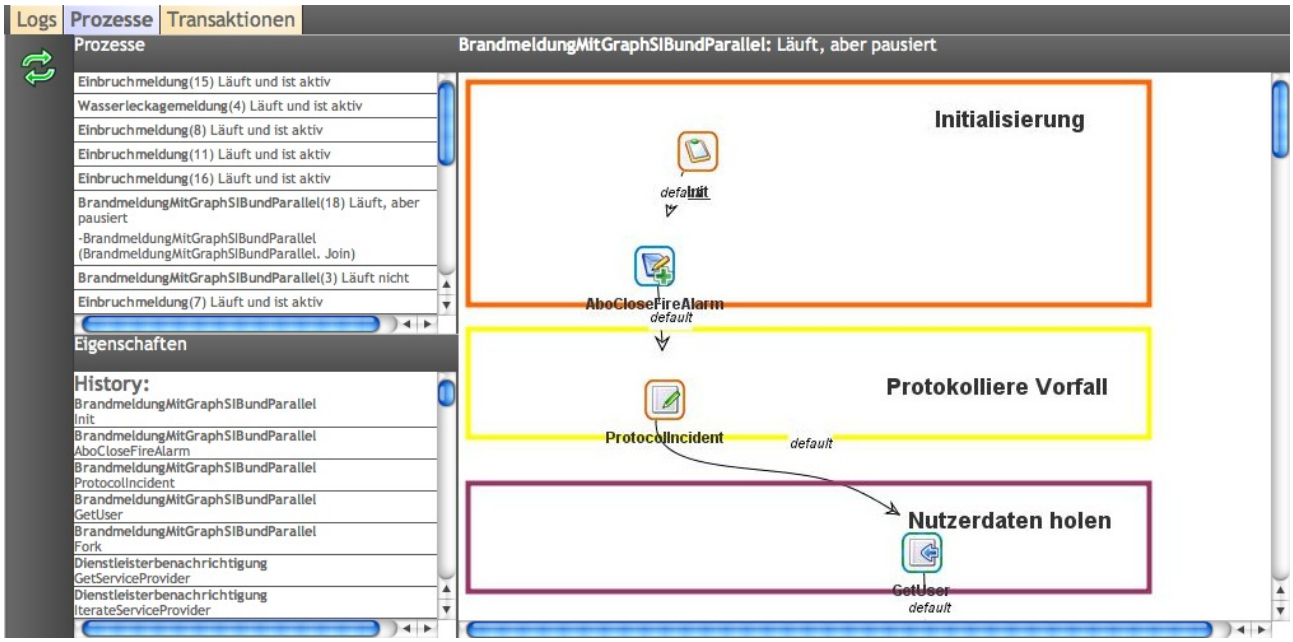


Abbildung 38: Das *Prozess-Panel*

Prozesse Das *Prozess-Panel* stellt die auf der *jABCProzessPlattform* laufenden Prozesse dar. Jeder Prozess, der auf der *jABCProzessPlattform* ausgeführt wird oder wurde, erscheint in der *Prozessliste*. Dieser wird jeweils durch seinen Namen (bzw. Namen des Prozessgraphen), seiner *ProzessID*, seinem aktuellen Ausführungszustand und seinen zugehörigen Threads repräsentiert.

Klickt man in der Prozessliste auf den Namen eines Prozesses bzw. Threads, wird in den *Eigenschaften* seine Ausführungshistorie und im rechten Teil des Prozess-Panels eine Abbildung des Prozess-Graphen des jeweiligen Prozesses bzw. Threads angezeigt.

Die Ausführungshistorie listet alle *SIBs* auf, die bisher in einem Prozess bzw. Thread abgearbeitet wurden.
Die Prozessliste kann über den Aktualisierungsbutton in der Toolbar aktualisiert werden.

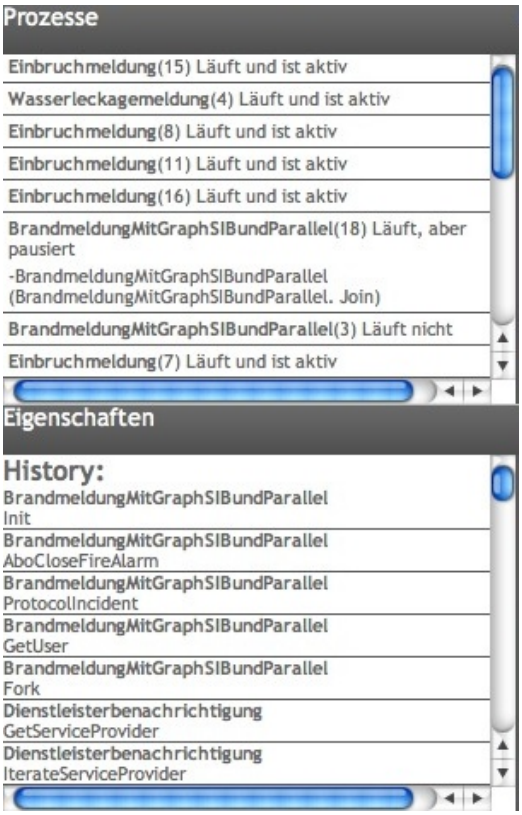


Abbildung 39: *Prozessliste* und *Eigenschaften* eines Prozesses

Transaktionen Das *Transaktions-Panel* stellt alle Transaktionen dar, die zwischen den beiden Plattformen stattgefunden haben.

Eine Transaktion beginnt mit einem von der *IGM-Plattform* gesendeten Event und endet mit dem Beenden des Prozesses, der aufgrund dieses Events vom *Gesamtprozess (BigProcess)* gestartet wurde. Somit beinhaltet eine Transaktion sämtliche Kommunikation, die durch diesen Prozess und seinen Unterprozessen mit der *IGM-Plattform* betrieben wurde.
Die verschiedenen Transaktionen werden in der *Transaktionsliste* aufgelistet. Eine Transaktion wird darin durch den initialen *sendEvent*-Aufruf repräsentiert.
Durch das Anklicken eines Transaktionseintrages wird in der rechten Hälfte des *Transaktions-Panels* die komplette Transaktion aufgeführt.
Um die Kommunikation zwischen den beiden Plattformen zu verdeutlichen, sind alle eingehenden Anfragen bzw. Antworten der *jABCProzessPlattform* auf der linken und alle eingehenden Anfragen bzw. Antworten der *IGM-Plattform* auf der rechten Seite aufgelistet. Die Einträge sind zeitlich sortiert. Diese Darstellung ähnelt somit einem Sequenzdiagramm.
Eine farbliche Box symbolisiert eine Anfrage bzw. eine Antwort. Die Beschriftung besteht aus der `TransaktionsID`, dem Namen der Webservice-Methode, dem Namen des *DTOs*, dem Schlüsselkontext (Keys) und dem Inhalt (Data) des *DTOs*. Antworten unterscheiden sich von Anfragen durch den Eintrag *Response* und dem in der Regel gefüllten Inhalt.

Mit Hilfe der *Transaktions-Ansicht* kann festgestellt werden, ob das jeweilige System auf eine Anfrage korrekt antwortet und die richtigen Daten zurückgegeben werden. Weiterhin kann beobachtet werden, in welcher Reihenfolge Anfragen beantwortet werden. So kann man gerade bei Prozessen mit Nebenläufigkeit (Teilprozesse in mehreren Threads)

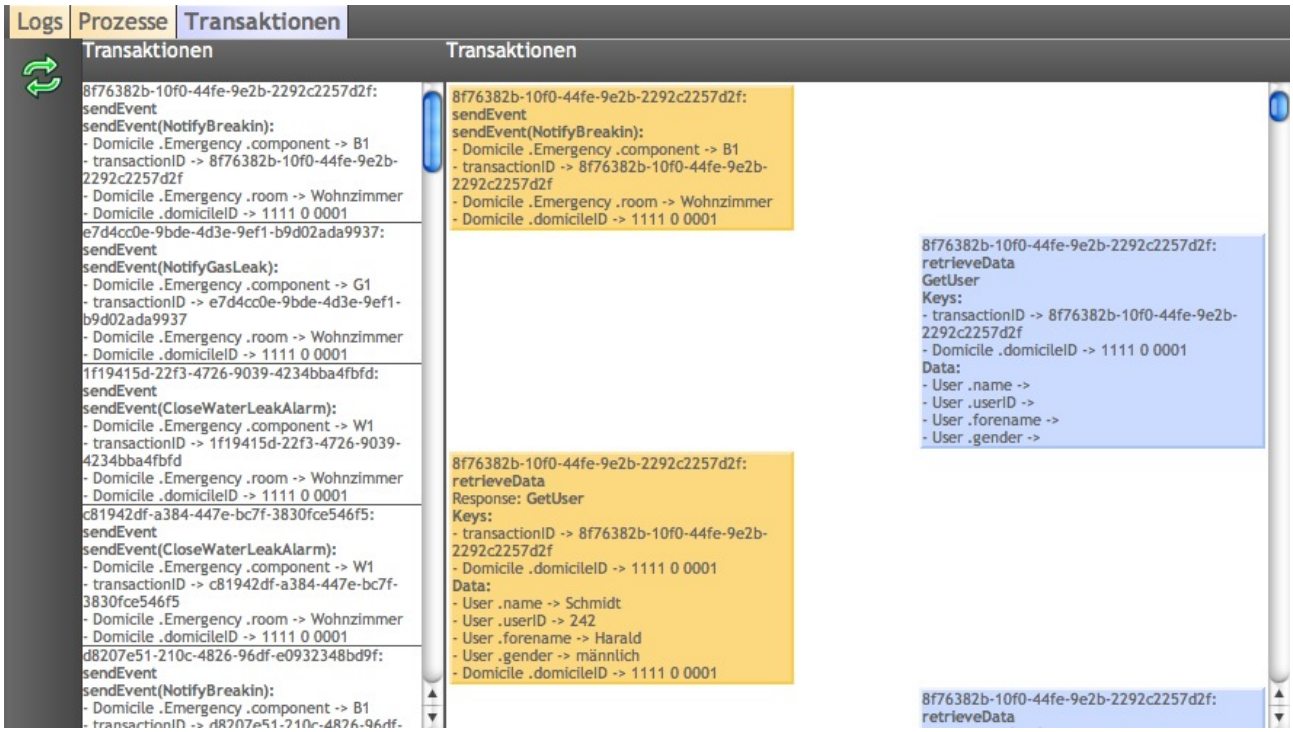


Abbildung 40: Das *Transaktions-Panel*

feststellen, dass Antworten nicht unmittelbar nach und in der selben Reihenfolge ihrer zugehörigen Anfragen erfolgen müssen.

9.5.2 BPMLoggingService

Um das Monitoring des *BPM-Subsystems* über das graphische User-Interface zu ermöglichen, wurde ein weiterer WebService entwickelt, über den Log-Einträge des *BPM-Subsystems* angefragt werden. Um diese Log-Einträge bereitzustellen, wurde die Implementierung des *BPMProcessHandlers* so erweitert, dass mittels *log4net*[13] Informationen zu allen eingehenden und ausgehenden *Events* sowie *DataTransferObjects* in eine Datenbank geschrieben werden. Drei unterschiedliche Logger wurden dafür definiert um eine semantische Trennung zu erhalten. Der Logger *Event* wird beim Eintreten von Events genutzt (Methode `sendEvent`). Der Logger *Data* wird beim Empfang von *DataTransferObjects* bei den Methoden `sendData` und `retrieveData` genutzt. Beim Empfang von *DataTransferObjects* über die Methode `sendMessage` kommt der Logger *Action* zum Einsatz. Die Konfiguration dieser Logger wird in den Kapiteln 7 und 10 des Entwickler-Handbuchs zum *BPM-Subsystem*[43] näher beschrieben. Alle Log-Einträge können aus der Datenbank durch den *BPMLoggingService* ausgelesen werden. Der *BPMLoggingService* stellt zu diesem Zweck unterschiedliche Methoden zur Verfügung. Über diese ist es möglich, Log-Einträge anhand ihrer ID oder des Datums sowie anhand des Log-Levels zu beziehen. Die genaue Beschreibung der Schnittstelle findet sich in der API-Dokumentation zum *BPM-Subsystem*[42] unter `De.UniDo.Poets.Isst.IGM.BpmSubSystem.Common.Contracts.Services.IBPMLoggingService` und deren Implementierung unter `De.UniDo.Poets.Isst.IGM.BpmSubSystem.BPMLoggingService.Services.BPMLoggingService`. Die wichtigsten Funktionen, in dem Sinne, dass sie momentan durch das graphische User-Interface genutzt werden, sind derzeit `LogEntry:getLastEntry()`, über die der zeitlich letzte Log-Eintrag ermittelt wird, sowie `LogEntryList:getEntrysById(System.Int32 from,System.Int32 to)`, über die der-

Abbildung 41: Anfrage und Antwort der Methode *retrieveData* mit dem *DTO GetUser*

zeit alle weiteren Einträge ermittelt werden.

9.5.3 Logfiles

jABC Die *jABCProzessPlattform* schreibt neben den Log-Ausgaben in die Log-Panels des Monitoring-Tools, diese auch mittels *log4j* in detaillierterer Form in eine Log-Datei. Diese Datei wird in dem Ordner erstellt, aus dem der Webserver gestartet wird. Mehr Informationen kann man dem Entwicklerhandbuch der *jABCProzessPlattform* entnehmen.

IGM Neben den Log-Ausgaben des *BPMLoggingServices* schreiben alle Komponenten des *BPM-Subsystems* detaillierte Log-Ausgaben in Log-Dateien. Zu diesem Zweck wurden die Standard-Mechanismen von *log4net*[13] (Version 1.2) genutzt. Damit steht auch die gesamte Flexibilität dieses Frameworks zur Verfügung. Detaillierte Informationen zur Konfiguration und zum Funktionsumfang finden sich auf der Website des Frameworks[13] sowie im Entwickler-Handbuch zum *BPM-Subsystem* in den Kapiteln 7 und 10. In der Standard-Konfiguration findet sich für jede Komponente eine eigene Log-Datei im Verzeichnis *C:\IGM-Platform\BPMSubSystem\log* sowie eine Log-Datei in der alle Einträge aller Komponenten gesammelt werden.

9.6 Sensoranbindung

9.6.1 Sensoren / IPSwitches

Sensoren sind für das *SmarterWohnen* Projekt von essenzieller Bedeutung. Ohne die entsprechenden Meldungen von Fensterkontakten, Rachmeldern, Temperaturfühlern und Pulsmessgeräten würde die Plattform zu einem reinen Informationsanbieter verkommen. Die in den Wohnungen verbauten Sensoren sind von unterschiedlichen Anbietern mal per Funk mal per Kabel vernetzt, bieten einen unterschiedlichen Funktionsumfang und lassen sich trotzdem für die gleiche Plattform benutzen. In Abbildung 16 sind einige zu sehen. Möglich wird dies durch ein so genanntes Gateway, das in den jeweiligen Wohnungen aufgestellt ist. Das Gateway empfängt die Sensorwerte, verknüpft sie mit weiteren Informationen und folgert so zum Beispiel, dass ein sich öffnendes Fenster während alle Bewohner außer Haus sind wahrscheinlich ein Einbruch bedeutet. An die *IGM-Plattform*

wird so nicht ein Sensorwert übermittelt sondern eine aufbereitete Information. Dieser Aufbau hat natürlich Vor- und Nachteile. Die *IGM-Plattform* abstrahiert so von dem eigentlichen Sensor und lässt außer Acht wie die Meldung zustande kommt. So hat man die freie Wahl beim Aufbau des Sensornetzes solange das Gateway die Daten entsprechend aufarbeitet. Als Nachteil wäre hier das Gateway selber zu nennen was durch einen Ausfall eine ganze Wohnung von der *IGM-Plattform* abschneidet. Außerdem gibt es so ein weiteres System, das durch Software- und Hardwarefehler, ja vielleicht sogar durch einen gezielten Angriff, das System im allgemeinen kompromittieren könnte.
Die Anbindung von *jABC* an die *IGM-Plattform* ist gänzlich von den Sensoren abgekoppelt, hier interessieren höchstens die schon aufbereiteten Informationen des Gateways. Trotzdem kam Anfang April in einer PG-Sitzung zum ersten mal die Idee auf, bei einer Präsentation doch nach Möglichkeit reale Sensoren zu verwenden. Das ISST stellt für dieses Vorhaben eine ganze Sammlung an Sensoren auf Basis des *IPSwitch* der Firma SMS-Guard zur Verfügung. *IPSwitche* gibt es in unterschiedlichen Ausführungen, für die Anbindung der Sensoren wurde eine Variante mit separater Anschlussbox gewählt (Die Anschlussbox ist in Abbildung 44 zu sehen). Diese Box hat Eingänge für drei Binärkontakte, ein Temperaturfühler sowie einen 0-10V Eingang. Im Rahmen der Projektgruppe werden nur die drei Binärkontakte verwendet.

9.6.2 Parser

Die Sensordaten eines *IPSwitch* für das Projekt *SmarterWohnen* zu nutzen gestalltete sich am Anfang schwierig. Der *IPSwitch* ist dafür konzipiert, in Abhängigkeit von bestimmten Sensorwerten selbständig eine Reaktion auszulösen und hierfür auch mit anderen *IPSwitchen* im Netzwerk zu kommunizieren. Dies ist praktisch, wenn man z.B. einen Ventilator einschalten möchte, sobald die Temperatur 40°C überschreitet. Es ist aber unpraktisch, wenn man nur die Sensorwerte haben möchte.
Um nur an die Sensorwerte zu kommen bieten sich beim *IPSwitch* zwei mögliche Ansätze. Der Erste ist die Sensorausgabe als HTML-Seite über den integrierten Webserver (siehe Abbildung 42). Auf einer einfach aufgebauten Webseite werden nur durch Text die jeweiligen Sensorzustände sowie einige Konfigurationsparameter aufgeführt. Da fast jede aktuelle Programmiersprache Routinen bereithält, um eine Webseite zum Beispiel als String zu laden und dann in diesem String nach einer Zeichenfolge suchen zu lassen bekäme man mit den Codezeilen im Beispiel 7 und der Webseite aus Abbildung 42 den Status des ersten Sensoreingangs des *IPSwitch*.

Listing 7: Codebeispiel zum Ermitteln eines Sensorzustandes über den Webserver

```
bool sensor_1;
WebClient client = new WebClient ();

string reply = client.DownloadString (address);
sensor_1 = (reply.Substring(reply.IndexOf("iC1=")+4, 2) == "On");
```

Dieser Ansatz produziert mehr Probleme als er löst. Wenn man die Abfrageintervalle zu niedrig setzt, erzeugt man zu viel Netzwerklast und der *IPSwitch* kommt den Anfragen nicht mehr hinterher. Wenn man die Abfrageintervalle zu hoch setzt läuft man Gefahr, dass bei einem sich schnell wechselnden Sensor, zum Beispiel einem Bewegungsmelder, möglicherweise eine Alarmierung verpasst wird. Außerdem widerspricht es dem normalen Verständnis von einem Sensor, man möchte eigentlich gemeldet bekommen, dass ein Ereignis eigetreten ist und möchte nicht explizit nach diesem Ereignis fragen müssen.
Der zweite Ansatz zur Lösung des Problems verbirgt sich in den Einstellungen des *IPSwitch*. Wie auf Abbildung 43 zu sehen ist, kann man einen *IPSwitch* in einen so genannten „PC data logger“ Modus versetzen. In diesem Modus werden alle Sensorwerte

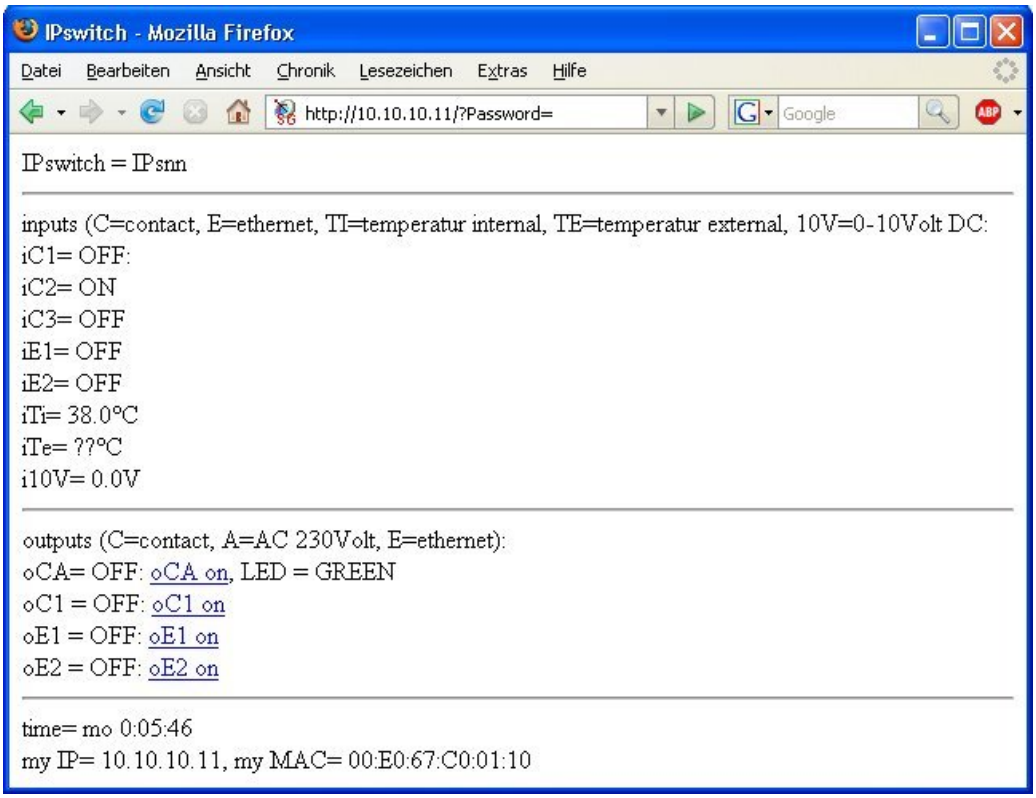


Abbildung 42: Webseite eines *IPSwitch*

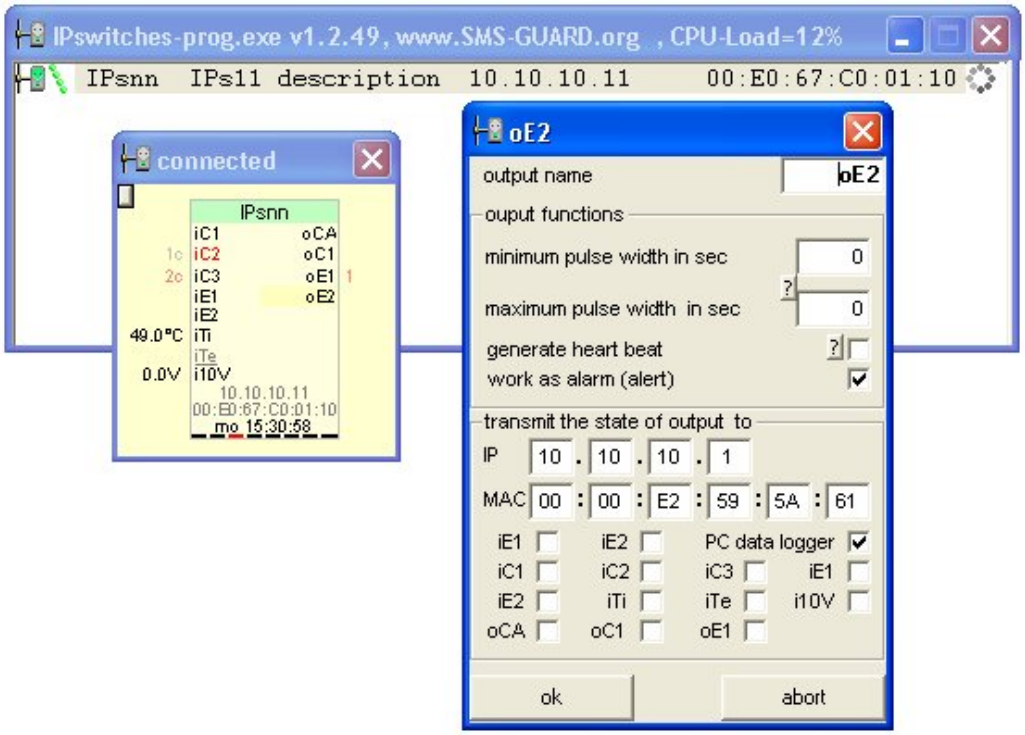


Abbildung 43: Konfiguration eines *IPSwitch*

an eine mitgelieferte Serversoftware verschickt, die diese dann auf dem Bildschirm ausgibt. Hier wurde durch die Projektgruppe die mitgelieferte Serversoftware durch eine eigenentwickelte Komponente ersetzt. Der Aufbau des verschickten Datenpacketes erschloss sich nicht von selbst, sodass an diesem Punkt nur eine Zusammenarbeit mit dem Hersteller des *IPSwitch* weiterhelfen konnte. Laut Auskunft der Firma *SMS-Guard* wird der String, der per UDP an den Port 60096 versendet wird, mit dem Befehl `sprintf(sudpE2, "%2x%1x%5s%5s%4s%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%2x%1x%2x%2x%2x", inp, oupe, iTiS, iTeS, i10VS, iC2z, iC3z, iC1t, iC2t, i C3t, iE1t, iE2t, oCAt, oC1t, oE1t, oE2t, iC1s, iE1s, iE2s, oCAs, oC1s, oE1s, oE2s, ee_eiC1[0], div, tsi, ts, dd, hh, mm, ss)` erzeugt, wobei hier nur die mit `inp` bezeichnete Variable von Interesse ist. `Inp` ist eine Byte-Repräsentierung der drei binären Eingänge des *IPSwitch*, wobei das Bit 0 den ersten, Bit 1 den zweiten und Bit 2 den dritten Eingang repräsentiert. `inp = 5` würde bedeuten, dass die Eingänge eins und drei `True` und Eingang zwei `False` melden.

Listing 8: IPSwitch UDP Datenpaket

```
0000   00 00 e2 59 5a 61 00 e0 67 c0 01 10 08 00 45 00   ...YZa..g.....E.
0010   00 60 2f bc 00 00 ff 11 63 b1 0a 0a 0a 0b 0a 0a   .`/.....c.......
0020   0a 01 ea c0 ea c0 00 4c eb f4 20 35 34 20 34 37   .......L.. 54 47
0030   2e 30 20 20 20 3f 3f 20 30 2e 30 35 61 20 66 33   .0    ?? 0.05a f3
0040   32 20 30 33 32 34 66 34 66 34 66 34 66 33 30 33   2 0324f4f4f4f303
0050   32 20 30 20 30 20 30 20 30 20 30 66 66 66 66 20   2 0 0 0 0 0ffff
0060   30 34 31 66 66 20 31 30 20 31 31 33 20 64         041ff 10 113 d
```

In dem Datenpaket, dargestellt in Listing 8 mit dem Programm *Wireshark*, wurde die Variable `inp` zur einfacheren Erkennung unterstrichen und rot eingefärbt. Da durch den Befehl `sprintf` das `Byte` auf zwei Zeichen vergrößert und in einen `String` umgewandelt wurde, wird in dem Beispiel anstelle einer `5` das Paar „`20 35`“ versandt. Die `20` steht hierbei laut *ASCII*-Tabelle für ein Leerzeichen, die `35` für eine 5.

Listing 9: Empfangen und verarbeiten des UDP-Paketes

```
// Empfange Daten vom Sensor
byte[] data = server.Receive(ref endpoint);
// Parst das Byte Array in ein IPSdata
IPSData tempdata = Parser.Parse2(Encoding.UTF8.GetString(data));
// Prüft ob sich die drei boolschen Eingänge geändert haben
tempsensoren.CheckSensor(endpoint.Address.ToString() + "_1", ((
    tempdata.inp & 1) != 0));
tempsensoren.CheckSensor(endpoint.Address.ToString() + "_2", ((
    tempdata.inp & 2) != 0));
tempsensoren.CheckSensor(endpoint.Address.ToString() + "_3", ((
    tempdata.inp & 4) != 0));
```

Die entsprechenden Codezeilen zum Empfangen und Verarbeiten des Paketes sind in Listing 9 zu sehen. `tempdata.inp` ist die ursprüngliche `Byte`-Repräsentation der binären Eingänge und wird mit einem bitweisen und-Vergleich den jeweiligen Sensoren zugeordnet. Mit diesem Wissen wurde das Programm *POETS Sensoren* (Abbildung 44) entwickelt. Es empfängt die Pakete der entsprechend konfigurierten *IPSwitche* im Netzwerk, bestimmt den aktuellen Status der angeschlossenen Sensoren und überprüft, ob der gemeldete Status über den Sensor-Service an die IGM-Plattform gemeldet werden muss. *POETS Sensoren* ermöglicht mehrere *IPSwitche* gleichzeitig zu verwenden und für jeden Sensor einzeln unterschiedliche Konfigurationen einzustellen. So kann man entscheiden, ob Zustandswechsel (`True` ⇒ `False` bzw. `False` ⇒ `True`) und das Beibehalten eines Zustandes (`True` ⇒ `True` bzw. `False` ⇒ `False`) gemeldet werden sollen. Mit `Intervall` kann die Zeitspanne zwischen zwei gleichbleibenden Meldungen eingestellt werden, `Invert` invertiert das eingehende Signal und `InitState` definiert den Startzustand des Sensors.

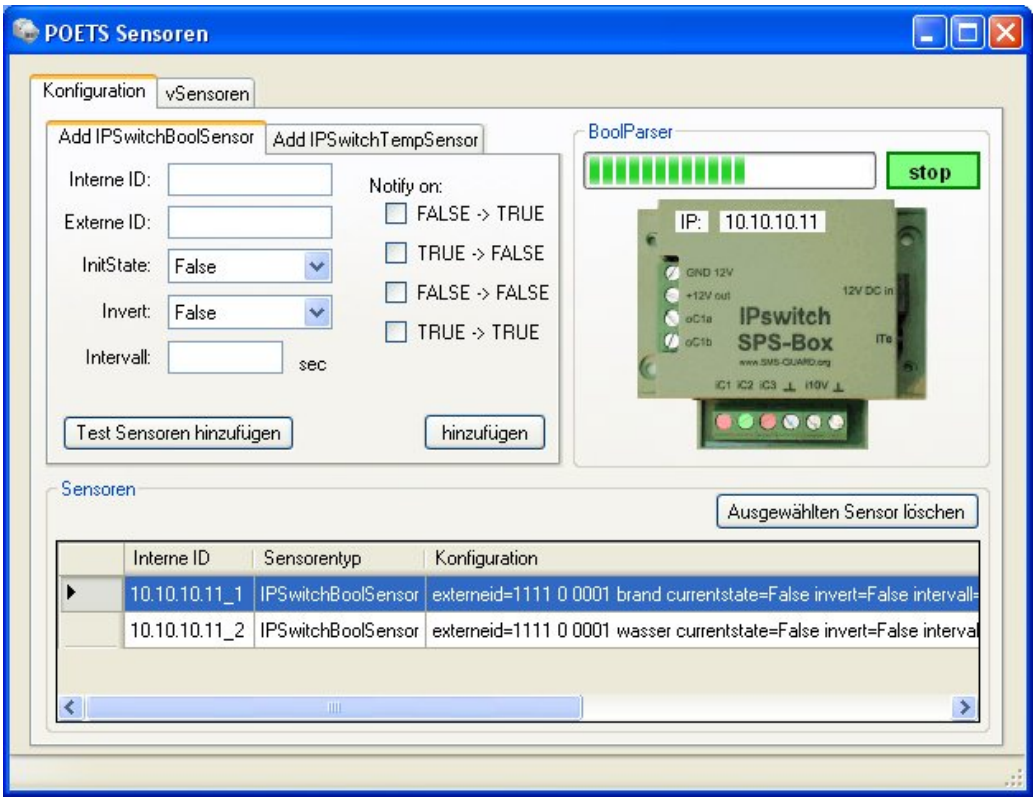


Abbildung 44: POETS Sensoren

`Interne ID` dient zur Identifizierung der jeweiligen Sensoren und wird durch das Muster „*IP-des-IPSwitch _ Sensoreingang*“ beschrieben. Der zweite Sensor des *IPSwitch* mit der IP 10.10.10.11 hätte folglich die `Interne ID` „*10.10.10.11_2*“. `Externe ID` dient der Zuordnung des Sensoren in das aktuelle Szenario. Die Zuordnung findet im Sensor-Service statt.

9.6.3 Sensor-Service

Die *IGM-Plattform* bietet im *Context-Subsystem* zwar auch eine Möglichkeit Sensoren anzubinden, allerdings steht noch keine Schnittstelle für den Zugriff auf die benötigten Funktionalitäten, wie das Setzen von Werten oder die Benachrichtigung über geänderte Sensordaten, zur Verfügung. Wir haben uns daher dazu entschlossen, einen eigenen *Sensor-Service* zu entwerfen, der direkt in der Datenbank des *Context-Subsystems* Änderungen der angebundenen Sensoren vermerkt. Der Service wurde, wie das *BPM-Subsystem*, als *.NET 3.0 WCF-Service* implementiert, ist aber getrennt und unabhängig vom *BPM-Subsystem* einsetzbar. Die Schnittstelle des *Sensor-Services* (siehe Abbildung 45) enthält nur eine Methode: `UpdateSensorData`. Diese Methode kann von externen Sensoren aufgerufen werden, die ihre aktuellen Daten in der `SensorData` Datenstruktur (siehe Abbildung 46) übergeben.
Der *Sensor-Service* ordnet die Daten dann anhand der übergebenen ID intern einem Sensor zu. Die Schnittstelle für interne Sensoren (siehe Abbildung 47) enthält ebenfalls nur die Methode `UpdateSensorData`. Die Sensordaten werden also vom Service nur an den konkreten Sensor weitergeleitet. Für die Benachrichtigung der *IGM-Plattform* gibt es die Sensor-Implementierung `IGMSensor`, die die Daten in die entsprechende Tabelle in der Datenbank schreibt.

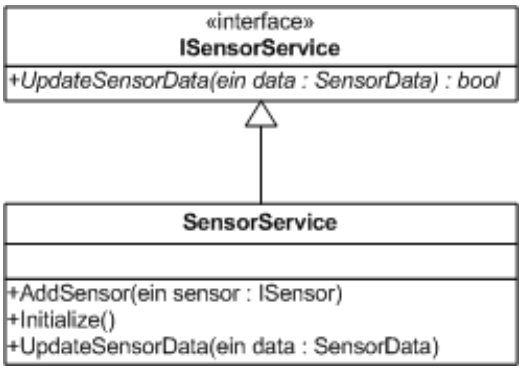


Abbildung 45: Die Schnittstelle `SensorService` und die Implementierung

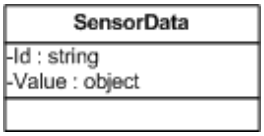


Abbildung 46: Die Klasse `SensorData`

Abbildung 47: Die Schnittstelle `ISensor` mit der abstrakten Hilfsimplementation `SensorSupport` und der konkreten Implementierung `IGMSensor`

DTO	Übergabewerte	Funktion
GetEmergency	Domicile.domicileID	Schlüssel
	Domicile.Emergency.emergencyType	Schlüssel
	Domicile.Emergency.emergencyTypeID	lesen
	Domicile.Emergency.isServiceActive	lesen
	transactionID	Schlüssel
GetUser	Domicile.domicileID	Schlüssel
	User.userID	lesen
	User.name	lesen
	User.forename	lesen
	User.gender	lesen
	transactionID	Schlüssel
GetDomicile	Domicile.domicileID	Schlüssel
	Domicile.street	lesen
	Domicile.houseNumber	lesen
	Domicile.city	lesen
	Domicile.postCode	lesen
	transactionID	Schlüssel
getServiceProvider	User.userID	Schlüssel
	User.Emergency.emergencyTypeID	Schlüssel
	User.Emergency.ServiceProvider[]	lesen
	transactionID	Schlüssel
getUserNotifyInfo	User.userID	Schlüssel
	User.Emergency.emergencyTypeID	Schlüssel
	User.Emergency.alarmNotification[]	lesen
	transactionID	Schlüssel
getSPNotifyInfo	ServiceProvider.serviceProviderID	Schlüssel
	ServiceProvider.Emergency.alarmNotification[]	lesen
	transactionID	Schlüssel
IGMPlatform.Time	IGMPlatform.Time.Received	lesen
IGMPlatform.Status	IGMPlatform.Status.Context	lesen
	IGMPlatform.Status.DIA	lesen
	IGMPlatform.Status.EVA	lesen
	IGMPlatform.Status.Service	lesen
	IGMPlatform.Status.User	lesen
	IGMPlatform.Status.BPM	lesen
	IGMPlatform.Status.BPM.BPMLoggingService	lesen
	IGMPlatform.Status.BPM.BPMProcessHandler	lesen
	IGMPlatform.Status.BPM.CommunicationManager	lesen
	IGMPlatform.Status.BPM.DataManager	lesen
	IGMPlatform.Status.BPM.ILogEventBroker	lesen
	IGMPlatform.Status.BPM.ILogEventManager	lesen
	IGMPlatform.Status.BPM.WSEventManager	lesen
	IGMPlatform.Status.BPM.Database	lesen
	IGMPlatform.Status.BPM.MSMQ	lesen

Tabelle 6: Diese *DTO*'s wurden für die Funktion *retrieveData* eingeführt. Die Datenpakte *IGMPlatform.Time* und *IGMPlatform.Status* sind der aus Vollständigkeitsgründen mit aufgeführt.

DTO	Übergabewerte	Funktion
WriteLog	database	
	dbEntry	schreiben
	transactionID	
WriteDiscount	notation	schreiben
	description	schreiben
	serviceDate	schreiben
	serviceProvisionDate	schreiben
	serviceProviderID	schreiben
	userID	schreiben
	quantity	schreiben
	transactionID	

Tabelle 7: Diese *DTO*'s wurden für die Funktion *sendData* eingeführt.

Name	Übergabewerte	Funktion
sendEmail	Message.body	schreiben
	Message.sender	schreiben
	Message.recipient	schreiben
	Message.returncode	lesen
	transactionID	Schlüssel
sendSMS	Message.body	schreiben
	Message.sender	schreiben
	Message.recipient	schreiben
	Message.returncode	lesen
	transactionID	Schlüssel

Tabelle 8: Diese *DTO*'s wurden für die Funktion *sendMessage* eingeführt.

Damit der *Sensor-Service* die Daten an den richtigen Sensor weiterleiten kann, muss der Sensor in der XML-Konfigurationsdatei des Services eingetragen sein. Im Fall des *IGM-Sensors* sieht Konfiguration für einen Brandsensor wie in Listing 10 aus.

Listing 10: XML-Konfiguration des Brandsensors

```
<Sensors>
    <addSensor id="1111 0 0001 brand" type="De.UniDo.Poets.Isst.
       Sensors.SensorService.IGMSensor" assembly="Sensors.
       SensorService.exe">
        <Attributes>
            <addAttribute name="type" value="Brand"/>
            <addAttribute name="component" value="F1"/>
            <addAttribute name="flat" value="1111 0 0001"/>
            <addAttribute name="room" value="Wohnzimmer"/>
        </Attributes>
    </addSensor>
</Sensors>
```

Der Sensor hat eine eindeutige ID. Mit dem Tag `<addSensor>` wird ein neuer Sensor zur Konfiguration hinzugefügt. Das Attribut `type` gibt den Namen der Klasse an, die den Sensor implementiert und das Attribut `assembly` bestimmt die *Assembly*, d.h. die C#-DLL-Bibliothek, die die kompilierte Klasse enthält. Mit den XML-Tags `<Attributes>` und `<addAttribute>` ist es prinzipiell möglich beliebige Attribute mit ihren Werten der Sensor-Klasse mitzugeben. Dabei muss allerdings berücksichtigt werden, dass die Sensor-Klasse bestimmte Werte erwartet. Der Vergleich von Listing 10 mit Abbildung 47 zeigt, dass die Klasse `IGMSensor` genau die in der Konfiguration angegebenen vier Attribute erwartet.

9.7 Testszenarien

9.7.1 jABC-Plattform

Um die *jABC-Plattform* unabhängig vom IGM-Server testen zu können, wurde ein Testserver entwickelt, der quasi die *IGM-Plattform* lokal zur Verfügung stellt. Dieser Testserver läuft auf dem selben Webserver, auf dem auch die *jABC-Plattform* läuft. Dies kann auch auf dem Entwicklungsrechner sein.
Diese Idee bringt mehrere Vorteile:

- es wird kein Internet gebraucht, da beide Server auf der gleichen Webserver-Engine (Tomcat) laufen.
- Aufruf von *Sensoren* (siehe 9.6) wird direkt vom Webservice gekapselt und weiter verarbeitet.
- schneller Zugriff auf die XML-Prozessdateien mittels *jABC*

Service-Methoden Der lokale Testserver soll folgende Service-Methoden zur Verfügung stellen:

- *retrieveData*,
- *sendData*,
- *sendEvent*,
- *sendMessage* und

- *updateSensorData*.

An dieser Stelle sollen nur die Methoden *retrieveData* und *updateSensorData* stellvertretend vorgestellt werden.

retrieveData Diese Methode besitzt als Parameter ein DTO und soll anhand der angegebenen Parameter die dafür entsprechenden Daten zurück liefern.
Im Fall der *DTO GetUser* könnten die ausgetauschten Nachrichten beispielsweise folgendermaßen aussehen:
Eingehende Nachricht:

```
GetUser : {transactionID=123, Domicile.domicileID=1111 0 0001} :
{User.userID= ,user.name= ,User.forename= ,User.street= ,User.gender= }
```

Ausgehende Nachricht:

```
GetUser : {transactionID=123, Domicile.domicileID=1111 0 0001} :
{User.userID=242,User.name=Schmidt, User.forename=Harald,
User.street=Am Tiergarten 13,User.gender=männlich}
```

UpdateSensor Diese Methode nimmt Sensordaten(`SensorID` und `Status`) entgegen und verarbeitet sie weiter. Eine `SensorID` besteht aus `domicileID` und `emergencyType`: z.B. `1111 0 0001 Brand`. `Status` erhält den Wert `true` oder `false`.

Database Die *Datenbank* des lokalen Testservers besteht aus XML-Dateien, die die Testdaten als Ersatz für die Datenbank der *IGM-Plattform* enthalten. Diese Datenbank besteht aus den XML-Dateien `user.xml`, `Domicile.xml`, `ServiceProvider.xml` und `Sensor.xml` Für deren Validierung wurden ebenfalls entsprechende Document Type Definitions (DTDs) erstellt.

user.xml Diese Datei enthält Daten, die den Bewohner betreffen: `User.user.ID`, `User.name`, `User.vorname`, `User.street`, `User.gender` und `DomicileID`. Alle Elemente sind vom Typ String (`#CDATA`) und die `userID` bekommt zusätzlich den Schlüsselbezeichner `ID`, damit diese eindeutig ist.

Ein Beispiel:

```
<User User.userID="U242">
 <User.name>Schmidt</User.name>
<User.forename>Harald</User.forename>
<User.street>AmTiergarten 13</User.street>
<User.gender>männlich</User.gender>
<domicileID>1111 0 0001</domicileID>
   ........
........
</User>
```

Hier können noch weitere Informationen nachgefragt werden, z.B. wie der Bewohner im Fall von Brand, Gas, Wasser und Einbruch benachrichtigt werden soll.

Domicile.xml Diese Datei enthält Daten, die die Wohnung betreffen: `domicileID`, `User.userID`, `Domicile.postCode`, `Domicile.city`, `Domicile.housenumber`, `Domicile.street` und `Domicile.Emergency`. Damit auch hier wieder `domicileID` eindeutig ist, wird diese ebenfalls mit dem Schlüsselbezeichner ID gekennzeichnet.
In *Domicile.Emergency* werden die folgenden Daten gekapselt:
`emergencyTypeID`, `emergencyType`, `room`, `component`, `isServiceActive` und alle für den `emergencyType` (Brand, Wasser, Gas, etc.) zuständigen `serviceProvider`.

```
<Domicile domicileID="d111100001">
  <User.userID>242</User.userID>
  <Domicile.postCode>45525</Domicile.postCode>
  <Domicile.city>Hattingen</Domicile.city>
  <Domicile.houseNumber>13</Domicile.houseNumber>
  <Domicile.street>Am Tiergarten</Domicile.street>
    ..........
    .......
</Domicile>
```

Zur Erklärung:
`domicileID` müßte beispielsweise so aussehen: `1111 0 0001`.
Da sie aber vom Typ ID ist, darf sie keine Leerzeichen enthalten und die erste Stelle muss mit einem Buchstaben belegt werden. Deswegen sieht sie sie dann so aus: `d111100001`.

ServiceProvider.xml Mit der `ServiceProviderID`, die mit dem Schlüsselbezeichner ID gekennzeichnet ist, wird der Dienstleister identifiziert. Im Element `alarmNotification.Media` wird angegeben, wie der Dienstleister benachrichtigt wird.

```
<ServiceProvider serviceProviderID="S47260">
  <ServiceProvider.Emergency>
    <emrgencyTypeID>7</emrgencyTypeID>
    <alarmNotification.Media>
      <email>
        <alarmNotification.Data>
          manuel@cwaid.de
        </alarmNotification.Data>
      </email>
      <SMS>
        <alarmNotification.Data>
          01777872209
        </alarmNotification.Data>
      </SMS>
    </alarmNotification.Media>
  </ServiceProvider.Emergency>
</ServiceProvider>
```

Sensor.xml Die Datei enthält alle in der Wohnung angemeldeten Sensoren (siehe Kap. 9.6) und hat folgende Form:

```
<?xml version="1.0"?>
<!DOCTYPE sensors SYSTEM "sensor.dtd">
<sensors>
```

```
    <sensor sensorID="s111100001">
        <sensorType>Gas</sensorType>
        <sensorType>Wasser</sensorType>
    </sensor>
    <sensor sensorID="s111100002">
        <sensorType>Gas</sensorType>
    </sensor>
</sensors>
```

9.7.2 IGM

Um die Funktionsweise der einzelnen Komponenten und Services des *BPM-Subsystems* sicher zu stellen und auch längerfristig Möglichkeiten zur Verifizierung der korrekten Implementierung der Service-Schnittstellen zu schaffen, wurden einige automatisierte *Unit-Tests*[18] geschrieben. Als Grundlage wurde dazu das Framework *NUnit*[8] für .Net 2.0 in der Version 2.2.9 genutzt. Zunächst sollten diese *Unit-Tests* dazu dienen alle Klassen und Assemblies isoliert von ihren Abhängigkeiten zu testen. Hierbei erwieß sich als großer Vorteil, dass WCF Services, im Gegensatz zu den älteren ASP.Net WebServices, bereits als Libraries vorliegen und somit verhältnismäßig einfach einem *Unit-Test* unterzogen werden konnten, indem die Klassen instanziert wurden. Listing 11 zeigt ein fiktives Beispiel.

Listing 11: einfaches (fiktives) Beispiel für einen *Unit-Test*

```
[Test]
public void InvokeSayHello() {
  MyService service = new MyService();
  string hello = service.SayHello("World");
  Assert.AreEqual("Hello World!", hello);
}
```

Die *Unit-Tests* beschränkten sich auf das Testen von Randfällen, wie z.B. das Testen des Verhaltens bei der Übergabe von `Null`-Parametern, der Übergabe nicht unterstützter *Event-* und *DataTransferObject* -Typen sowie nicht in der Parametertaxonomie definierter Parameter. Wie beim Testen üblich ergab sich das Problem der Abhängigkeiten einiger Klassen und Komponenten untereinander. So stießen die *Unit-Tests* schnell dort an ihre Grenzen, wo beispielsweise Abhängigkeiten zu Datenbanken und andere Services vorhanden waren. Aus zeitlichen Gründen wurde darauf verzichtet sämtliche Tests durch sogenannte *Mock-Objects*[19] zu erweitern. Diese hätten zwar den Vorteil gehabt, die Kopplung der einzelnen Komponenten zu umgehen, allerdings hätte das Testen soviel Zeit in Anspruch genommen, dass die eigentlichen Ziele der Implementierung in den Hintergrund getreten wären. Stattdessen wurde versucht durch *Integration-Tests*[20] das Zusammenspiel der einzelnen Services zu testen. Auch diese *Integration-Tests* wurden teils durch den Einsatz von *NUnit* automatisiert. Der Unterschied zu den *Unit-Tests* bestand dabei im wesentlichen darin, dass die einzelnen Services per *Self-Hosting*[21] gehostet wurden und somit die volle Funktionalität als Service getestet werden konnte. Dies schließt auch die Serialisierung und Deserialiserung von gesendeten und empfangen Nachrichten ein. Listing 12 zeigt ein fiktives Beispiel für einen solchen Test mittels *Self-Hosting* für einen zustandslosen Service.

[18] Modul- bzw. Komponententest zur Verifikation der Korrektheit von Softwaremodulen wie z.B. Klassen

[19] Dummy-Objekte, die als Platzhalter für Objekte im Rahmen von Unit-Tests eingesetzt werden.

[20] Einzeltests, um unterschiedliche, aber voneinander abhängige, Komponenten eines Gesamtsystems im Zusammenhang zu testen.

[21] Im Umfeld von WCF wird so das Hosten eines Services durch eine (eigenständige) Applikation bezeichnet.

Listing 12: einfaches (fiktives) Beispiel für einen *Integration-Test* mit *Self-Hosting*

```
public class MyService : IMyService {
  public string SayHello(string name)    {
      return "Hello " + name + "!";
  }
}

[TestFixture]
public class MyServiceTest {

  private static ServiceHost sh;

  [TestFixtureSetUp]
    public static void TestFixtureSetUp() {
    MyServiceTest.sh = new ServiceHost(typeof(MyService));
    MyServiceTest.sh.Open();
  }

  [TestFixtureTearDown]
  public static void TestFixtureTearDown() {
    MyServiceTest.sh.Close();
  }

  [Test]
  public void InvokeSayHello()
  {
    using (MyServiceClient client = new MyServiceClient())
    {
      string hello = client.SayHello("World");
      Assert.AreEqual("Hello World!", hello);
    }
  }
}
```

Bei den *Integration-Test* wurden vor allem die Komponenten *BPMProcessHandler* und *IGMEventBroker* getestet. Dies ist der Tatsache geschuldet, dass sie die meisten Verknüpfungen und Abhängigkeiten zu den meisten anderen Services haben. So dient der *BPMProcessHandler* als Proxy vor den Komponenten *WSEventManager*, *CommunicationManager* und *DataManager*. Durch das Testen des *BPMProcessHandlers* werden diese Services implizit mitgetestet. Der *IGMEventManager* wird durch Tests des *IGMEventBrokers* ebenfalls implizit mitgetestet (vgl. hierzu auch Abbildung 31). Darüber hinaus stellt der *BPMProcessHandler* die Schnittstelle zur *jABC-Plattform* dar und der *IGMEventBroker*, die Schnittstelle für den Austausch von Events mit der *IGM-Plattform*. Das Hauptaugenmerk der Tests auf diese Komponenten zu beschränken, ist daher in zweierlei Hinsicht begründet – es konnte eine Menge Zeit für das Testen der anderen Komponenten eingespart werden und gleichzeit die Schnittstellen zu anderen Applikationen wie dem *jABC* und zu anderen Subsystemen der *IGM-Plattform* getestet werden.

Neben diesen automatischen Tests, wurden die Implementierungen aller einzelnen Komponenten immer wieder durch manuelle Tests überprüft. So wurden beispielsweise mit einer Test-Applikation immer wieder *Events* und *DataTransferObjects* erzeugt und mit diesen die einzelnen Methoden der Services und Klassen getestet.

9.7.3 Testdaten

Die Dantenbanken der *IGM-Platform* die uns zur Entwicklung zur Verfügung gestellt wurde, war zu Beginn mit einigen Datensätzen gefüllt. Es existierten schon die nötigen Dienst-

leister sowie deren Kontaktinformationen, einige Häuser mit Wohnungen und Bewohnern waren eingepflegt und einige Bewohner hatten auch schon Dienste abboniert. Obwohl die meisten Daten durch Restraints auf den Datenbanken abgesichert waren, wiesen einige Datensätze bei den Bewohnern Lücken auf. Aus diesem Grund wurden fünf neue Bewohner, Herr Schmidt, Herr Andrak, Frau Licard, Herr Pocher und Herr Feuerstein angelegt, die jeweils eine Wohnung aus zwei unterschiedlichen neuen Häuser bewohnten. Für diese neuen Bewohner wurden unterschiedliche Dienste abboniert und unterschiedliche Benachrichtigungskonfigurationen hinterlegt. Auf Seiten der Dienstleister wurden nur die entsprechenden Benachrichtigungseinstellungen angepasst, hier waren E-Mail-Adressen von ISST-Mitarbeiten eingetragen. Als Beispiel dient Herr Schmidt: *Herr Schmidt wohnt in der Wohnung „1111 0 0001“ und möchte, dass im Falle von Brand/Wasser/Gas/Einbruch er eine SMS (Prio1) und falls nicht eine E-Mail (Prio2) bekommt. Alle Dienstleister sollen so benachrichtigt werden, wie die Dienstleister es eingestellt haben.*

9.7.4 Gesamtsystem

Voraussetzung für den Test des Gesamtsystems sind erfolgreiche Tests der Einzelplattformen.
Aufgrund der verteilten Architektur läßt sich der Gesamttest kaum automatisch durchführen, so dass eine Überprüfung des Systems anhand der visuellen Kontrolle der durch das Monitoring gelieferten Daten erfolgen muss. Als Grundlage hierfür dienen die zuvor genannten Testdaten.

Ein solcher Testvorgang läßt sich in zwei Phasen unterteilen

- dem Kommunikationstest und
- dem Interaktionstest.

Kommunikationstest Zum Testen der jABC-zu-IGM-Kommunikation wurde auf der *jABC-Plattform* ein Testprozess ausgeführt, der alle Webservicemethoden der *IGM-Plattform* mit sinnvollen Parametern aufrief. Anhand der Transaktions-Logs und der Log-Dateien auf den beiden Plattformen konnten dann deren Rückgabewerte verifiziert werden.
Für die andere Kommunikationsrichtung mussten Werte in der Datenbank manipuliert werden, damit ein Event über die `sendEvent`-Methode der *jABC-Plattform* übermittelt wurde. Das Resultat auf der *jABC-Plattform* konnte nun ebenfalls anhand des Monitorings überprüft werden.
Prinzipiell testet jede Ausführung eines von uns entwickelten Prozesses auf der *jABC-Plattform* die komplette Kommunikation, da dort viele Daten mit der *IGM-Plattform* ausgetauscht werden müssen.

Interaktionstest Ist die Korrektheit der Kommunikation sicher gestellt, kann überprüft werden, ob die beiden Systeme erwartungsgemäß miteinander korrekt interagieren können.
Ganz lapidar bedeutet dies, ein Event auf der *IGM-Plattform* auszulösen, dass eine Prozessausführung auf der *jABC-Plattform* in Gang setzt, die wiederum Daten anfordert und gegebenenfalls eigene Events auf der *IGM-Plattform* auslöst. Dieser Vorgang muss nun anhand des Monitorings auf Korrektheit überprüft werden.

Teil IV
Erweiterbarkeit

10 Erweiterung des Gesamtsystems

10.1 Voraussetzungen

10.1.1 IGM

Für Erweiterungen der *IGM-Plattform* bzw. des *BPM-Subsystems* sind grundsätzliche Vorkenntnisse in den Bereichen Microsoft .Net, C# und Windows Communication Foundation (WCF) notwendig. Der Einsatz einer Entwicklungsumgebung wie Microsoft Visual Studio ist empfehlenswert, wenn auch nicht zwingend notwendig. Weitere benötigte Tools werden in den nachfolgenden Kapiteln sowie im Entwicklungshandbuch[43] erwähnt.
Für die Erweiterung sind darüber hinaus Kenntnisse über die Schnittstellen aller Subsysteme der *IGM-Plattform* erforderlich. Für die Erweiterung um neue *DTOs* werden außerdem die nachfolgenden Anforderungen gestellt:

DataMapper/DTOs Um *DTOs* zu ändern oder evtl. neue *DTOs* zu entwickeln sind Kenntnisse über das *User-Subsystem*, das *Service-Subsystem* und das Datenbankschema des Projektes *SmarterWohnen* notwendig. Die Kenntnisse der beiden Subsysteme sind wichtig, weil diese nur einen stark eingeschränkten Zugriff auf den Datenbestand erlauben. Die Datenanfrage wird bei beiden Subsystemen mit *Profile*-Objekten[22] formuliert. Als Rückgabe erhält man ein Feld (Array) von *Profile*-Objekten, das die gesuchten Daten enthält. Die Rückgabe muss immer auf `null` und Null-Länge geprüft werden. Weil die Rückgabe auch bei einzelnen Methoden eines Subsystems nicht konsistent ist, müssen immer beide Fälle geprüft werden. Die Kenntnisse über das Datenbankschema sind insofern wichtig, dass man wissen muss, welche Daten abgefragt werden können und welche Relationen zwischen Daten bestehen.

10.1.2 jABC

Für Erweiterungen der *jABC-Plattform* sind Java-Kenntnisse sowie grundlegende Kenntnisse über den Aufbau des *jABC* nötig. Als Entwicklungsumgebung kommt *Eclipse* zum Einsatz, die Weiterentwicklung ist aber auch mit jeder anderen IDE möglich. Details zur Projekt-Einrichtung finden sich im jABC-Entwicklungshandbuch.

10.2 Erweiterung der Events

10.2.1 IGM

Um die *IGM-Plattform* um neue Events zu erweitern müssen zunächst zwei "Arten" von Events unterschieden werden. Zum einen gibt es Events, die von der *jABC-Plattform* an das *BPM-Subsystem* gesendet werden, zum anderen Events, die vom *BPM-Subsystem* an das *jABC* gesendet werden. Erstere werden in diesem Abschnitt als *jABC-Events* und letztere als *IGM-Events* bezeichnet.

[22] s. auch Listing 20

jABC-Events Um die *IGM-Plattform* um *jABC-Events* zu erweitern, ist prinzipiell nur eine Erweiterung der Konfiguration nötig. *JABC-Events* werden dem *BPM-Subsystem* durch die Konfiguration von *WSEventHandlern* bekannt gemacht. Damit ein *jABC-Event* den anderen Subsystemen zur Verfügung gestellt werden kann, genügt es die Konfiguration des `GenericLogHandlers` zu erweitern. Dieser *WSEventHandler* stellt einen Implementierung des Interfaces `IWSEventHandler` bereit. Ein eingehendes *jABC-Event* wird von dem Handler entgegengenommen und ein Eintrag über den Empfang in eine Log-Datei geschrieben. Anschließend wird das *jABC-Event* an alle Subsysteme übermittelt, die eine entsprechende *Subscription* für Events dieses Typs am *BPM-Subsystem* gemacht haben. Die Funktionalität der Übermittlung des *jABC-Events* ist allerdings unabhängig von der Implementierung eines *WSEventHandlers*, da diese Funktionalität im *WSEventManager* implementiert ist. Eine Konfiguration, die das *BPM-Subsystem* auf diese Weise um ein Event *NewEvent* erweitert, sieht folgendermaßen aus:

Listing 13: Auszug aus der Konfiguration des *WSEventManagers* zur Erweiterung um den Eventtypen *NewEvent*

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="WSEventManager"
      type="De.UniDo.Poets.Isst.IGM.BpmSubSystem.WSEventManager.
          ConfigurationHandlers.WSEventManagerSection, BPMSubSystem.
          WSEventManager" />
    <section name="GenericLogHandler"
      type="De.UniDo.Poets.Isst.IGM.BpmSubSystem.WSEventManager.
          WSEventHandlers.ConfigurationHandlers.
          GenericLogHandlerSection, BPMSubSystem.WSEventManager.
          WSEventHandlers" />
    <!-- other config sections omitted-->
  </configSections>

  <!-- appSettings omitted-->

  <!-- WSEventManager config -->
  <WSEventManager>
    <!-- WSEventHandler config-->
    <WSEventHandlers>
      <clearHandlers />
      <addHandler name="GenericLogHandler"
        assembly="BPMSubSystem.WSEventManager.WSEventHandlers.dll"
        implementation="De.UniDo.Poets.Isst.IGM.BpmSubSystem.
            WSEventManager.WSEventHandlers.GenericLogHandler" />
      <!-- add other handlers here -->
    </WSEventHandlers>
  </WSEventManager>

  <!-- GenericLogHandler config -->
  <GenericLogHandler Logger="">
    <SupportedEventTypes>
      <clearEventTypes />
      <addEventType typeName="NewEvent" />
      <!-- add other event tpyes here -->
    </SupportedEventTypes>
  </GenericLogHandler>

  <!-- config of system.serviceModel omitted -->
```

```
</configuration>
```

Sollen auf der *IGM-Plattform* durch das *BPM-Subsystem* beim Empfang eines Events weitere Aktivitäten ausgeführt werden, muss durch die Implementierung des Interfaces `IWSEventHandler` ein neuer *WSEventHandler* implementiert werden. Dieser kann dann wie der `GenericLogHandler` durch die Konfiguration dem *BPM-Subsystem* bekannt gemacht werden (vgl. Listing 13).

Eine Implementierung kann folgendermaßen aussehen:

Listing 14: Implementierung eines *WSEventHandlers*

```
using System;
using System.Collections.Generic;
using System.ServiceModel;
using System.Text;

using De.UniDo.Poets.Isst.IGM.BpmSubSystem.Common.Handlers;
using De.UniDo.Poets.Isst.IGM.BpmSubSystem.Common.Contracts.Data;
using De.UniDo.Poets.Isst.IGM.BpmSubSystem.Common.Contracts.Faults;

namespace De.UniDo.Poets.Isst.IGM.BpmSubSystem.WSEventManager.
    WSEventHandlers
{
    public class ExampleHandler : IWSEventHandler
    {
        public ExampleHandler() {
        }

        #region IWSEventHandler Member

    // defines which event types are handled by this handler
        public List<string> SupportedEventTypes
        {
            get {
                List<string> eventTypes = new List<string>();
                eventTypes.Add("ExampleEvent");
                return eventTypes;
            }
        }

    // handles an event
        public bool handleEvent(Event evt)
        {
            if (null == evt)
            {
        // this can only happen if WSEventManager implementation is
            broken
                EventNotSupportedException ensex =
          new EventNotSupportedException("evt must not be null");
                throw new FaultException<EventNotSupportedException>(
                    ensex, "evt was null");
            }
            if (!SupportedEventTypes.Contains(evt.Type))
            {
        // this can only happen if WSEventManager implementation is
            broken
                EventNotSupportedException ensex =
```

```
            new EventNotSupportedException("events of type<" + evt.Type
                + "> are not supported by handler<" + this.GetType().
                ToString() + ">");
                throw new FaultException<EventNotSupportedException>(
                    ensex, "events of type<" + evt.Type + "> are not
                    supported");
            }
            checkEventParameters(evt);

        // implement custom actions here
            return true;
        }

        public bool checkEventParameters(Event evt)
        {
            if (null == evt || null == evt.Keys)
            {
        // this can only happen if WSEventManager implementation is
            broken
                InvalidParametersException ipex = new
                    InvalidParametersException("evt and evt.Keys must
                    not be null");
                throw new FaultException<InvalidParametersException>(
                    ipex, "evt or evt.Keys was null");
            }

        // check all mandatory parameters here and throw a
        // FaultException<InvalidParametersException> if parameters are
            invalid

            return true;
        }

        #endregion
    }
}
```

Für weitere Details sei an dieser Stelle auf die API-Dokumentation für das *BPM-Subsystem*[42] verwiesen.

IGM-Events *IGM-Events* werden über den WebService an das *jABC* versendet. Zu diesem Zweck kann aus der WSDL ein Proxy erzeugt werden oder über die *ChannelFactory* (`System.ServiceModel.ChannelFactory<TChannel>`) ein entsprechender Kanal aufgebaut werden. Das benötigte Interface `De.UniDo.Poets.Isst.IGM.BpmSubSystem.Common.Contracts.Services.IJABCWSProxy` findet sich in der Assembly *BPMSubSystem.Common*. Nachfolgendes Code-Beispiel zeigt, wie der Versand von Events prinzipiell funktioniert:

Listing 15: Beispiel zum Versand von Events an die *jABC-Plattform*

```
ChannelFactory<IJABCWSProxy> factory = null;
try
{
  // create an event
    Event wsEvent = new Event();
  // set event type
    wsEvent.Type = "ExampleEvent";
  // add keys (parameters) like this
```

```
    wsEvent.Keys.Add(new KeyValueStringString("ExampleKey", "
        ExampleValue"));

    // create channel
    BasicHttpBinding binding = new BasicHttpBinding();
    EndpointAddress address = new EndpointAddress("http://pulicis.cs.
        uni-dortmund.de:8080/JABCPlatform/service/ISST2JABC");
    factory = new ChannelFactory<IJABCWSProxy>(binding, address);
    IJABCWSProxy proxy = factory.CreateChannel();
    if (proxy.sendEvent(wsEvent))
    {
    // maybe you want to do something in case of succes ...
    }
    else
    {
    // ... or in case of failure
    }

}
// catch defined soap faults
catch (FaultException<EventNotSupportedException> ensex)
{
  // handle soap fault
    if (null != factory)
        factory.Abort();
}
catch (FaultException<InvalidParametersException> ipex)
{
  // handle soap fault
    if (null != factory)
        factory.Abort();
}
// catch otder soap faults
catch (FaultException fex)
{
  // handle soap fault
    if (null != factory)
        factory.Abort();
}
// handle other exceptions
catch (CommunicationException cex)
{
   if (null != factory)
        factory.Abort();
}
catch (Exception ex)
{
    if (null != factory)
        factory.Abort();
}
// close the channel
finally
{
    if (null != factory &&
        factory.State != CommunicationState.Faulted &&
        factory.State != CommunicationState.Closed)
    {
        try
```

```
            {
                factory.Close();
            }
            catch
            {
                factory.Abort();
            }
        }
    }
```

Um in der *IGM-Plattform* auftretende Events an die *jABC-Plattform* weiterzuleiten, muss für jeden Eventtypen ein entsprechender *IGMSubscriptionHandler* implementiert werden und mittels der Konfiguration des *IGMEventManagers* dem *BPM-Subsystem* bekannt gemacht werden. Eine Implementierung eines solchen *IGMEventManagers* stellt der `GenericEmergencyHandler` dar. Dieser Handler leitet die in der Parametertaxonomie definierten Events wie *NotifyBreakin*, *NotifyGasLeak* etc. an das *jABC* weiter. Durch diesen Handler ist es einfach durch eine Konfiguration möglich weitere Events zu definieren. Voraussetzung dafür ist allerdings, dass die Entitäten im *Context-SubSystem* genau so wie die Entitäten für die o.g. Events definiert sind. Existiert zum Beispiel eine Entität *Stromausfall*, könnte ein Event *PowerFailure* wie folgt konfiguriert werden:

Anmerkung: Die Komponente IGMEventManager und die IGMEventHandler werden in den nachfolgenden Beispielen als ILogEventManager und ILogEventHandler bezeichnet, da dies die internen Namen der Komponenten sind und in der Implementierung nicht mehr geändert wurden.

Listing 16: Auszug aus der Konfiguration des *IGMEventManagers* zur Erweiterung um den Eventtypen *PowerFailure*

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

  <configSections>
    <!-- other config sections omitted -->
    <section name="ILogEventManager" type="De.UniDo.Poets.Isst.IGM.
        BpmSubSystem.ILogEventManager.ConfigurationHandlers.
        ILogEventManagerSection, BPMSubSystem.ILogEventManager"/>
    <section name="ILogGenericSubscriptionHandlers" type="De.UniDo.
        Poets.Isst.IGM.BpmSubSystem.ILogEventManager.
        ILogSubscriptionHandlers.ConfigurationHandlers.
        ILogGenericSubscriptionHandlersSection, BPMSubSystem.
        ILogEventManager.ILogSubscriptionHandlers"/>
  </configSections>

  <appSettings>
    <!-- database connection-->
    <add key="ILogEventManager.DBServer" value="databaseserver"/>
    <add key="ILogEventManager.DBUser" value="username"/>
    <add key="ILogEventManager.DBPassword" value="password"/>
    <add key="ILogEventManager.DBDatabase" value="BPMSubSystem"/>
    <add key="ILogEventManager.DBTableExternalRequests" value="
        ExternalRequests"/>
    <!-- other appSettings omitted -->
  </appSettings>

  <!-- ILogEventManager -->
  <ILogEventManager>
    <!-- ILogSubscriptionHandlers-->
```

```
    <ILogSubscriptionHandlers>
      <clearHandlers />
      <addHandler name="PowerFailure" assembly="BPMSubSystem.
          ILogEventManager.ILogSubscriptionHandlers.dll" implementation
          ="De.UniDo.Poets.Isst.IGM.BpmSubSystem.ILogEventManager.
          ILogSubscriptionHandlers.GenericEmergencyHandler" />
      <!-- other handlers omitted -->
    </ILogSubscriptionHandlers>
  </ILogEventManager>

  <!-- config for generic subscription handlers -->
  <ILogGenericSubscriptionHandlers Logger="">
  <!-- config for GenericEmergencyHandler -->
    <GenericEmergencyHandlers>
      <clearHandlers />
      <addHandler name="PowerFailure" entityType="Stromausfall"
          triggeringValue="true"/>
    </GenericEmergencyHandlers>
  </ILogGenericSubscriptionHandlers>

  <!-- other configurations omitted -->

</configuration>
```

Für weitere Informationen zur Konfiguration dieses Handlers sei auch an dieser Stelle auf die API-Dokumentation des *BPM-Subsystems*[42] verwiesen.

Wenn andere, in der *IGM-Plattform* auftretende, Events an das *jABC* gesendet werden sollen, muss ein entsprechender *IGMSubscriptionHandler* implementiert werden. Dafür ist das Interface `IILogSubscriptionHandler` zu implementieren und durch die Konfiguration dem *IGMEventManager* bekannt zu machen. Alternativ kann auch von der Klasse `BaseHandler` geerbt werden. Bei der Implementierung ist entsprechend dafür zu sorgen, dass eine Subscription an einem der anderen Subsysteme gemacht wird. Dazu dient die Methode `Subscribe`. Diese muss eine ID zurück geben, mit der später Events im *BPM-Subsystem* eintreffen. Anhand dieser ID wird die Zuordnung der eintreffenden Events zu den jeweiligen Handlern gemacht. In der Methode `Unsubcribe` muss demnach die Rücknahme einer solchen Subscription implementiert werden. Letztendlich ist in der Methode `handleEvent` dafür zu sorgen, dass ein eintreffendes Event an die *jABC-Plattform* gesendet wird (s. Listing 15). Eine Implementierung für das fiktive, o.g. Event *PowerFailure* auf diese Weise könnte dementsprechend wie folgt aussehen:

Listing 17: Beispiel zur Implementierung eines *IGMSubscriptionHandlers*

```
using System;
using System.Collections.Generic;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.Text;

using De.Fraunhofer.ISST.Ilog.CoreBasicLayer.EventHandling;
using De.Fraunhofer.ISST.Ilog.ContextSubsystem.Query;
using De.Fraunhofer.ISST.Ilog.ModelAccess;

using De.UniDo.Poets.Isst.IGM.BpmSubSystem.Common.Contracts.Data;
using De.UniDo.Poets.Isst.IGM.BpmSubSystem.Common.Contracts.Faults;
using De.UniDo.Poets.Isst.IGM.BpmSubSystem.Common.Contracts.Services;
```

```
namespace De.UniDo.Poets.Isst.IGM.BpmSubSystem.ILogEventManager.
    ILogSubscriptionHandlers
{
    public class PowerFailureHandler : BaseHandler
    {
        // context subsystem
        IContextComponent subSysContext;

        public PowerFailureHandler() : base()
        {
            subSysContext = new ContextComponent();
        }

        public override string Subscribe()
        {
            // create request
            Request request = new Request(0, @".\Private$\BPMSubSystem
                ", "ContextEntityEvent");
            request.AddParameter("EntityType", "Stromausfall"); //
                conforms to entityType in GenericEmergencyHandler
                example
            request.AddParameter("profile_IsTrue", "true"); //
                conforms to triggeringValue in GenericEmergencyHandler
                example
            request.AddParameter("profile_Raum", "Changed");
            request.AddParameter("profile_Komponente", "Changed");
            request.AddParameter("profile_Zeitpunkt", "Changed");
            string extId = null;
            try
            {
                // register request / subscribe
                extId = subSysContext.RegisterRequest(request);
            }
            catch
            {
                // in case of failure return null
                extId = null;
            }
            return extId;
        }

        public override void Unsubscribe(string requestId)
        {
            try
            {
                subSysContext.CancelRequest(requestId);
            }
            catch
            {
                // exception handling omitted
            }
        }

        public override bool handleEvent(De.Fraunhofer.ISST.Ilog.
            CoreBasicLayer.EventHandling.Event evt)
        {
            // entities (domicileIDs in case of SmaWo
            object payload = evt.GetPayload("ContextShift");
```

```
            // return value
            bool retVal = true;

            // check if event and payload is valid here

            IList<string> entities = (IList<string>) payload;
            foreach (string entityId in entities)
            {
                // send event to jabc for every entitiy here
            }

            // set appropriate return value here
            return retVal;
        }

    }
}
```

Weitere Informationen finden sich in der API-Dokumentation des *BPM-Subsystems*[42] sowie in der Spezifikation zur Event-Behandlung[29].

10.2.2 jABC

Um ein Event auf der *jABC-Prozessplattform* innerhalb eines Prozesses zu erstellen, wird ein *SendEvent*-SIB benötigt. Diesem übergibt man den Namen des SIBs sowie eine *HashMap* mit Kontext-Schlüsseln, die auf Daten im Kontext verweisen. Diese Daten werden ausgelesen und zu dem Event hinzugefügt. Dieses Event kann dann mit der Funktion `SendEventToISST` übertragen werden. Weitere Anpassungen sind nicht notwendig.

10.3 Erweiterung der DTOs

10.3.1 jABC

Wie in Kapitel 9.1.2 beschrieben werden die *DTOs* im *jABC* über XML-Dateien definiert. Um die Kommunikation um ein neues *DTO* zu erweitern, muss eine solche XML-Datei erstellt und in den DTO-Ordner (siehe Entwicklerhandbuch) eingefügt werden. Danach ist sie bei der Modellierung im *jABC* als Parameter für ein retrieveData- bzw. sendData-SIB auswählbar.

10.3.2 IGM

Der ursprüngliche Ansatz einer konfigurierbaren *DTO*-Implementierung wurde wegen Unwägbarkeiten im Zugriff auf Daten über die Subsysteme der *IGM-Plattform* verworfen. Stattdessen wurde ein statischer Ansatz, d.h. zu jeder *DTO*-Definition existiert mindestens eine Klasse, die die Definition umsetzt gewählt. Um Erweiterungen an den *DTOs* vorzunehmen muss man entsprechende Klassen anpassen. Im Fall eines vollständig neuen *DTO*s muss eine entsprechende Klasse implementiert werden. Alle *DTO*-Klassen müssen entweder das Interface `IDataMapper` oder `ISendData` implementieren. Fertige Klassen werden in der `DataMapperFactory` verankert. In der aktuellen Version des `DataMappers` ist die `DataMapperFactory` auf eine Hashmap reduziert, die die *DTO*-Objekte unter dem jeweiligen *DTO*-Namen, als Schlüssel, speichert. Die Zustandslosigkeit der *DTO*-Objekte stellt sicher, dass ”concurrency problems” auch bei parallelem Zugriff auf ein *DTO* nicht auftreten.

Listing 18: Implementierung eines *DataMappers*

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data;

using De.UniDo.Poets.Isst.IGM.BpmSubSystem.Common.Contracts.Data;
/// in der Regel braucht man die beiden nachfolgenden Subsysteme
/// um auf Daten der Plattform zu zugreifen.
using De.Fraunhofer.ISST.Ilog.UserSubsystem.Management;
using De.Fraunhofer.ISST.Ilog.ServiceSubsystem;
using De.Fraunhofer.ISST.Ilog.ModelAccess;
using System.ServiceModel;
using De.UniDo.Poets.Isst.IGM.BpmSubSystem.Common.Contracts.Faults;

namespace De.UniDo.Poets.Isst.IGM.BpmSubSystem.DataManager.Mappers{
      public class ExampleDataMapper : IDataMapper     {
      /// <summary>
      /// DTO Name. Sollte jeder DataMapper definieren.
      /// </summary>
      public const string DTO_NAME = "ShowExample";

      #region IDataMapper Members
      public DataTransferObject retrieveData(DataTransferObject data)
                {
         /// 1.Transformation des "data" Objekts in ein "Profile"
            Objekt
         /// 2. Auslesen der Daten über das entsprechende Subsystem
         /// 3. Auswerten der Ergebnismenge
         /// 4. Transformation der Ergebnismenge in ein
            DataTransferObjekt
       }
       #endregion
     }
}
```

Die Implementierung, sowie das Einbinden in die `DataMapperFactory`, des Interfaces `ISendData` verläuft analog.

Listing 19: Einbinden eines *DataMappers* in das System

```
using System;using De.UniDo.Poets.Isst.IGM.BpmSubSystem.Common.
    Contracts.Data;
using De.UniDo.Poets.Isst.IGM.BpmSubSystem.Common.Contracts.Faults;

namespace De.UniDo.Poets.Isst.IGM.BpmSubSystem.DataManager{
      class DataMapperFactory      {
       protected void CreateDataMappers()       {
     //retrieveData
     .....
     this.mappers.Add(ExampleDataMapper.DTO_NAME, new ExampleDataMapper
         ());

                     // sendData

     .....
   }
      }
}
```

10.4 Hinzufügen neuer SIBs

SibCreator Neue *SIBs* müssen im Paket `de.unido.poets.ProcessPlatform.Sibs` erstellt werden. Wenn der *SIBCreator* (siehe Kapitel 7.4.5) benutzt wird, sollte der *jABC*-Projektordner mit dem Projektordner der Entwicklungsumgebung (zum Beispiel *Eclipse*) übereinstimmen. Dann können die erstellten *SIBs* direkt weiter bearbeitet werden. Im *SIBCreator* ist eine Bilddatei als Icon auswählbar, die in Base64-Enkodierung in den Java-Code eingebettet wird. Bei der Gestaltung sollte man sich an die bestehende Farbkodierung (siehe Kapitel 9.4) halten.
SIBs können auch direkt erstellt werden, dabei muss vor allem darauf geachtet werden, dass eine eindeutige `UID` vergeben wird.
Beim späteren Bearbeiten von *SIBs* ist zu beachten, dass bei Veränderungen an den Parametern das *SIB* in allen Graphen ausgetauscht werden muss.

10.5 Hinzufügen neuer Prozesse

10.5.1 Hinzufügen von Prozessmodellen

Die in Kapitel 9.3 beschriebene *jABCProzessPlattform* ermöglicht durch die strikte Trennung von Prozessmodellen einerseits und Prozessmanagementplattform und Ausführungsumgebung andererseits eine dynamische Erweiterbarkeit bei der Erstellung von zusätzlichen Prozesstypen. Sollen neue Prozesstypen zum System hinzugefügt werden, so werden diese zunächst im *jABC* wie in Abschnitt 9.4 modelliert. Die dabei entstehende Prozessbeschreibung in Form von XML-Dateien wird im Modellverzeichnis des Apache Tomcat abgelegt, damit sie von der *jABCProzessPlattform* zur Instanziierung der Prozesse genutzt werden kann. Hierbei entspricht der Name der Datei der Bezeichnung des zu realisierenden Prozesses. Um das neue Prozessmodell der *jABCProzessPlattform* zur Laufzeit bekannt zu machen, ist zudem der Aufruf der Initialisierungsprozedur erforderlich (siehe Entwicklerhandbuch). Erfolgt die Erweiterung hingegen innerhalb eines deaktivierten Systems, so sind die hinzugefügten Prozessmodelle beim Systemstart verfügbar.

10.5.2 Anpassung des Gesamtprozesses

Anpassungen in der Eventverarbeitungsphase Die Erweiterung des Systems um zusätzliche Prozessmodelle erfordert die Anpassung des Gesamtprozesses, da dort das Mapping zwischen Eventtyp und Prozesstyp definiert wird und somit die notwendigen Bedingungen für die Instanziierung eines Prozesstyps festgelegt werden. Im Detail muss daher innerhalb der Eventverarbeitungsphase ein *SIB* vom Typ *StartProcess* eingefügt werden, dessen Parameter `processType` der Name des Prozessmodells zugewiesen wird. Dabei ist zu berücksichtigen, dass dieser Name mit dem Dateinamen des zum Modellverzeichnis hinzugefügten Modells übereinstimmt. Weiterhin ist dieses *SIB* durch eine eingehende Kante ausgehend vom *SIB SwitchAbos* sowie einer ausgehenden Kante zurück zum *SIB WaitingForEvents* in den Prozessgraphen zu integrieren. Die erstgenannte Kante ist dabei mit der Bezeichnung des instanziierenden Eventtypen zu versehen. Anhand von Abbildung 35 in Abschnitt 9.3.4 kann dieses Verfahren analog zu den dort dargestellten *Instanziierungs-SIBs* graphisch nachvollzogen werden.

Anpassungen in der Eventabonnementphase Ist das zur Instanziierung des neu hinzugefügten Prozessmodells benötigte Event innerhalb des Gesamtprozesses bisher noch nicht verwendet worden, so muss dieses durch Modifikationen innerhalb der Eventabonnementphase von dem Gesamtprozess zusätzlich abonniert werden. Dies wird durch Hinzufügen eines SIBs vom Typ *AboEvent* innerhalb dieser Phase erreicht. Da die Abonnierung der Events dort sequentiell erfolgt, kann das neue *SIB* leicht in die Kette der

Abonnements eingefügt werden. Weiterhin ist die Zuweisung der Eventtypbezeichnung zum Parameter `eventType` dieses SIBs notwendig. Die Struktur der Eventabonnementphase ist in Abbildung 34 in Abschnitt 9.3.4 ersichtlicht.

Aktualisieren des Gesamtprozesses Die in den beiden vorherigen Abschnitten beschriebenen Änderungen an den Phasen Eventverarbeitung und -abonnement stellen die Modifikation eines bestehenden Prozesses, also des Gesamtprozesses dar. Die hierzu notwendigen Aktionen werden im folgenden Abschnitt beschrieben.

10.6 Modifikation bestehender Prozesse

Modifikation des Prozessmodells Um ein bereits auf der Plattform vorhandenes Prozessmodell zu modifizieren, wird die zugehörige XML-Datei im *jABC* entsprechend den Änderungswünschen bearbeitet und anschließend wieder im Modellverzeichnis des Systems abgelegt. Die dort bereits vorhandene, nun überarbeitete Version wird dabei überschrieben. Im Anschluss daran ist der Aufruf einer Initialisierungsprozedur notwendig, um das aktualisierte Prozessmodell am System bekannt zu machen. Daraufhin werden alle nach der Aktualisierung gestarteten Prozesse dieses Prozesstyps vom modifizierten Prozessmodell instanziiert und die bei der Modifikation durchgeführten Änderungen sind in diesen Instanzen wirksam. Prozesse, die bereits vor der Aktualisierung instanziiert wurden, sind jedoch nicht von den Modifikationen betroffen und führen bis zu deren Terminierung das nun veraltete Prozessmodell aus. Dies ist bei einer zur Laufzeit durchgeführten Aktualisierung unbedingt zu beachten, um Inkonsistenzen zu vermeiden.

Modifikation des Eventtyp-Mappings Sobald Änderungen in dem Mapping zwischen Eventtyp und Prozesstyp durchgeführt werden sollen, also in der Zuordnung, welcher Prozesstyp von welchem Eventtyp instanziiert wird, ist das Modell des Gesamtprozesses entsprechend im *jABC* anzupassen. Die dabei notwendigen Änderungsaktionen der beiden Phasen Eventabonnement und -verarbeitung sind dabei, wie in Abschnitt 10.5 beschrieben, durchzuführen. Änderungen am Modell des Gesamtprozesses stellen dabei die Modifikation eines bereits bestehenden Prozesses dar, daher ist entsprechend dem vorherigen Abschnitt bei der Aktualisierung desselben zu verfahren.

10.7 Durchführung von Tests

Nachdem die in den vorangegangen Kapiteln beschriebenen möglichen Erweiterungen der *jABC-Plattform* durchgeführt wurden, sollten diese auch getestet werden.
Auf der *IGM-Plattform* kann dies durch Erweiterung der vorhandenen *nUnit*-Tests erfolgen.
Die zur *jABC-Plattform* hinzugefügten *DTOs* können natürlich am besten mit Hilfe von Prozessen, die diese *DTOs* verwenden, getestet werden. Das in die Plattform integrierte Monitoring-Tool kann dann als Debugging-Tool verwendet werden.

Teil V
Ergebnisse

11 Continuous Process Improvement

11.1 Überblick

In der ininitialen Analysephase wurde ein Top-Down-Ansatz gewählt um die vorgegebenen *SmarterWohnen*-Abläufe in Geschäftsprozesse umzusetzen. Durch die Anbindung der *IGM-Plattform* an das *jABC* per Webservice steht der Projektgruppe die Benutzer- und Datenverwaltung sowie die Kommunikationskanäle der Plattform zur Verfügung. Durch das Nachrichtenkonzept der *IGM-Plattform*, das auf die Kommunikation zwischen den beiden Systemen ausgeweitet wurde, besteht außerdem die Möglichkeit ECA-Regelsysteme und Geschäftsprozesse gleichzeitig und in Abhängigkeit voneinander auszuführen. Auf diese Weise können die Stärken von beiden Ansätzen genutzt werden.

Die grundlegenden Funktionen, wie die Erweiterungen der beiden Systeme und deren Verbindung durch Webservices, wurden durch die Projektgruppe umgesetzt. Einige *SmarterWohnen*-Abläufe wurden in Geschäftsprozesse umgesetzt. Hiermit sind wichtige Ziele des Projektes erreicht worden. Auf dieser Basis kann die Projektgruppe das entstandene Gesamtsystem experimentell auf die Fähigkeiten im Bereich des Continuous Process Improvement (CPI) bewerten. Hierbei wird ein Bottom-Up-Ansatz verfolgt um die Prozesse einerseits fachlich und technisch zu verbessern, aber auch die Prozessmodellierung für den Benutzer zu vereinfachen.

Es wurden folgende Punkte gefunden an denen die Prozesse zu verbessern sind:

1. Wiederkehrende Teilprozesse werden in *Graph-SIBs* zusammengefasst.
2. Parallelisierung von Teilprozessen.
3. Modellierung von Ausnahmebehandlungen direkt in den Prozessen.
4. Verbindung von ECA-Regelsystemen und Geschäftsprozessen.
5. Einbindung zusätzlicher Aktivitäten durch neue *SIBs*. (siehe 10.4)

11.2 Wiederverwendbarkeit von Funktionsblöcken

Abläufe wie die Bewohner- und Dienstleisterbenachrichtigung oder das Sammeln von Informationen zu Beginn der Prozesse treten in vielen Modellen in gleicher oder ähnlicher Form auf. Das *jABC* Unterstützt die Wiederverwendbarkeit von Teilmodellen und die Hierarchisierung durch verschiedene *Graph-SIBs*. Die technischen Eigenschaften der verschiedenen, vom *jABC* zur Verfügung gestellten Varianten spielen eine untergeordnete Rolle.

Das Teilmodell wird in einem separaten Modell umgesetzt. In dem Ursprungsmodell wird an der Stelle des Teilmodells ein *Graph-SIB* eingefügt. Dem *Graph-SIB* wird das seperate Modell des Teilprozesses zugeordnet. Alle ein- und ausgehenden Kanten können in dem Untermodell als Modellkanten festegelegt werden und so im Obermodell wieder verwendet werden. Parameter innerhalb des Untermodells, die im Obermodell verfügbar sein müssen, werden als Modellparameter gekennzeichnet.

Die Bewohner- und Dienstleisterbenachrichtigung sind komplexe Teilprozesse, die bei der wiederholten Modellierung aufwändig und fehlerträchtig sind. Außerdem finden sich diese Teilmodelle in vielen Geschäftsprozessen wieder. Die Benachrichtigungen wurden durch *Makro-SIBs* mit den entsprechenden Teilmodellen ersetzt. Dieser Schritt dient einerseits der Vereinfachung der Modellierung und der Vermeidung von Modellierungsfehlern, da die benötigten Untermodelle in getesteter Form bereits vorhanden sind. Andererseits können bekannte Fehler in den Teilmodellen korrigiert werden oder die Teilmodelle erweitert werden, ohne dass die Änderung in jedem Prozess durchgeführt werden muss. Die entstehenden Modelle werden übersichtlicher. Auf der anderen Seite steigt die Komplexität der Modellierung durch die entstehende Hierarchie zwischen den Teilmodellen. Durch Verwendung von Untermodellen wird der Modellierung eine zusätzliche Dimension hinzugefügt, so dass auch in die Tiefe modelliert werden kann. Der Text der Benachrichtigung wird dem Untermodell als Modelparameter übergeben.

Viele Prozesse übertragen zu Beginn die benötigten Daten von der *IGM-Plattform* in ihre lokale Datenbasis im *jABC*. Da hierfür immer die gleichen *DTOs* verwendet werden bietet sich dieser Teilprozess ebenfalls an um in einem Untermodell umgesetzt zu werden.

11.3 Nebenläufigkeit von Teilprozessen

Teilprozesse, die unabhängig voneinander sind, können nebenläufig modelliert werden. Das *jABC* unterstützt die Nebenläufigkeit von Prozessen auf verschiedene Weisen. Ein Prozess kann mittels eines *Fork-SIB* in zwei unabhängige Teilprozesse aufgespalten werden, die dann mittels eines *Join-SIB* zusammengeführt und synchronisiert werden. Mit einem Teilprozess in einem *Thread-SIB* kann dieser Teil unabhängig von dem Elternprozess ausgeführt werden, aber der Elternprozess wartet nicht auf das Ergebnis des Kindprozesses.

Bei einer Einbruchsmeldung sollen außer dem Bewohner auch der beauftragte Sicherheitsdienst benachrichtigt werden. Diese Benachrichtigungen sind unabhängig voneinander. Der Prozess wird also, nachdem die Informationen die für beide Prozesse benötigt werden, übertragen wurden, mittels *Fork* aufgespalten. Die beiden Teilprozesse führen ihre Aktivitäten aus und werden wieder zusammengeführt. Erst wenn beide Teilprozesse beendet sind, wird der Gesamtprozess fortgeführt. Die Ergebnisse aus den Teilprozessen können ausgewertet werden. Konnte nur der Bewohner erreicht werden, kann dies also z.B. bei der Abrechnung berücksichtigt werden, ohne dass beide Teilprozesse eine Abrechnung durchführen müssen. Dieses Konzept ist auch auf weitere Prozesse anwendbar.

In anderen Fällen ist es eventuell notwendig mehre Bewohner oder Dienstleister zu benachrichtigen. Die Teilprozess können also auch weiter parallelisiert werden. In Verbindung mit der in 11.2 erläuterten Verwendung von *Graph-SIBs* kann in einem Untermodell zur Dienstleisterbenachrichtigung auch ein Modell eingeführt werden, dass entweder alle hierfür konfigurierten Dienstleister benachrichtigt werden, oder nachdem der erste konfigurierte Dienstleister erreicht wurde, wird abgebrochen. Der Parameter für die Entscheidung welche Benachrichtigungsvariante gewählt wird kann als Modellparameter an den Elternprozess weitergereicht werden. Auf diese Weise kann bei der Modellierung entschieden werden welche Variante gewählt werden soll. Für den Teilprozess der Bewohnerbenachrichtigung ist ein analoges Vorgehen möglich, falls außer dem Bewohner noch Nachbarn oder Angehörige benachrichtigt werden sollen.

Teilprozesse, deren Ergebnis den Prozessverlauf nicht beeinflusst, können mittels eines *Thread-SIBs* abgespalten werden. Soll die Wohnungsverwaltung bei Notfällen, wie einer Wasserleckage benachrichtigt werden, beeinflusst dies den Gesamtprozess nicht. Durch die Parallelisierung behindert dies die Ausführung des Gesamtprozesses nicht.

11.4 Ausnahmebehandlung in Prozessen

Grundsätzlich wird im Rahmen der Glättung von Prozessen eine standardisiert Ausnahmebehandlung eingeführt. Diese muss auf Basis des Fachwissens des Modellierers erstellt werden. Das *jABC* als grafisches Modellierungstool stellt dem Modellierer ein mächtiges Werkzeug bereit. Nachdem ein Prozess modelliert ist müssen die Stellen im Ablauf erkannt werden, die fachlich potentiell fehlerträchtig sind.

Die Ausnahmebehandlung kann aber auch zwischen zwei unterschiedlichen Teilprozessen nötig sein. Die Existenz eines anderen Prozesses, aber auch der Zustand eines anderen Prozesses kann für die Fortführung eines Prozesses entscheidend sein. Das prinzipielle Vorgehen hierbei wird an dem Brand- und Gasprozess dargestellt. Tritt in einer Wohnung eine Gasleckage und ein Brand auf, so soll eine Ausnahmebehandlung stattfinden. Diese Ausnahmebehandlung muss in beiden Prozessen stattfinden.

Zu Beginn der Projektgruppe wurde hierfür durch die Priorisierung und das dynamische Abonement von Events eine wichtige Grundlage geschaffen. Als Reaktion auf einen Event wird der zuständige Prozess gestartet, also der Prozess der Brandbehandlung wird von einem Brand-Event gestartet, der Prozess der Gasleckage von einem Gas-Event. Der Brandprozess muss also zuerst die Gas-Events abonnieren. Der Gasleckageprozess wird aber trotzdem bei einem Gas-Event gestartet. Der Brandprozess abonniert das Gas-Event mit der Priorität HIGH. Die Prozessausführung wird beim Auftreten des Events abgebrochen und ein Ausnahmeprozess wird gestartet. Hierfür muss in dem *SIB*, das das Gas-Event abonniert, das entsprechende *Start-SIB* angegeben werden.
In der Ausnahmebehandlung wird der Dienstleister, der wegen dem Brand alarmiert wird (meist die Feuerwehr), benachrichtigt, dass in der Wohnung eine Gasleckage aufgetreten ist. Dabei muss natürlich zuerst geprüft werden, ob die Gasleckage in der gleichen Wohnung aufgetreten ist. Eine Überprüfung der Nachbarwohnungen ist momentan aber nicht ohne weiteres möglich. Auf diese Weise könnte auch eine Gasleckage in einer Nachbarwohnung zu einer zusätzlichen Benachrichtigung der Dienstleister führen. Nach der Ausnahmebehandlung kann der Prozess an der Stelle fortgeführt werden an der er unterbrochen wurde, beendet werden, neugestartet werden oder bei einem beliebigen *SIB* in dem Prozess weitergeführt werden.

Der Gasprozess überprüft ob ein Brandprozess existiert, der in seinem Kontext die gleiche domicileID hat. Hierfür wurde ein *SIB* erstellt, das überprüft, ob ein bestimmter Prozesstyp existiert und dieser als zusätzliche Bedingung einen angegebenen Wert im Kontext hat. Existiert kein solcher Prozess wird der Gasprozess normal ausgeführt. Wird zur Ausführungszeit des Gasprozesses ein Brandprozess in der gleichen Wohnung erkannt, so wird der normale Prozessablauf nicht weitergeführt, sondern eine Ausnahmebehandlung durchgeführt. Der Bewohner wird informiert, dass er die Wohnung nicht betreten darf. Der Dienstleister, der die Gasleckage beheben soll wird nicht informiert. Damit soll verhindert werden, dass ein Mitarbeiter in eine explosionsgefährdete Wohnung geschickt wird. Die Warnung der Feuerwehr wird durch den Brandprozess vorgenommen.

Die Ausnahemebehandlung innerhalb von Prozessen ist ein vielschichtiges Problem und benötigt das Fachwissen des Modellierers über die Prozesse. Auf welche Weise eine Ausnahmebehandlung stattfinden soll ist aber außerdem eine politische Entscheidung des Betreibers. Hier wurde exemplarisch gezeigt welche Möglichkeiten das System aus *jABC* und *IGM-Plattform* zur Ausnahmebehandlung bietet.

11.5 Unterstützung der Prozessausführung durch ECA-Regelsysteme

Das bisherige Vorgehen der Projektgruppe ist es, für die *IGM-Plattform* spezifizierte Abläufe der *SmarterWohnen*-Dienstleistungen in eine prozessorientierte Sicht zu überführen und auf dem verteilten System bestehend aus *jABCProzessManager* und *IGM-Plattform* zu implementieren. Dabei wird der Ablauf der Dienste komplett dem Ausführungsbereich der *IGM-Plattform* entzogen und *jABC*-seitig implementiert. Lediglich Kontextinformationen und daraus resultierende Ereignisgenerierung sowie Kommunikationsdienste verbleiben auf der *IGM-Plattform* und werden aus dem *jABC* heraus genutzt.

Neben dem Business-Process-Management-Subsystem (*BPM-Subsystem*) steht auf der *IGM-Plattform* jedoch außerdem das *Evaluation-Subsystem* (*EVA-Subsystem*) zur Ablaufsteuerung und somit zur Abbildung von Teilen der Prozesslogik zur Verfügung. Das *EVA-Subsystem* stellt eine Ausführungsumgebung für *ec-jobs* bereit, ein IGM-spezifisches Derivat zur Definition von ECA-Regeln. Das Subsystem ermöglicht es somit, Teilfunktionen zur Realisierung der *SmarterWohnen*-Dienste als ECA-Regelsystem zu implementieren und die Implementation der Dienste auf beide Systeme verteilen zu können. Die Kommunikation beider Systeme erfolgt dann naturgemäß mit Hilfe von Ereignissen.

Bei der zukünftigen Entwicklung und Erweiterung von Dienstleistungen des *SmarterWohnen*-Projektes wird die Realisierung der Dienste nicht ausschließlich durch ein Business-Process-Subsystem vorgenommen werden, sondern zusätzlich durch das EVA-Subsystem. Welches System die Ausführung des Dienstes übernimmt wird dabei hauptsächlich von Art des Dienstes abhängen. Bei Dienstkomponenten der Struktur eines Geschäftsprozesses eignet sich die Implementierung auf dem BPM-Subsystem, während bei Ereignis basierten Komponenten eine Implementierung mit Hilfe von *ec-jobs* innerhalb des EVA-Subsystems vorzuziehen ist. Von Seiten des Fraunhofer ISST ist die Evaluierung einer solchen Verteilung ausdrücklich erwünscht und wird daher innerhalb dieses Abschnitts anhand einer Diensterweiterung durchgeführt.

11.5.1 Realisierung

Die Prozesse Brand-, Gasleckage- und Einbruchmeldung ähneln sich in Ihrer Ablauflogik sehr stark. Insgesamt umfassen alle Prozesse einen Alarmplan und eine Beendigungsprozedur. Nach Bekanntwerden des zentralen Alarmereignisses wird zunächst der Alarmplan abgearbeitet, in dem hauptsächlich die Benachrichtigung von Dienstleister und Bewohner durchgeführt wird. Die Beendigungsprozedur sieht einen Abschluss des gesamten Alarmprozesses vor, bei dem ebenfalls eine Benachrichtigung über das Alarmende an die beteiligten Akteure Dienstleister und Bewohner verschickt wird.

Darüber hinausgehende Maßnahmen, wie die zusätzliche Benachrichtigung von Nachbarn zum Zweck der Alarmierung, sind allerdings bisher weder auf der *IGM-Plattform* des ISST noch auf dem System der Projektgruppe vorgesehen. Daher stellt die Modellierung

und Implementierung einer solchen Funktionalität ein modellhaftes Anwendungsszenario zur Umsetzung der Kombination beider Systeme dar.

Bei den genannten Alarmszenarien bietet die Alarmierung von Nachbarn im Fall eines Brandes, einer Gasleckage sowie eines Einbruchs zusätzliche Sicherheit, um die Bewohner auf das Ereignis aufmerksam zu machen und somit die Dienstleistung insgesamt sicherer zu gestalten. Es kann zudem eine Integration von Kontextinformationen hinzugezogen werden, sodass auch tatsächlich nur solche Nachbarn über einen Alarm benachrichtigt werden, die sich zum Zeitpunkt des Vorfalls im Gebäude aufhalten. Das Verlassenszenario stellt der Plattform genau diese Information zur Verfügung.

Die Evakuierungsfunktion wird somit durch *Boolean EC-jobs* innerhalb des *EVA-Subsystems* auf der IGM-Plattform realisiert. Da der Kontrollfluss der Alarmverarbeitung für die Notfallarten Brand, Gasleckage, Wasserleckage und Einbruch durch die *jABCProzessPlattform* gesteuert wird, soll auch dort eine Evakuierung in Abhängigkeit vom Prozessmodell ausgelöst werden können. Dem jABC-Modellierer wird ein Evakuierungs-SIB zur Verfügung gestellt, mit dem er die Evakuierung initiieren kann. Das Evakuierungs-SIB ist ein SendEvent-SIB in Verbindung mit dem entsprechenden DTO. Das Evakuierungs-SIB veranlasst also den Versand eines Evakuierungs-Events an die IGM-Plattform. Dieses Event enthält folgende Informationen gemäß der vereinbarten Parametertaxonomie:
`Domicile.domicileID`
`Domicile.Emergency.component`
`Domicile.Emergency.room`
`Domicile.Emergency.emergencyType`
`transactionID`
Es wird somit innerhalb des Events der Wohnungsidentifikator der Wohnung, in der der Alarm ausgelöst wurde, übermittelt. Weiterhin werden Informationen zum meldenden Sensor, dessen Standort sowie die Art des Alarmes und eine Transaktionskennung versendet. Auf der IGM-Plattform wird mit Hilfe von *Boolean EC-jobs* die eigentliche Evakuierung durchgeführt. Da die Implementierung des *EVA-Subsystems* nur keine ECA-Regelkaskaden zulässt, ist der gesamte Evakuierungsprozess von nur einem einzigen *Boolean EC-job* durchzuführen. Dieser Job abonniert das oben genannte Evakuierungsevent und führt daraufhin seinen nachfolgend beschriebenen Aktionsblock aus:

1. Es werden zunächst Adressinformationen zu der Wohnung ermittelt, in der der Notfall aufgetreten ist. Der Identifikator dieser Wohnung wird durch das Evakuierungs-Event übermittelt und kann somit hierzu verwendet werden. Der Straßenname, die Hausnummer, die Postleitzahl und der Ort der Wohnung stehen somit zur weiteren Bearbeitung der Evakuierung zur Verfügung.
2. Es soll nun die Menge der Wohnungen bestimmt werden, die in Bezug auf den vorliegenden Notfall mit der Notfallwohnung benachbart sind. Die IGM-Plattform stellt jedoch nur die in Punkt 1 ermittelten Adressinformationen zur impliziten Bestimmung einer Nachbarrelation zwischen Wohungen zur Verfügung. Daher wird nun anhand dieser Informationen die Menge der Wohnungen bestimmt, denen dieselbe Adresse zugewiesen wurde wie der Wohnung, in welcher der Notfall vorliegt. Leider ist eine solche Lösung fehlerbehaftet, da beispielsweise unterschiedliche Schreibweisen desselben Straßennamens dazu führen, dass solche Wohnungen nicht als benachbart erkannt werden.

3. Für jede der in der ermittelten Menge enthaltenen Wohnungen wird nun der zugehörige Bewohner bestimmt und diesem anschließend eine Nachricht mit Informationen zum benachbarten Notfall versendet.

12 Evaluierung

12.1 Fremdsysteme

12.1.1 IGM-Plattform

Eines der Ziele, das im Rahmen der Projektgruppe festgelegt wurde, sollte der Vergleich von Prozessimplementierungen durch ECA-Regelsysteme und graphenorientierte Prozessbeschreibungen sein (vgl. Abschnitt I). Auf dieser Basis wurde davon ausgegangen, dass die Möglichkeit besteht auf der *IGM-Plattform* durch das *EVA-Subsystem* komplexe ECA-Regelsysteme bzw. ECA-Kaskaden auszuführen. Eine Anwendungsmöglichkeit für eine solche Implementierung wird in Kapitel 11.5 erläutert. Leider war es nicht möglich eine Implementierung in Form von ECA-Regelketten umzusetzen, da eine Unterstützung für ECA-Regelsysteme im *EVA-Subsystem*, in der ursprünglichen Form, nicht unterstützt wird. Bisher ist es nur möglich einzelne ECA-Regeln in Form von *EC-Jobs* auszuführen. Die Erzeugung eines neuen Events in deren Aktionsteil wird derzeit aber nicht unterstützt. Eine Umsetzung komplexer Regelketten würde eine sinnvolle Erweiterung darstellen, durch die vor allem die hohe Komplexität bzw. der teilweise große Umfang von Aktions-Teilen in vorhandenen ECA-Regeln (*EC-Jobs*) vermindert werden könnte.

Die teils recht umfangreichen Aktions-Teile in den ECA-Regeln bzw. die teils komplexen Abfragen von Daten in den *DataMappern* des *BPM-Subsystems*, sind aber auch der Tatsache geschuldet, dass Daten aus dem *User-Subsystem* und dem *Context-Subsystem* zusammengeführt werden müssen. Die Trennung der zwei Systeme erscheint nach den Erläuterungen in [59] (Kapitel 1.2.3, S. 15f) und dort zitierter Literatur sinnvoll, erfordert in der Praxis aber häufig eine aufwendigere Konsolidierung bzw. Zusammenführung der benötigten Daten aus beiden Systemen. Innerhalb der Projektgruppe wurde daher einige male darüber diskutiert, ob eine Ermittlung der Daten auf Basis von reinen SQL-Statements mit direktem Datenbankzugriff an diesen Stellen nicht sinnvoller sei. Dies ist allerdings zu verneinen, da dies zum einen den Ansatz der serviceorientierten Architektur zerstören und zum anderen die Kapselung der Daten durch die Subsysteme umgehen würde. Der Ansatz, Daten über *Profile*(vgl. Listing 20) zu ermitteln, ist zwar mit verantwortlich für die aufwendigere Ermittlung von Daten, trägt aber maßgeblich zur Erhaltung der zuletzt genannten Aspekte bei.

Listing 20: Ermittlung von Daten anhand von Profilen

```
// read main occupant of domicile with id "0000 0 0000" from
   user subsystem

// create profile for entity "bewohner"
Profile query = new Profile("class", "bewohner");

// add data to identify domicile
query.Add("wohneinheit_id", "0000 0 0000");
query.Add("ist_hauptbewohner", "True");

// data to receive (user data)
```

```
query.AddGet("bewohner_id");
query.AddGet("name");
query.AddGet("vorname");
query.AddGet("geschlecht");

// get user profiles matching query
UserSubsystem us = new UserSubsystem();
Profile[] result = us.GetUserProfile(null, "AppName", query);
```

Diese Abstraktion von der Datenhaltung in Form einer Datenbank fehlt momentan leider noch, wenn Daten in die Datenbank geschrieben werden sollen. Gemeint ist, dass bspw. der Kontext für eine Entität geändert werden soll. In den von uns analysierten Szenarien wäre dies z.B. der Fall, wenn Sensoren den Kontext ändern möchten. Für die Implementierung ist es derzeit noch notwendig, die von einem Sensor gelieferten Daten, direkt in die Tabellen der Datenbank zu schreiben. Dies wurde während mehrerer Projektgruppen-Sitzungen als Schwäche der Architektur angesehen. Eigentlich wurde erwartet, dass die Änderung von Daten bzw. des Kontextes auch direkt über die einzelnen Subsysteme gemacht wird. Dies hätte den Vorteil, dass ein Polling auf der Datenbank, wie es bspw. im *Context-Subsystem* stattfindet, um nach Änderungen zu suchen, entfallen würde. Ein Subsystem würde eine Änderung der Daten in diesem Fall direkt mitbekommen. Auch der Einsatz eines "Active Database"-Systems (vgl. [35] Kapitel 2) wurde mehrfach angesprochen, um hier Abhilfe zu schaffen.
Wenn man allerdings die Architektur des *Context-Subsystems*, wie sie in [59] (Abb. 10) dargestellt ist, mit der derzeitigen Implementierung vergleicht, wird klar, dass dieses Verhalten dadurch entsteht, dass derzeit nur die Schichten "Context Model Query", "Context Model Management" und "Context Entity" in Teilen implementiert sind. Die zum Schreiben von Daten benötigten (tieferen) Schichten sind derzeit nicht in Form einer Implementierung vorhanden. Dies ist aktuell Thema einiger Diplomarbeiten. Die Kapselung der schreibenden Datenzugriffe ist daher konzeptionell vorgesehen und an einer Implementierung wird gearbeitet.

Im Rahmen der Implementierung der *SmarterWohnen*-Prozesse auf der *IGM-Platform*, ergab sich häufig das Problem, das nicht klar war, wie auf vorhandene Daten konkret zugegriffen werden kann. Häufig war nicht genau beschrieben, welche Felder in der Datenbank für welchen Zweck vorgesehen waren bzw. welche Felder in einigen Szenarien überhaupt Relevanz besaßen. Auch die Verknüfpung der vorhandenen Daten mit den Subsystemen bzw. deren Eingliederung in diese war teils undurchsichtig. Hier hätte eine Dokumentation des Datenbankschemas Abhilfe schaffen können. Dies hätte auch dazu beitgetragen die teils recht unklar definierten Beziehungen zwischen Datensätzen in der *SmarterWohnen* -Datenbank und deren Bedeutung zu erläutern. Damit es hier nicht zu Missverständnissen kommt, soll an dieser Stelle festgehalten werden, dass dies aber ein Problem der Umsetzung des *SmarterWohnen*-Projektes auf der Plattform und kein Problem der Plattform an sich ist.
Auch eine umfangreichere Dokumentation für die eigentliche Plattform wäre insoweit wünschenswert gewesen, dass vor allem Beispiele für die Implementierung gegen die vorhandenen Schnittstellen hilfreich gewesen wären. Zwar existieren sehr umfangreiche Konzepte, aber die eigentliche Implementierung ist derzeit nur wenig dokumentiert. Eine Dokumentation wäre aber gerade in Bezug auf die Erweiterbarkeit und Anbindung des *BPM-Subsystems* wünschenswert gewesen. Dokumentation in Form eines Entwicklungsleitfadens oder Handbuches, in dem die, zum Großteil in den Sourcen vorhandene, API-Dokumentation Erwähnung finden könnte, wäre darüber hinaus hilfreich gewesen und

hätte das Durchsuchen von Code an der einen oder anderen Stelle vermeiden können.
Die Realisierung der Anbindung des *BPM-Subsystems* stellte sich allerdings dennoch verhätnismäßig einfach dar, da ausreichend Unsterstützung durch die Mitarbeiter am Fraunhofer ISST vorhanden war. Auch die Schnittstellen der einzelnen Subsystem erwießen sich in der Regel als sehr robust. Vor allem die Kernfunktionalität zum Eventhandling und zum Datenaustausch mit den anderen Systemen funktionierten tadellos.

Die *IGM-Plattform* stellt daher ein solide Implementierung einer serviceorientierten Architektur zur Prozessausführung mit Hilfe von ECA-Regeln dar. Vor allem konzeptionell ist einige Arbeit in die Planung der Plattform geflossen, die sich auch in der Implementierung wieder findet. Die Konzepte sind zwar noch nicht in allen Punkten durch die Implementierung abgedeckt, das wäre aber auch von einer, sich in der Entwicklung befindlichen Plattform, nicht zu erwarten gewesen.

12.1.2 jABC

Wie schon erwähnt, wird das *jABC* genutzt, um die *SmarterWohnen*-Prozesse grafisch abzubilden und auszuführen.

Konzept Die Trennung des Entwicklungsprozesses in eine fachliche (Graphenmodellierung) und eine technische Sicht (*SIB*-Entwicklung) erleichterte die Partnerarbeit ehrheblich. So war es problemlos möglich, zeitlich sowie örtlich getrennt an beiden Sichten zu arbeiten, ohne auf fehlende Teile warten zu müssen. Das hierarchische Schachteln von Prozessen erhöht die Übersichtlichkeit bei der Modellierung.

Usability Die Bedienung des *jABC* ist teilweise sehr komplex und undurchschaubar. Viele Funktionen und Tastenkürzel sind undokumentiert, so dass man sie erst nach einer längeren Eingewöhnung findet (Beispiel: Nutzung der *jABC-Taste*). Außerdem sind die Funktionen nicht einheitlich über die Menüs und Kontextmenü verteilt, was oft verwirrend ist. Geänderte *SIBs* müssen einzeln per Hand in den Graphen ersetzt werden, außerdem fehlt dem *SIBCreator* die Möglichkeit, `ELExpressions` direkt zu erzeugen.

Dokumentation und Hilfe Die Dokumentation zur Nutzung des *jABC* sowie die Entwicklerdokumentation für Erweiterungen ist noch nicht in allen Bereichen ausführlich genug, oft ist dadurch ein zeitaufwändiges Ausprobieren nötig. Außerdem wurde dadurch die Einarbeitungsphase deutlich verlängert.
Positiv hervorzuheben ist die Unterstützung durch das *jABC*-Entwicklerteam, die uns oft sofort weiterhelfen konnten. Da sich das *jABC* noch in der Weiterentwicklung befindet, wurden viele angemerkte Punkte direkt aufgenommen und umgesetzt.

Technische Details Hier einige technische Details, die uns aufgefallen sind:

- Im *jABC* existiert eine *Undo*-Funktion
- Erzeugte Graphen sind optisch sehr ansprechend
- *SIBs* lassen sich auch nachträglich generalisieren
- Es ist nur eine Prozessinstanz ausführbar, ein *Deploying* wäre wünschenswert
- die JGraph-Bibliothek hat Anzeigeprobleme

12.2 Eigene Systeme

12.2.1 BPM-Subsystem

Das *BPM-Subsystem* stellt wie in Kapitel 9.1 beschrieben einen Proxy für das *jABC* bzw. die Schnittstelle zwischen *jABC* und *IGM-Plattform* dar.
Die Implementierung ist prototypischer Art, allerdings besteht aufgrund des serviceorientierten Aufbaus die Möglichkeit alle Komponenten einzeln auszutauschen. Dies wird vor allem durch die für alle Komponenten definierten Interfaces unterstützt, durch die auch eine klare Abgrenzung aller Komponenten gegeben ist.
Die Implementierungen der einzelnen Komponenten sind vollständig getestet worden (vgl. Kapitel 9.7.2), was sich auch in der Robustheit, vor allem des WebServices, niederschlägt.
Eine Installation und Grundkonfiguration sollte aufgrund des vorhandenen Installations-Paketes einfach für jedermann ohne große Vorkenntnisse möglich sein. Auch die Erweiterung des Systems ist verhältnismäßig einfach. Dazu trägt vor allem die serviceorientierte Arhchitektur bei, aber auch die vollständige Dokumentation der API[42] und das Entwicklerhandbuch[43] in denen die wichtigsten Schritte dokumentiert sind.

Letztendlich handelt es sich aber um einen Prototypen, der zwar alle grundlegenden und geforderten Anforderungen erfüllt, aber sicherlich nicht alle denkbaren Szenarien abdeckt. So sind etliche Erweiterungen und Verbesserungen denkbar. Einige sind in den Kapiteln 10.2 und 10.3 bereits erwähnt. Vor allem, dass für die Implementierung neuer *Events* und *DataTransferObjects* eine Implementierung vorgenommen werden muss, ist recht umständlich. Eine mögliche Erweiterung durch reine Konfiguration (vgl. z.B. Listing 13 auf Seite 106) für alle *Events* und *DataTransferObjects* wäre ein generischerer Ansatz, aber auch nur mit viel Aufwand möglich, da diese immer vom speziellen Einsatzgebiet der Plattform abhängig sind.

12.2.2 jABC-Plattform

Die *jABC-Plattform* stellt wie in Kapitel 9.3 bereits beschrieben eine Prozessverwaltungs- und Prozessausführungsumgebung dar.
Das Hauptaugenmerk lag bei der Entwicklung auf der Trennung von Verwaltungsmechanismen und den fachlich geprägten Prozessen. Implementierungsmaßnahmen sind zur Prozessüberarbeitung oder -neuerstellung nicht erforderlich (vgl. Kapitel 10.5):

- Prozesse können getrennt von der *jABC-Plattform* mit Hilfe des *jABCs* graphisch erstellt oder modifiziert werden.
- Die dort verwendeten *DTOs* sind über XML-Files realisiert.
- Die in den Prozessen eingesetzten *SIBs* können über Parameter konfiguriert werden und lassen sich somit immer wieder benutzen.

Da die Prozesse aber außerhalb der *jABC-Plattform* modelliert werden, sind Wechselwirkungen zwischen parallel ausgeführten Prozessen, nicht ohne *Live*-Tests auf der Plattform, auszuschliessen. Um dieses und andere Probleme schon während der Modellierung vermeiden und testen zu können, sollte das *jABC* um entsprechende Tools und Assistenten (z.B. als Plugins) erweitert werden.
Das Einbinden neuer oder überarbeiteter Prozesse in die laufende *jABC-Plattform* ist nur begrenzt *Hot-Deploy*-fähig (siehe Kapitel 10). Hier besteht also auch noch Verbesserungsbedarf.
Obwohl die *jABC-Plattform* über keinen *richtigen* Installer, wie die *IGM-Plattform* verfügt, geht die Installation, aufgrund vorhandener Build-Skripte, nicht viel schwerer von der Hand .

Alles in allem ist die *jABC-Plattform* noch sehr prototypisch, erfüllt aber, Dank Ausnutzung der umfangreichen Fähigkeiten des *jABCs* (z.B. hierarchische oder auch nebenläufige Prozesse), alle in der Konzeptionierungsphase festgelegten Anforderungen.

13 Evaluierung des Gesamtsystems

Um eine Bewertung des Gesamtsystems vorzunehmen, wird das Zusammenspiel der einzelnen System und nicht die Systeme selbst betrachtet.
Der auffälligste und am Anfang der PG komplizierteste Punkt beim Zusammenspiel war die Implementierung der Webserviceschnittstellen. Obwohl diese in einem Standard definiert sind, interpretieren einige Firmen Details etwas anders, sodass hier Probleme auftraten. So unterstützte die von uns genutzte *.Net Framework*-Umgebung das XML Schema Element `choice` nicht. Hier musste durch das Design des Webservicees eine Umgehung des Problems gefunden werden.
Die von uns gewählte Parametertaxonomie und die erstellten *DTOs* bereiteten ab und an Probleme. Die fachliche Sicht auf die Kommunikation der System wurde zwar deutlich vereinfacht, doch die technische Umsetzung wurde dadurch um so komplizierter. So kam es unter den PG-Teilnehmern zu unterschiedlichen Auffassungen wie die Hashtables, mit deren Hilfe Daten zwischen des Systemen ausgetauscht werden, zu realisieren und befüllen sind. Hier musste die eigene Spezifikation noch einmal aufgegriffen und überarbeitet werden.
Kritisch zu betrachten ist, auch wenn dies nicht die Aufgabe der Projektgruppe war, die fehlende Sicherheit der Systeme. In der von uns genutzten Version der *IGM-Plattform* gibt es keine Rechteverwaltung in Form von Access-Control-List oder Verschlüsselung und Signierung. Auch die Kommunikation zwischen den Systemen verläuft unverschlüsselt und offen. Da das Ziel der Projektgruppe eine prototypische Implementierung werden sollten, wurden diese Punkte von vornherein aus der Arbeit ausgeschlossen. Desweiteren fanden aufgrund fehlender Zeit und Software kein Lasttest des Gesamtsystems statt. Hier hätten noch deutlich mehr konsistente Datensätze erzeugt und Software für automatische Tests entwickelt werden müssen.
Positiv anzumerken ist, dass die Projektgruppe die gestellte Aufgabe voll umsetzen konnte. Es gelang *jABC* mit Hilfe von Web-Schnittstellen in die *IGM-Plattform* zu integrieren. Die entwickelten Monitoringwerkzeuge stellten bei der Suche nach Fehlern eine unverzichtbare Hilfe dar und Dank der entwickelten Installations und Konfigurationsroutinen bedarf eine Installation der System nur wenig Zeit.

Teil VI
Diskussion

14 Vergleich mit anderen Workflow-Management Systemen

14.1 JBoss

14.1.1 Überblick jBoss Middleware

Die *jBoss Middleware* ist eine Open Source Applikation, um webbasierte Anwendungen und Dienste innerhalb einer service-orientierten Architektur zu erstellen, einzubinden, zu orchestrieren und darzustellen.
Die *JBoss Enterprise Application Platform* fasst einen *Java EE* Application Server, Object-Relational-Mapping (O/R-Mapping) und Persistenz zu einer Plattform zusammen. Die *JBoss Enterprise Portal Platform* stellt die angebotenen Dienste in einer personalisierbaren Oberfläche dar. Die Plattformen beinhalten verschiedene Frameworks. Das *jBoss Hibernate Framework* setzt das O/R-Mapping und die Persistenzfunktionalität um, während *JBoss Seam* ein Framework zur Entwicklung von Web 2.0 Anwendungen ist. Das Erstellen und Ausführen von Geschäftsprozessen wird durch das *jBoss jBPM Framework* unterstützt. Regelbasierte Systeme werden mit Hilfe des *jBoss Rules Frameworks* umgesetzt.

14.1.2 jBoss jBPM

Das *jBPM Framework* soll bei der Orchestrierung von *Enterprise Applications* unterstützen. Die *jBPM-Engine* kann als Web-Applikation oder als Standalone-Anwendung eingesetzt werden. Mit dem Framework werden drei Ziele verfolgt: (1) Als Modellierungswerkzeug für Workflows und Geschäftsprozesse; (2) Zur Erstellung von prozessbasierten Anwendungen (insbesondere als Ergänzung zu ERP-Systemen); und (3) Als unterstützende Komponente in der IT-Architektur von Unternehmen.

Innerhalb des *jBPM* sind drei zentrale Elemente spezifiziert:

1. Die ***Process Engine*** überwacht den Status und alle Ein- und Ausgabevariablen der aktiven Prozesse.
   Dazu setzt sie einen *Request Handler* ein, der die Kommunikationsinfrastruktur bereitstellt und Aufgaben an die verantwortlichen Prozesse, Benutzer oder Anwendungen weiter gibt.
   Interaction Services sind standardisierte oder benutzerdefinierte Dienste, die existierende Anwendungen als Funktion in einen Prozess einbinden können.
   Der *State Manager* überwacht den Zustand von Prozesse, aber auch den Zugriff von Prozessen auf Daten.
2. Der ***Process Monitor*** erlaubt die visuelle Überwachung von Prozessen über ihre gesamte Laufzeit. Damit gibt er Aufschluss über den Status von in den Prozess eingebundenen Benutzern oder Anwendungen.
3. Die ***Process Language*** basiert auf einem gerichteten Graphen. Die aktuell eingesetzte Sprache JPDL ist eine Erweiterung der Ursprungssprache. Die JPDL spezifiziert durch ein XML-Schema den benötigten Mechanismus um eine Prozessdefinition und die benötigten Dateien in ein Prozessarchiv zu überführen. Auf der Basissprache

können andere Standards, wie BPEL, BPELJ, BPML und BPSS aufsetzen.
Der Prozess in Abbildung 48 wird in die XML-Datei in Listing 21 umgewandelt.

Listing 21: Umsetzung des Auktionspozesses in XML

```
<process-definition>
  <start-state>
    <transition to="auction" />
  </start-state>
  <state name="auction">
    <transition name="auction ends" to="salefork" />
    <transition name="cancel" to="end" />
  </state>
  <fork name="salefork">
    <transition name="shipping" to="send item" />
    <transition name="billing" to="receive money" />
  </fork>
  <state name="send item">
    <transition to="receive item" />
  </state>
  <state name="receive item">
    <transition to="salejoin" />
  </state>
  <state name="receive money">
    <transition to="send money" />
  </state>
  <state name="send money">
    <transition to="salejoin" />
  </state>
  <join name="salejoin">
    <transition to="end" />
  </join>
  <end-state name="end" />
</process-definition>
```

Die Elemente in einem jBPM-Prozess sind die sogenannten Nodes. Es werden sechs verschiedene Nodes unterschieden. (1) Task-nodes repräsentieren eine Aktivität die von einem Menschen ausgeführt werden muss. (2) State-Nodes warten auf ein Signal, um dann den Prozess fortzuführen. (3) Decision-Nodes verzweigen aufgrund einer Entscheidung. (4) Fork-Nodes teilen einen Prozess in parallele Teilprozesse auf, die durch (5) Join-Nodes wieder zusammengeführt und synchronisiert werden. (6) Nodetype-Nodes sind in Java frei programmierbar.
Nodes werden durch *Transitions* verbunden. Innerhalb eines Prozesses können *Actions* aufgrund von Ereignissen ausgelöst werden. *Actions* sind frei in Java programmierbar. Die Ausnahmebehandlung wird durch das Java Exception-Handling umgesetzt.
Über das *jBoss Eclipse* Plugin können Prozesse mit graphischer Unterstützung modelliert werden. Die frei programmierbaren Nodes und Aktivitäten können direkt in *Eclipse* bearbeitet werden.

14.1.3 jBoss Rules

jBoss Rules ist in der Lage Regelsysteme zu interpretieren und auszuführen. Hierzu wird die *Drools Rule Engine* eingesetzt. Diese liest XML-Dateien ein, prüft diese und erzeugt

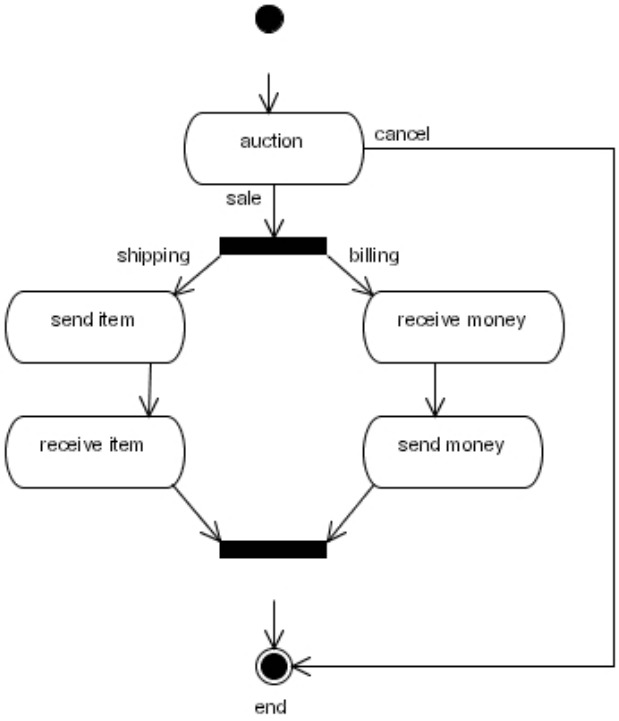


Abbildung 48: Auktionsprozess [*Quelle: jboss.com*]

ein ausführbares Paket.
Eine einzelne Regel hat die einfache Grundstruktur:

```
rule "name"
    ATTRIBUTES
    when
        LHS
    then
        RHS
end
```

LHS (Left-Hand-Side) ist der konditionale Teil der Regel, während der RHS (Right-Hand-Side) der aktive Teil der Regel ist. Sowohl für den LHS als auch für den RHS steht eine große Funktionsbibliothek zu Verfügung. Auch für *jBoss Rules* existiert ein *Eclipse* Plugin.

14.1.4 jBoss im Vergleich

jBoss ist ein Projekt mit einer großen Entwicklergemeinde. Dementsprechend groß ist auch der Funktionsumfang. Die beiden Systeme sind aber durchaus vergleichbar. Beide Systeme arbeiten als Middleware und beherrschen O/R-Mapping. Die Stärken von relationalen Datenbanken können ausgenutzt werden und im Anwendungsbereich kann der objektorientierte Ansatz durchgängig benutzt werden.

Während *jBoss* ein Framework zur integration von Web 2.0 Techniken bereithält wurde in unserer Arbeit auf das GWT (Google Web Toolkit) zurückgegriffen. In unserer Monitoringoberfläche wird die Leistungsfähigkeit von AJAX ausgenutzt.

jBoss Rules und *jBPM* unterstützen das erstellen und ausführen Regelsystemen und Workflows. Die grafische Modellierung von Workflows ist mit *jBoss* als auch mit

der POETS Prozess Plattform möglich. Während *jBoss* ein Eclipse Plugin einsetzt wird in unserer Arbeit das *jABC* eingesetzt. Solange das *jABC* aber nur als grafisches Modellierungswerkzeug benutzt wird, kann es seine Vorteile, wie z.B. den Modelchecker, nicht zeigen. Für die Regelsysteme wird in der *jBoss* Lösung eine eigene Sprache eingeführt, während die ECA-Regeln der *IGM-Plattform* in C# umgesetzt werden. Beide Systeme unterstützen und benutzen Webservices und zur asynchronen Kommunikation Message Queues.

Die Systeme sind sich in ihrem Leistungs– und Funktionsumfang sehr ähnlich. Wobei *jBoss* alle Funktionen und Leistungsmerkmale unserer Plattform unterstützt, aber unsere Plattform nicht alle Merkmale von *jBoss* umsetzt.

14.2 *TriGS$_{flow}$*

14.2.1 Überblick

TriGS$_{flow}$ ist ein vom Fachbereich Informatik der Universität Linz entwickeltes prototypisches Workflow-Management System. Arbeiten zu diesem System wurden im Jahr 1995 veröffentlicht [34]. Obwohl das System aufgrund seines prototypischen Charakters im Gegensatz zu produktiven Workflow-Management Systemen eher als Forschungsprojekt angesehen werden sollte, bietet es dennoch interessante Aspekte in Hinblick auf die auf ECA-Regeln basierende Architektur des Systems. Mit ***TriGS$_{flow}$*** können Geschäftsprozesse wie beispielsweise im jABC in Form von Workflows graphisch modelliert und ausgeführt werden, die darunter liegende technische Realisierung im Bereich der Ausführungsumgebung der Workflows basiert hingegen auf einer Implementierung mittels ECA-Regelsystemen. Weiterhin werden Aktivitäten innerhalb eines Workflows durch die Definition von Rollen mit den ausführenden Agenten lose verknüpft, um die dynamische Anpassbarkeit der Geschäftsprozessmodelle zu gewährleisten. Durch die Verwendung eines objektorientierten Datenbanksystems können sowohl die innerhalb des Workflow-Management Systems in Form von Objekten repräsentierten Entitäten der Anwendungsdomäne persistiert, als auch Datenstrukturen zur Definition und Speicherung der Event-Condition-Action Regelsysteme geschaffen werden.

14.2.2 Aufbau des Systems

Abbildung 49 zeigt die Architektur von ***TriGS$_{flow}$***. Das System basiert auf dem objektorientierten Datenbanksystem *GemStone*, das Datenmodell wird durch die ebenfalls objektorientierte Programmiersprache *OPAL*, einer *Smalltalk80* Variante, implementiert. Die bereits genannte rollenbasierte Zuweisung von Akteuren zu Aktivitäten wird durch das *Roles*-Modul realisiert. Als aktive Komponente zur Bereitstellung von ECA-Regelsystemen kommt *TriGS*[23] zum Einsatz und bietet auf Grundlage von *GemStone*-Mechanismen zur Beobachtung von Objektzuständen. Aufbauend auf die bereits genannten Komponenten stellt ***TriGS$_{flow}$*** Funktionen zum Design von Workflows zur Verfügung und setzt diese Modelle in auf *TriGS* basierende ECA-Regelsysteme um. Das Werkzeug *MasterTOOL* stellt eine graphische Oberfläche zur Modellierung von Workflows bereit und visualisiert laufende Prozesse und verschiedenen Parameter des Systems. Die Applikation *AgentTOOL* hingegen stellt eine Benutzerschnittstelle für die bei der Ausführung von Workflows zur Interaktion mit beteiligten Akteuren notwendige Interaktion bereit.

[23] Akronym für Trigger System for GemStone

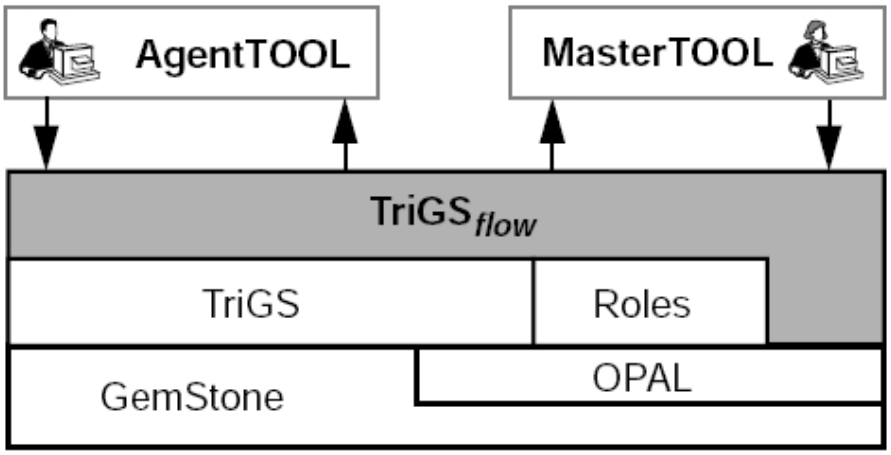


Abbildung 49: Die Architektur von *TriGS$_{flow}$* [34]

14.2.3 Modellierung und Realisierung von Workflows

Mit Hilfe von *MasterTOOL* wird zunächst die Organisationsstruktur der Anwendungsdomäne durch Objekte der Klassen `Department`, `Agent` und `Role` abgebildet. Das hierdurch entstehende Modell kann dann in weiteren Entwicklungsschritten, beispielsweise bei der Modellierung von Workflows, verwendet werden und einzelnen darin definierten Aktivitäten zugewiesen werden. Durch die Spezialisierung des Objekttyps `Agent` in `HumanAgent` und `SoftwareAgent` unterstützt *TriGS$_{flow}$* sowohl menschliche als auch durch Software bereitgestellte Ressourcen zur Ausführung von auf die Durchführung einer bestimmten Aufgabe ausgerichtete Aktivitäten. Hierbei wird das System mit menschlichen Agenten interaktiv agieren, während die Kommunikation mit Softwareagenten im Hintergrund abläuft. Durch die Bereitstellung spezieller Software-Treiber ist es zudem möglich, auf Hardwareressourcen wie beispielsweise Drucker und Faxgeräte als Softwareagent als Ressource zur Aufgabenbearbeitung zuzugreifen. Durch die Zuweisung von Rollen zu Agenten mit Hilfe der Komponente *Role* fließen das kontextabhängige Verhalten und die vom Anwendungsfall abhängigen Kompetenzen der Agenten mit in das Modell ein. Durch in *TriGS* realisierte ECA-Regeln werden zudem Richtlinien zur dynamischen Zuweisung von Agenten zur Verrichtung von Aktivitäten bereitgestellt. Unter Berücksichtigung bestimmter Randbedingungen können auf diese Weise durch die Rollenzuweisung von Agenten in Abhängigkeit vom Aktivitätskontext geeignete Agenten zur Ausführung von Aufgaben ausgewählt werden. Weiterhin wird die Kommunikation zwischen Agenten durch ein Objekt `Folder` realisiert. Jeder laufende Workflow verfügt zur Laufzeit über eine solche Instanz, über die Kontextinformationen zwischen Aktivitäten und den zugewiesenen Agenten ausgetauscht werden können. Vergleicht man Aktivitäten mit den aus dem jABC bekannten SIBs, so ensprechen diese `Folder` dem gemeinsamen Kontext der Ausführungsumgebung im jABC. Mechanismen zum konkurrierenden Zugriff auf diese `Folder` werden durch die darunterliegende Datenbank *GemStone* bereitgestellt. Wird einem Agenten eine Aktivität zur Ausführung zugewiesen, so wird eine neue Instanz der Klasse `WorklistItem` in die von jedem Agenten verwaltete `Worklist` eingetragen. Das Objekt `WorklistItem` enthält Informationen über die mit der Aktivität verbundenen zu verrichtenden Aufgaben.

Innerhalb der Anwendung *MasterTOOL* werden die zu modellierenden Workflows graphisch erstellt. In Abbildung 50 ist ein solcher Workflow schematisch durch einen Graphen von Aktivitäten dargestellt. Der Kontrollfluss wird dabei durch Kanten visualisiert. Die nebenläufige Ausführung von Prozessketten innerhalb eines Workflows wird durch

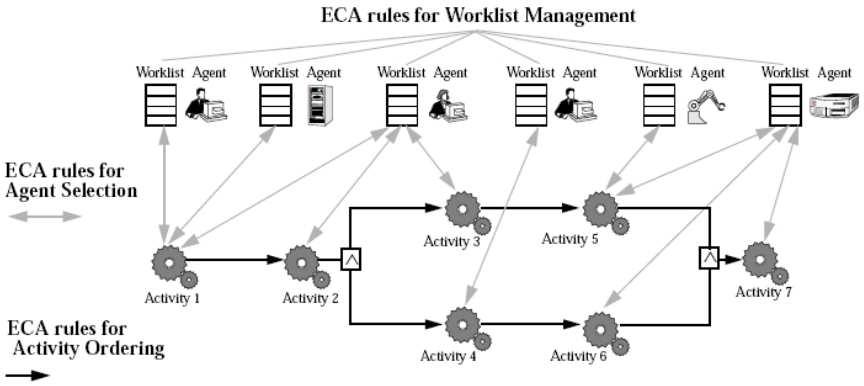


Abbildung 50: Realisierung von Workflows durch ECA-Regeln [34]

spezielle Verknüpfungsoperatoren realisiert, entsprechend den *Fork-* und *Join-SIBs* im jABC. Weiterhin werden für jede Aktivität durch Definition von Anforderungen Informationen zur dynamischen Auswahl von Agenten festgelegt. Das graphisch modellierte und vom System visualisierte Workflowmodell wird im Anschluss transparent vom System $TriGS_{flow}$ in ECA-Regelsysteme übertragen. Diese Regelsysteme werden dabei vor dem modellierenden Benutzer verborgen und dienen der niederschichtigen Realisierung des Workflows. Somit erreicht das System eine zur Modellbildung übersichtlichere und somit geeignetere Sicht als Prozessdarstellung, während es intern auf ECA-Regeln basiert. Dabei wird sowohl die dynamische Zuordnung von Agenten zu Aktivitäten durch ECA-Regeln realisiert, als auch der transitionsbasierte Kontrollfluss der einzelnen Aktivitäten zur Bildung eines Workflows. Weiterhin ist es möglich, durch die direkte Verwendung des in $TriGS_{flow}$ enthaltenen Systems *TriGS* von der Prozesssicht des Systems unabhängige ECA-Regelsystem zu definieren, um diese nativ einsetzen zu können. Als Anwendungsgebiet hierfür ist beispielsweise die Behandlung von innerhalb des normalen Prozessablaufes auftretenden Ausnahmefällen zu nennen.

14.2.4 $TriGS_{flow}$ im Vergleich

Das System $TriGS_{flow}$ stellt eine umfassende Lösung zur Modellierung und Ausführung von Workflows dar. Gerade die rollenbasierte Verknüpfung von Akteuren zu Aktivitäten innerhalb der Geschäftsprozesse ist hilfreich, um die Anwendungsdomäne detailliert in das Modell zu übertragen. Die Persistierung dieses Modells sowie die Speicherung von Informationen zu menschlichen und durch Software implementierte Agenten (Benutzerverwaltung) wird durch das fest mit dem System verankerte Datenbanksystem *GemStone* realisiert. Allein durch die Namensgebung der Komponenten $TriGS_{flow}$ und *TriGS* wird klar, dass es sich hierbei nicht um eine lose Koppelung sondern um ein tief verwurzeltes Konzept handelt. Durch die Abstraktion der Semantik der durch *TriGS* bereitgestellten ECA-Regelsysteme in Form einer Prozesssicht und der dennoch vorhandenen Möglichkeit der nativen Nutzung von ECA-Regelsystemen bietet das System beide von der IGM-Plattform bereitgestellten Konzepte an. Es können somit sowohl Prozesse modelliert als auch ECA-Regeln gesondert eingesetzt werden. Die granulare Zuweisung von Rollen zu Aktivitäten hingegen ist mit Hilfe der IGM-Plattform in Kombination mit der jABCProzessPlattform nicht möglich. Dass es sich bei $TriGS_{flow}$ um einen Prototypen handelt, der laut den verfügbaren Quellen nicht produktiv zum Einsatz gekommen ist, kann jedoch kein direkter Vergleich zwischen beiden Systemen durchgeführt werden. Zudem ist das System nunmehr 12 Jahre alt und verwendet mit *OPAL* und *GemStone* heutzutage nicht mehr eingesetzte Komponenten. Dennoch stellt gerade die Kombination aus ECA-Regelsystemen

und prozessbasierter Realisierung von Workflows in $TriGS_{flow}$ einen maßgeblichen Grund dar, weshalb das System in dieser Projektgruppe betrachtet wurde.

14.3 Windows Workflow Foundation (WF)

14.3.1 Beschreibung des Systems

Die als *Windows Workflow Foundation (WF)* bezeichnete Technologie ist ein Bestandteil des von Microsoft entwickelten *.NET Frameworks* in der Version 3.0. Es erweitert dabei das *.NET Framework* um ein Workflow unterstützendes Entwicklungs- und Programmiermodell und bietet somit Funktionalitäten zur Definition, Ausführung und Verwaltung von als Arbeitsfluss identifizierten Programmkomponenten. Es handelt sich daher nicht um ein Workflow-Management System, sondern um ein Framework zur Entwicklung von Workflow-basierten Systemen. Durch die separate Möglichkeit zur Definition der Ablauflogik eines Workflow einerseits und die Implementierung einzelner als *Activities* bezeichneten Arbeitsschritte andererseits ermöglicht das Framework die isolierte Entwicklung von Ausführungskomponenten unabhängig von der Entwicklung des eigentlichen Arbeitsflusses. Die *Activities* entsprechen hierbei im direkten Vergleich mit dem jABC den dort verwendeten SIBs.

Die Ausführungsumgebung Als Ausführungsumgebung dient der *Windows Workflow Foundation* die *Workflow Runtime*, die in jeder Common Language Runtime Anwendungsdomäne ausgeführt werden kann. Somit ist diese Ausführungsumgebung in Diensten, GUI- und Web-Applikationen sowie als Konsolenanwendungen verfügbar und kann dort zur Ausführung von Workflows zum Einsatz kommen. Die Kommunikation zwischen mehreren Workflows, zwischen Workflows und externen Applikationen sowie zwischen Ausführungsumgebung und den darin laufenden Workflows erfolgt ereignisbasiert. Die Laufzeitumgebung bietet weiterhin Funktionen zur Fehlerbehandlung und implementiert einen als *Passivisation* bezeichneten Mechanismus zur serialisierten Persistierung von untätigen Workflows zum Ressourcenmanagement. Weiterhin werden automatische und vom Entwickler definierte Roll-Back-Mechanismen zur Ausnahmenbehandlung bereitgestellt.

Entwicklung von Workflows Die Ablaufstruktur eines Workflows kann sowohl mit graphischer Unterstützung als auch codebasiert erfolgen. Zur graphischen Modellierung steht der *Visual Workflow Designer* innerhalb der Entwicklungsumgebung *Visual Studio 2005* zur Verfügung, mit dessen Hilfe einzelne Arbeitsschritte in einer Graphdarstellung zu einem Workflow zusammengefasst werden. Hierbei können die aus imperativen Programmiermodellen bekannten Kontrollstrukturen wie z. B. Sequenz, bedingte Ausführung sowie Schleifen verwendet werden. Außerdem kann die nebenläufige Ausführung von Arbeitsschritten implementiert werden. Eine Besonderheit dabei ist, dass die graphische Darstellung und somit auch die logische Sicht auf das Workflow-Modell zwar als Transitionssystem erfolgt, die oben genannten Kontrollstrukturen allerdings ihrerseits zusammengesetzte Arbeitsschritte (*composite activities*) darstellen. Beispielsweise die Sequenz einzelner Arbeitsschritte entspricht einem Sequenzarbeitsschritt (*sequence activity)*, der wiederum aus der sequentiellen Komposition mehrerer Arbeitsschritte zusammengesetzt ist. Die logische Sicht auf ein derart definiertes System stellt dennoch ein Transitionssystem dar, das graphisch modelliert werden kann.

Erfolgt die Modellierung hingegen direkt codebasiert, so steht einerseits die Sprache *XAML*[24] (*Extensible Application Markup Language*) zur Implementierung des Work-

[24] vgl. [2]

flows zur Verfügung. Weiterhin kann jede in *.NET* verfügbare Sprache wie beispielsweise *C#* oder *Visual Basic* zur Workflow-Definition verwendet werden. In Listing 22 ist eine XAML-Definition eines Beispielsworkflows dargestellt. Die graphische Repräsentation dieses Workflows ist in Abbildung 51 dargestellt. Der Workflow implementiert die bedingte Ausführung von drei Arbeitsschritten in Abhängigkeit eines Booleschen Parameters. Die Arbeitsschritte realisieren dabei lediglich die Ausgabe eines Strings.

Listing 22: Implementierung eines Workflows in *XAML* [2]

```
<!-- MyWorkflow.xaml -->
<SequenceActivity
x:Class="MyNamespace.MyWorkflow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/workflow"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:my="http://schemas.example.org/MyStuff"
>
    <IfElseActivity>
        <IfElseBranchActivity>
            <IfElseBranchActivity.Condition>
                <CodeCondition Condition="Is05"/>
            </IfElseBranchActivity.Condition>
            <my:WriteLine Text="Circa-Whidbey"/>
        </IfElseBranchActivity>
        <IfElseBranchActivity>
            <IfElseBranchActivity.Condition>
                <CodeCondition Condition="Is06"/>
            </IfElseBranchActivity.Condition>
            <my:WriteLine Text="Circa-Vista"/>
        </IfElseBranchActivity>
        <IfElseBranchActivity>
            <my:WriteLine Text="Unknown Era"/>
        </IfElseBranchActivity>
    </IfElseActivity>
</SequenceActivity>
```

Activities Ein Workflow setzt sich aus verschiedenen Arbeitsschritten zusammen, deren Ausführung sich anhand eines Kontrollflusses orientieren. Die Möglichkeiten zur Definition dieser Workflows wurden im vorherigen Abschnitt zusammenfassend dargestellt. Demgegenüber wird jeder Arbeitsschritt in der *Windows Workflow Foundation* als *Activity* realisiert. Es werden hierbei *Atomic Activities* und *Composite Activities* unterschieden. Letztere dienen der Komposition von zusammengesetzen und atomaren *Activities* anhand einer zuvor definierten Semantik. Die im Framework bereits vorhandenen *Composite Activities*, wie sequentielle und parallele Komposition sowie bedingte Ausführung und die Realisierung von Schleifen, können dabei um vom Entwickler selbst implementierte Konstrukte erweitert werden. Außerdem müssen atomare Arbeitsschritte mit dem aus der Anwendungsdomäne repräsentierten Pendant identifiziert und deren zu realisierende Funktionalität implementiert werden. Hierzu stehen alle im *.NET Framework* existierenden Programmiersprachen zur Verfügung. Auch alle dort vorhandenen APIs, beispielsweise zur Realisierung eines Datenbankzugriffs, können verwendet werden.

Um die Kommunikation zwischen Ausführungsumgebung und Aktivität gewährleisten zu können, erfolgt die Implementierung einer *Activity* streng nach vorgegebenem Mus-

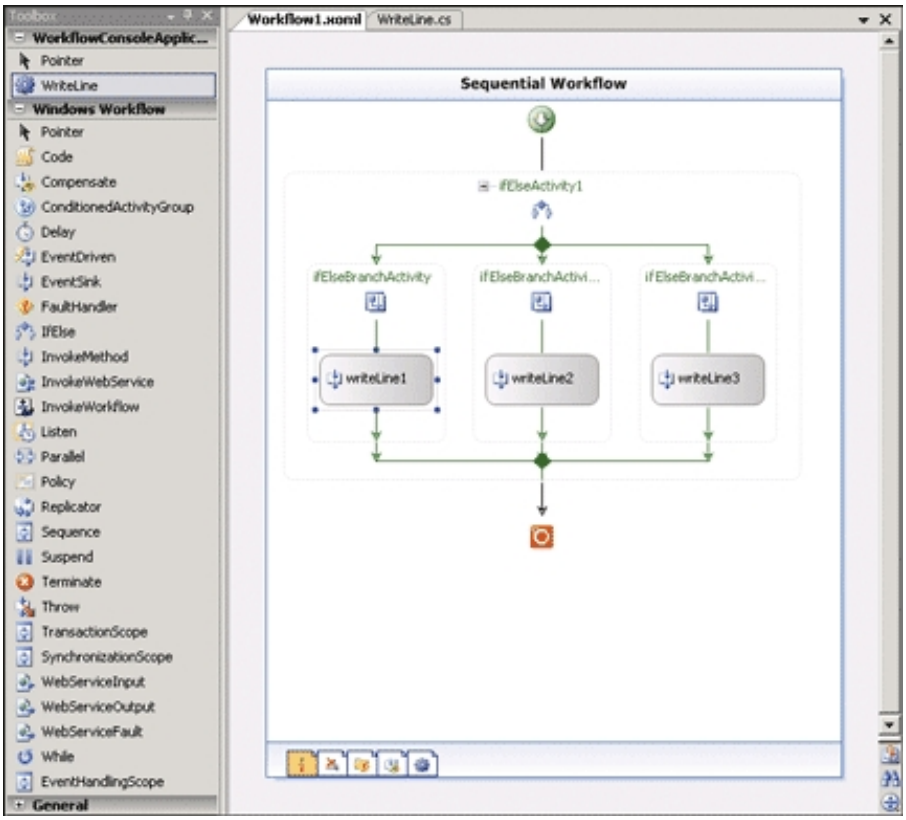


Abbildung 51: Darstellung des Beispiel-Workflows im *Visual Workflow Designer* [*Quelle: [2]*]

ter. Somit hat jede *Activity* einen Zustandsautomaten zu implementieren, durch den der Zustand jeder *Activity* modelliert wird und somit von der Laufzeitumgebung ausgewertet, kontrolliert und manipuliert werden kann. Abbildung 52 zeigt eine schematische Darstellung dieses Zustandsautomaten mit den sechs Zuständen. Die normale Ausführung einer *Activity* führt zur Zustandsfolge *Initialized*, *Executing* und endet im Zustand *Closed*. Dabei wird jeder Transistionswechsel eines Zustandsübergangs in Form eines Ereignisses der nächsthöheren Struktur der *Activity* angezeigt. Dabei kann es sich um eine *Composite Activity* oder um die Workflow-Instanz selbst handeln.

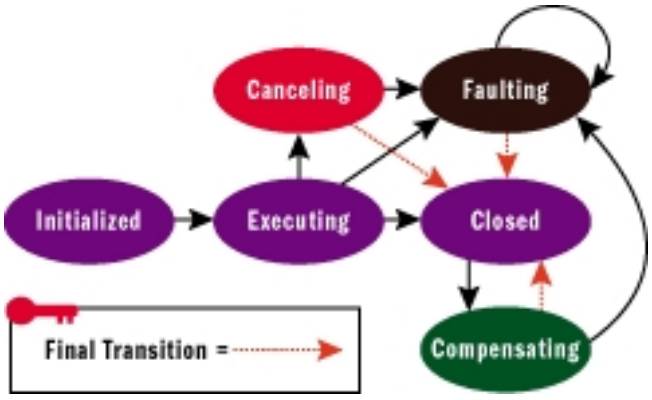


Abbildung 52: Schematische Darstellung des Zustandsautomaten einer *Activity* [*Quelle: [2]*]

14.3.2 Windows Workflow Foundation im Vergleich

Die *Windows Workflow Foundation* erweitert das *.NET Framework* um Fähigkeiten zur Workflow-basierten Entwicklung von Softwaresystemen bzw. deren Komponenten. Es bietet dabei Mechanismen zur graphischen Modellierung von Arbeitsabläufen und Festlegung des Kontrollflusses und der komponentenweisen Entwicklung von Arbeitsschritten. Diese *Activities* sind in einer *.NET*-Programmiersprache gesondert zu implementieren. In der Definition des Arbeitsflusses können den einzelnen *Activities* Parameter übergeben werden, um das Verhalten kontextbasiert zu steuern. Vergleicht man diese *Activities* mit aus dem jABC bekannten SIBs, so weisen beide Komponenten sowohl in ihrem Zweck als auch in der strukturierten und hierarchisierbaren Komposition Ähnlichkeiten auf. Auch SIBs müssen generell unabhängig von der genutzten Programmiersprache implementiert werden und sind in ihrem Funktionsumfang als Einzelkomponente genau so ausdrucksstark, wie die zur Implementierung verwendete Sprache. Diese Eigenschaft lässt sich auf die in der *Windows Workflow Foundation* verwendeten *Activities* übertragen. Somit stehen Datenbankzugriff und die Kommunikation mittels Webservices bei beiden Systemen durch die APIs der Programmiersprachen C# und Java zur Verfügung. Bei beiden Systemen kann die Definition von Workflows durch XML vorgenommen werden, wenn auch bei *Windows Workflow Foundation* in dem XML-Derivat XAML. Durch spezielle *Composite Activities* können Event-Condition-Action-Workflows und -Regelsysteme aufgebaut werden, wie es auch eingeschränkt mit Hilfe des EVA-Subsystems der IGM-Plattform möglich ist.

Dennoch ist ein direkter Vergleich beider Technologien nicht möglich, da es sich bei *Windows Workflow Foundation* um ein Entwicklungs-Framework handelt, die Kombination aus IGM-Plattform und das um die jABCProzessPlattform erweiterte jABC im SmarterWohnen-Kontext hingegen eine konkrete Implementierung einer Anwendungsdomäne darstellt. Die IGM-Plattform selbst ist zwar grundsätzlich auch ein Framework zur Entwicklung von informationslogistischen Anwendungen, jedoch spezieller und höherschichtiger als die Komponente *Windows Workflow Foundation* des *.NET Frameworks*. Somit implementiert die IGM-Plattform eine Benutzerverwaltung in Form des User- und des Kontext-Subsystems, eine entsprechende Komponente auf Seiten der *Windows Workflow Foundation* muss mit Hilfe von *.NET* erst implementiert werden. Die Tatsache, dass die genannten Subsysteme der IGM-Plattform *.NET*-Implementierungen darstellen, verdeutlicht die Unmöglichkeit eines konkreten Vergleichs zwischen beiden Produkten.

15 Bewertung der Projektorganisation

Projektgruppenorganisation und Gruppenaufteilung Bereits zu Anfang der Projektgruppe zu Beginn des Wintersemesters 2006/2007 wurde die elfköpfige PG in drei Teams aufgeteilt, um den verschiedenen Aufgabenbereichen zur Erfüllung der PG-Ziele gerecht zu werden. Da das Hauptziel der Projektgruppe der Zusammenschluss der Systeme IGM-Plattform und JavaABC zu einem lose gekoppelten Gesamtsystem war, wurden naturgemäß je ein Team mit der Analyse und Erweiterung der Systeme beauftragt. Auf diese Weise konnten sich beide vierköpfigen Teams Expertenwissen bei der intensiven Einarbeitung aneignen und die technische Realisierung der Systeme vorantreiben. Weiterhin wurde das aus drei PG-Mitgliedern bestehende Design-Team gebildet, welches die Analyse und Modellierung der *SmarterWohnen*-Abläufe durchführte und sich in die Anwendungsdomäne des Gesamtsystems einarbeitete. Durch die Bildung dieser nicht technisch orientierten und auf die Modellbildung ausgerichteten Teilgruppe, konnte eine abstrakte Sichtweise auf das System geschaffen werden, mit deren Hilfe sich Lösungskonzepte erarbeiten ließen, die über den rein technischen und auf die Implementierung ausgerichteten Blickwinkel

	WWF	*TriGS$_{flow}$*	jBoss	POETS
Datenbankanbindung	relationale Datenbank via ODBC	GemStone (OODB)	relationale Datenbank	relationale Datenbank
Programmiersprache	alle .NET-Sprachen	OPAL	Java	Java & C#
Unterstützung von ECA-Regelsystemen	ja ECA-Workflows	ja TriGS	ja (jBoss Rules)	ja (EVA)
graphische Prozess-modellierung	ja (Visual WF Designer)	ja (MasterTOOL)	ja (jBPM)	ja (jABC)
Benutzerverwaltung	Active Directory	Roles (rollenbasiert)	jBoss SX	User Subsystem

Tabelle 9: Direktvergleich von zentralen Eigenschaften der diskutierten Business Process Management Plattformen

hinaus gingen. Besonders innerhalb des ersten Semesters der Projektgruppe erwies sich diese Dreiteilung der Zuständigkeitsbereiche als sinnvoll, da sie gerade durch die Arbeit in kleinen Gruppen eine intensive und problemorientierte Aufgabenbewältigung ermöglichte. Weiterhin konnte jede Gruppe durch die Spezialisierung auf die Teilaspekte des Gesamtsystems mit Hilfe dieser Divide-and-Conquer-Methode „ihren“ Zuständigkeitsbereich als Expertengruppe besonders intensiv bearbeiten, was einerseits zu höherem Engagement und andererseits zu einer starken Identifizierung mit dem zu bearbeitenden Aufgabenbereich führte. Der Kommunikationsaufwand zwischen den Gruppen war jedoch somit recht groß, da gerade bei globalen Fragestellungen zur **Entscheidungsfindung** die übrigen PG-Teilnehmer jeweils zunächst mit der Problemstellung vertraut gemacht und angelernt werden mussten. Durch die Wahl eines Teamleiters in den drei Gruppen konnten Koordinierungsprobleme minimiert werden, da sich bei der Zusammenarbeit von mehreren Gruppen nicht alle Mitglieder sondern lediglich die Teamleiter abstimmen mussten und auf diese Weise Entscheidungen im kleinen Kreis getroffen und mit den Gruppenmitgliedern abgestimmt werden konnten. Eine weitere wichtige Instanz zur Koordination und Lenkung der Projektgruppe hin auf ihre eigentlichen PG-Ziele stellte der Projektmanager dar. Die in anderen Projektgruppen übliche Round-Robin-Methode, jedes PG-Mitglied mit der Rolle des Projektmanagers zu beauftragen, wurde in unserer Projektgruppe bewusst nicht angewendet; stattdessen wurde Stephan Schulte mit der Erfüllung dieser Aufgabe zunächst für das erste Semester betraut. Auch für das zweite Semester wurde einstimmig beschlossen, dass diese Aufgabe weiterhin von Stephan durchgeführt werden sollte. Der Projektmanager konnte stets den Gesamtüberblick über das Projekt als solches, sowie die Aufgaben und Ziele der einzelnen Teams behalten. Termine konnten auf diese Weise besser abgestimmt und eingehalten werden. Durch die Zusammenarbeit mit den Teamleitern der drei Gruppen war es ihm zudem möglich, den Projektablauf zu koordinieren und gemeinsam mit den Leitern globale Fragestellungen in kleinem Kreis anzugehen. Um die Rolle des Projektmanagers erfüllen zu können, war jedoch während des gesamten Durchführungszeitraums der PG eine intensive Zusammenarbeit mit allen Teilgruppen vonnöten, was schlichtweg in einer zu den übrigen PG-Teilnehmern vergleichsweise höheren Arbeitsbelastung resultierte. Aus diesem Grund wurde in einer der letzten PG-Sitzungen dieses überdurchschnittliche Engagement gewürdigt und festgestellt, dass die Erfüllung der Rolle des Projektmanagers ohne eine vergleichbare Belastung nicht möglich wäre, da sonst der

Überblick über den Gesamtzustand der PG nicht hätte gewahrt werden können. Weiterhin wurde die Sitzungsleitung der wöchentlichen Projektgruppentreffen durch den Projektmanager durchgeführt und somit ebenfalls von der gängigen Praxis bei der PG-Durchführung abgewichen, diese Rolle abgwechselnd auf jedes PG-Mitglied zu verteilen.

Zeitlicher Ablauf der Projektgruppe Beim Start der Projektgruppe kam es anfänglich zu einigen Verzögerungen, da die Lizenzabsprachen zwischen Prof. Steffen (JavaABC) und dem Fraunhofer ISST (IGM-Plattform) zu teilweise widersprüchlichen Aussagen führten und von den PG-Teilnehmern nicht unterzeichnet werden konnte. Erst durch eine spätere Klärung und Abänderung der Lizenzvereinbarung mit dem ISST konnte die IGM-Gruppe ihre Arbeit aufnehmen und sich im ISST mit der IGM-Plattform vertraut machen. Diese anfängliche Verzögerung konnte jedoch im weiteren Verlauf der PG wieder ausgeglichen werden. Der erste Teil der Projekts zeichnete sich durch eine lange Konzeptionsphase aus, während der es möglich war, durch die intensive Einarbeitung in die Materie der Systeme jABC und IGM sowie die Analyse der *SmarterWohnen*-Abläufe Probleme im Vorfeld ausschließen zu können. Die zeitliche Organisation der PG wurde durch die Definition von Meilensteinen erreicht, die vom Projektmanager in Zusammenarbeit mit den Teamleitern festgesetzt wurden. Auf diese Weise war es stets möglich, auf Teilerfolge der Projektgruppe hinzuarbeiten und die Projektziele erreichen zu können. Auch wurden diese terminlichen Festsetzungen bis auf die Fertigstellung dieses Endberichts hauptsächlich eingehalten, sodass die Projektgruppe zum Ende des Sommersemesters 2007 ihre Ziele erreichen konnte. Auch in der vorlesungsfreien Zeit wurden stets Projektgruppentreffen organisiert, um den weiteren Ablauf koordinieren zu können, Arbeitsergebnisse vorzustellen und die PG permanent „lebendig“ zu halten. Eine Übersicht über die für beide Semester definierten Meilensteine kann den Abbildungen 53 sowie 54 entnommen werden.

Arbeitsumfeld und Projektgruppenschnittstellen Problematisch für den Arbeitsfluss der Projektgruppe erwies sich zunächst die räumliche Trennung der beiden technischen Teams jABC und IGM. Während das erstgenannte Team naturgemäß im Lehrstuhl V und XIV an der Universität arbeitete, war das ISST-Team an die im Fraunhofer ISST bereitgestellten Arbeitsplätze gebunden, da die Entwicklungsumgebung sowie der zur Erweiterung der IGM-Plattform notwendige Server nur am Institut bereit standen. Daher arbeiteten in der späteren Entwicklung der PG beide Gruppen am ISST, um die notwendige Kommunikation zur Abstimmung der Aufgaben aufrecht halten zu können. Auch halfen häufig Kommunikationsmedien wie E-Mail, Chat und Skype, um die Barriere der räumlichen Trennung zu überwinden, sodass selbst die Live-Teilnahme von verhinderten PG-Mitgliedern an Sitzungen möglich war. Obwohl das ISST wenig Erfahrung mit Projektgruppen des Fachbereichs Informatik hatte, war die Zusammenarbeit jedoch stets sehr entgegenkommend und reibungslos. Auch wurde die PG durch die Bereitstellung von Sensoren und Räumlichkeiten unterstützt. Bei Problemen ergriff Prof. Rehof die Initiative und koordinierte die Zusammenarbeit, falls Ressourcen wie beispielsweise der anfängliche Zugang zu Rechnerarbeitsplätzen fehlten. Insbesondere ist die Betreuung von Prof. Rehof hervorzuheben, da er im Gegensatz zu Veranstaltern paralleler Projektgruppen anderer Lehrstühle trotz stets gefülltem Terminkalender häufig Zeit fand, die wöchentlichen Projektgruppentreffen zu besuchen, auf diese Weise den PG-Ablauf verfolgen konnte und durch die Steuerung fachlicher Diskussionen die PG unterstützte. Des Weiteren trug die intensive Betreuung von Markus Doedt und Martin Sugioarto maßgeblich dazu bei, dass die Projektgruppe ihre Ziele innerhalb des zeitlichen Rahmens erreichen konnte, da sie der PG bei Problemen zur Seite standen und durch konstruktive Kritik den Blick auf wichtige Fragestellungen lenkten.

Teil VII
Anhang

Zeitplanung PG POETS 1. Semester

Zeitraum	Design	jABC	ISST
KW 49	- Beginn Abbildung der Kernprozesse als eEPK	- Eclipse-Projektstruktur festlegen. - Basisklassen anlegen - Schnittstellen definieren	- Einarbeitung in die Plattform - Verständnis des Zusammenspiels der einzelnen Subsysteme - Kennenlernen der Schnittstellen

KW 50		- Basisklassen implementieren
		  ○ TimeoutQueue (mit Tests)
		  ○ ProcessManager
		  ○ ProcessHandler
		- Beispiel jABC Prozess aufstellen
		- Basis SIBs implementieren
		- Doku	

KW 51	- Einarbeitung in das tatsächliche System beim ISST - Beginn Transformation eEPK in jABC (Teil1)		- Konzeption für die Implementierung der Anbindung des jABC und des Webservices sowie der Schnittstelle für die BPM-SubSystem - Festlegung der technischen Rahmenbedingungen für die Umsetzung (Teil1)
Milestone I 21. Dezember	**Teamleitermeeting: Anforderungen jABC/ISST / Konzeptabsprache** Anforderungen an SIBs (Design -> jABC) Anforderungen an ISSTPlattform (Design -> ISST)		
KW 52	- Einarbeitung in das tatsächliche System beim ISST - Beginn Transformation eEPK in jABC (Teil2)	- Tests (ProcessManager, ProcessHandler) mit dem BeispielProzess - Doku	- Konzeption für die Implementierung der Anbindung des jABC und des - Webservices sowie der Schnittstelle für die BPM-SubSystem - Festlegung der technischen Rahmenbedingungen für die Umsetzung (Teil2)
KW 1	- Modellierung der Kernprozesse im jABC - feierliche Übergabe der SIB's an die jABC-Gruppe - Wahl Sonderausschuss Zwischenbericht, alle PG-Teilnehmer (Teil1)		
Milestone II 11. Januar	**Doku: (komplette) Doku der Teams** ISST: IST-Zustand und Idee/Konzept Design: IST-Zustand, EPK (Beschreibung der Prozesse -> jABC-Graphen), Probleme bei der Umsetzung auf Prozesse jABC: IST-Zustand(Kurzbeschreibung jABC), Konzept (Komponentendiagramm) ALLE: Webservicekonzept		
KW 2	- Modellierung der Kernprozesse im jABC - feierliche Übergabe der SIB's an die jABC-Gruppe	- Webserviceanbindung (Testanwendung) - Tests	- detailierte Spezifikation der Schnittstellen und Festlegung der - genauen technischen

Abbildung 53: Terminplanung der PG erstes Semester

Zeitplanung PG POETS 2. Semester

Zeitraum	Design	jABC	ISST
14. – 16. KW	Präsentationsvorbereitung		
17. – 18. KW	- Beispielprozess/ Testprozess - Datenpakete definieren/ zusammenbauen - Beschreibung der SIBs	- Server jABCPlattform einrichten - Webservice installieren - Erste Tests auf ABC Seite - Monitoring - Beispielprozess / Testprozess	- Server einrichten - Tests - Logging - DataMapper
2. Mai I. MStone	**1. lauffähiger Prototyp mit Testprozess**		
			Machbarkeitsanalyse Sensordaten
19. – 23. KW	- Modellierung der Prozesse (vor allem fachliche Ebene) - Implementieren der SIBs - Testen der Prozesse - Testen des Datamappings		
6. Juni II. MStone	**Komplette Modellierung der Prozesse auf fachlicher Ebene**	**Elementare technische SIBs fertiggestellt und getestet**	**Anbindung der Sensoren**
24. – 25. KW	- Modellierung der Prozesse komplett - Testen der Prozesse - Erstellen / Testen der Demo - Endbericht vervollständigen		
27. Juni III. MStone	**1. Version Endbericht (not reviewed)**		
26. -28. KW	- Endbericht Reviewen / Vervollständigen - Erstellen / Testen der Demo		
11. Juli IV.MStone	**Abgabe Endbericht** **Vorstellung der Abschlussdemonstration (Prototyp)**		

Abbildung 54: Terminplanung der PG zweites Semester

Literatur

[1] Elena Baralis and Jennifer Wisdom. Better static rule analysis for active database systems, Technical report. 1996.

[2] Don Box and Dharma Shukla. Simplify Development With The Declarative Model Of Windows Workflow Foundation. *MSDN Magazine*, 21(1):10–14, Januar 2006.

[3] Youssef Ben Cheikh. Workflow Management. Seminar pg poets, Universtät Dortmund, September 2006.

[4] Microsoft Corporation. Data Contract Schema Reference. http://msdn2.microsoft.com/en-us/library/ms733112.aspx.

[5] Microsoft Corporation. MSDN Library - .Net Development - Samples - Windows Communication Foundation Samples - Windows Communication Foundation Technology Samples - Windows Communication Foundation Extensibility Samples - Message Inspectors . http://msdn2.microsoft.com/en-us/library/aa717047.aspx.

[6] Microsoft Corporation. MSDN Library .Net Development - Class Library. http://msdn2.microsoft.com/en-us/library/aa388745.aspx.

[7] Unterguppe Design der Projektgruppe POETS. Anforderungsanalyse - parameter?ubersicht, 2006/2007.

[8] NUnit development team. NUnit. http://www.nunit.org/.

[9] Markus Doedt. TaxonomieEditor. http://jabc.cs.uni-dortmund.de/plugins/taxonomyeditor/index.html, Sept. 2006.

[10] Markus Doedt. Tracer. http://jabc.cs.uni-dortmund.de/plugins/tracer.html, Sept. 2006.

[11] Dogewo21. Wohnen in den besten Jahren, Informationsbroschüre, 2006.

[12] Apache Software Foundation. Apache logging services - log4j. http://logging.apache.org/log4j/.

[13] Apache Software Foundation. Apache Logging Services - log4net. http://logging.apache.org/log4net/.

[14] Fraunhofer-Institut für Software-und Systemtechnik (ISST) Dortmund. Blueprint Brandmeldung Vers. 1.1 - Dienstnutzung.pdf, 2006.

[15] Fraunhofer-Institut für Software-und Systemtechnik (ISST) Dortmund. Blueprint Einbruchmeldung Vers. 1.1 - Dienstnutzung.pdf, 2006.

[16] Fraunhofer-Institut für Software-und Systemtechnik (ISST) Dortmund. Blueprint Gesamtprozess Vers. 1.1.pdf, 2006.

[17] Fraunhofer-Institut für Software-und Systemtechnik (ISST) Dortmund. Blueprint Vitalcheck Vers. 1.1 - Dienstnutzung.pdf, 2006.

[18] Fraunhofer-Institut für Software-und Systemtechnik (ISST) Dortmund. Plattformabläufe.pdf, 2006.

[19] Annette Franke and David Patrick Wilde. Diplomarbeit Die silberne Zukunft gestalten. Master's thesis, Ruhr-Universität-Bochum, 2005.

[20] Serwo GmbH. http://www.serwo.biz/, 09. september 2006, 2006.

[21] Zenit GmbH. http://www.zukunftswettbewerb.de/, 09. september 2006, 2006.

[22] google. google web toolkit. http://code.google.com/webtoolkit/, 2007.

[23] Joseph M. Hellerstein, Alexander Aiken, and Jennifer Wisdom. Static analysis techniques for predicting the behavior of active database rules, Technical report. 1999.

[24] Uni Dortmund Informatik Lehrstuhl 5. jABC Dokumentation. http://jabc.cs.uni-dortmund.de, Sept. 2006.

[25] inHaus. http://www.inhaus-duisburg.de/, 09. september 2006, 2006.

[26] Fraunhofer ISST, R. Gellrich, S. Haseloff, T. Vogd, N. Weissenberg, and M. Wojciechowski. Coarse Design - Logical architecture of the IGM platform, January 2005.

[27] Fraunhofer ISST, Ralf Gellrich, and Peter Binner. Komponentenmodell - Subsystem Delivery / Interaction / Application, January 2006.

[28] Fraunhofer ISST, Ralf Gellrich, and Peter Binner. Komponentenmodell - Subsystem Delivery / Interaction / Application, January 2006.

[29] Fraunhofer ISST, Ralf Gellrich, Thomas Vogd, and René Harings. IGM-Plattform - Event-Handling, August 2005.

[30] Sven-Torben Janus. Business Process Management. *Seminar PG POETS*, September 2006.

[31] Sven-Torben Janus. Konzeptvorschlag zur Validierung von Parametertypen der DataTransferObjects und Events und deren Implementierung innerhalb des BPMSubSystems, Mai 2007.

[32] Sven Jörges. FormulaBuilder. http://jabc.cs.uni-dortmund.de/plugins/formulabuilder.html, Sept. 2006.

[33] Sven Jörges and Michael Drazek. CodeGenerator. http://jabc.cs.uni-dortmund.de/plugins/codegenerator.html, Sept. 2006.

[34] Gerti Kappel, Peter Lang, S. Rausch-Schott, and Werner Retschitzegger. Workflow management based on objects, rules, and roles. *Data Engineering Bulletin*, 18(1):11–18, 1995.

[35] David Karla. ECA-Regelsysteme 2. Universität Dortmund, Lehrstuhl 5, 21.09.2006.

[36] Christian May. ECA-Regelsysteme 1. Universität Dortmund, Lehrstuhl 5, 27.09.2006.

[37] Johannes Neubauer. LocalChecker. http://jabc.cs.uni-dortmund.de/plugins/localchecker.html, Sept. 2006.

[38] Johannes Neubauer. SIBCreator. http://jabc.cs.uni-dortmund.de/plugins/sibcreator.html, Sept. 2006.

[39] Department of Computer Science, Elena Baralis, and Jennifer Widom. Better static rule analysis for active database systems, 1996.

[40] A. Picot and P. Rohrbach. Organisatorische Aspekte von Workflow-Management-Systemen,. *Information Management*, 1, 1995.

[41] PG Poets. Anforderungsanalyse. unpublished, 2006.

[42] Projektgruppe POETS. API-Dokumentation BPM-SubSystem (digitale version), 2007.

[43] Projektgruppe POETS. BPMSubSystem - Entwicklerhandbuch, 2007.

[44] Projektgruppe POETS. Zwischenbericht, April 2007.

[45] Clemens Renner and Marco Bakera. GEAR. A Model Checking Tool. http://jabc.cs.uni-dortmund.de/modelchecking/gear.html, Sept. 2006.

[46] August-Wilhelm Scheer. *ARIS - Modellierungsmethoden Metamodelle Anwendungen*. Springer Verlag Berlin Heidelberg New York, 1. edition, 1992.

[47] August-Wilhelm Scheer. *ARIS - Modellierungsmethoden Metamodelle Anwendungen*. Springer Verlag Berlin Heidelberg New York, 3. edition, 1998.

[48] August-Wilhelm Scheer. *ARIS - Modellierungsmethoden Metamodelle Anwendungen.* Springer Verlag Berlin Heidelberg New York, 4. edition, 2001.

[49] Markus Schlesinger. ALFRED - Konzepte und Prototyp einer aktiven Sicht zur Automatisierung von Geschäftsregeln, Kapitel 3 - Aktive Datenbanksysteme, PhD thesis. 1999.

[50] Lothar Schöpe and Jochen Meis. Smart Living mit dem Fraunhofer ISST: IT-Lösungen für ein Wohnen mit MehrWert. 2006.

[51] Lothar Schöpe. Lebensmittellieferservice - Eine Anwendung, 2005.

[52] Peter Stahlknecht and Ulrich Hasenkamp. *Einführung in die Wirtschaftsinformatik.* Springer, 11. edition, 2005.

[53] Fraunhofer ISST Thomas Vogd. Konzeption von BooleanJobs im Rahmen des ILOG-Frameworks, February 2003.

[54] Fraunhofer ISST Thomas Vogd. IGM-platform - Specification of action-jobs, July 2005.

[55] Fraunhofer ISST Thomas Vogd. IGM-platform - Specification of the evaluation-subsystem, January 2006.

[56] Thomas Vogd. Konzeption von BooleanJobs im Rahmen des ILOG-Frameworks. Fraunhofer Institut, 21.02.03.

[57] World Wide Web Consortium (W3C). W3C XML Schema. http://www.w3.org/XML/Schema.

[58] Fraunhofer ISST Norbert Weißenberg. Component Model - Service Subsystem, July 2005.

[59] Fraunhofer ISST Manfred Wojciechowski. Component Model - Context Subsystem (Version 0.3), August 2005.