Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Johann Bronsch

Entwicklung einer domänenspezifischen Sprache zur Konfiguration von komplexen Smart Homes

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Johann Bronsch

Entwicklung einer domänenspezifischen Sprache zur Konfiguration von komplexen Smart Homes

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Wirtschaftsinformatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Kai von Luck
Zweitgutachter: Prof. Dr. Philipp Jenke

Eingereicht am: 1. Februar 2016

Johann Bronsch

Thema der Arbeit

Entwicklung einer domänenspezifischen Sprache zur Konfiguration von komplexen Smart Homes

Stichworte

Smart Home, Smart Environments, Benutzer, Prozess, Prozesssteuerung, Szenarien, Backend, Modellierungssprachen, MDD, Model Getriebene Architektur, domänenspezifischen Sprache

Kurzzusammenfassung

Ziel dieser Bachelorarbeit ist die Entwicklung einer domänenspezifischen Sprache zur Konfiguration von komplexen Smart Homes. Dabei soll es, vorwiegend für Softwareentwickler, möglichst einfach sein neue Prozesse anzulegen und diese in das System einzubinden. Dafür wurde eine eigene domänenspezifische Sprache, im weiteren auch DSL genannt, entwickelt und deren Semantik definiert. Darauf basierend wurde ein Prototyp einer Prozess-Engine entwickelt.

Johann Bronsch

Title of the paper

Development of a domain specific language for the configuration of complex Smart Homes

Keywords

Smart Home, Smart Environments, user, process control, scenarios, backend, modeling languages, MDD, Model Driven Architecture, domain specific language

Abstract

The goal of this bachelor thesis is the development of a domain specific language to configure complex smart homes. It is hereby crutial to software developers that it is as easy as possible to apply new processes and integrate them into a system. A separate Domain Specific Language has been developed and it semantics defined. Based on that a prototype of a process-engine was developed.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Zielsetzung 2

1.2 Gliederung 2

2 Analyse **4**

2.1 Einleitung 4

2.2 Smart Home 4

2.3 Domain Specific Languages 7

2.3.1 Einführung 7

2.3.2 Grundlagen 8

2.3.3 Bestandteile von Domain Specific Languages 9

2.3.4 Vor- und Nachteile 10

2.4 Entitäten im Livingplace 10

2.4.1 Einleitung 10

2.4.2 Agenten 11

2.4.3 Timer 14

2.4.4 Zusammenfassung 15

2.5 Szenarien 15

2.5.1 Zeitlich vordefiniertes Lüften anhand von Wetterdaten 15

2.5.2 Zeitlich vordefiniertes Lüften 18

2.5.3 Event-Basiertes Lüften 19

2.5.4 Event-Basiertes Lüften anhand von Wetterdaten 20

2.5.5 Angesprochene Entitäten 21

2.5.6 Zusammenfassung 22

2.6 Anforderungen 23

2.6.1 Einleitung 23

2.6.2 Vision 23

2.6.3 Minimal-Entwurf 26

2.7 Zusammenfassung 27

3 Entwurf der Domain Specific Language **28**

3.1 Einleitung 28

3.2 Grundlagen 28

3.2.1 JavaScript Object Notation 29

3.2.2 Business Process Model And Notation 31

3.3 Definition der Sprache 31
3.3.1 Sprachstruktur 31
3.3.2 Regeln 32
3.4 Sprachelemente 34
3.4.1 Einleitung 34
3.4.2 Objektdefinition 35
3.5 Architektur-Übersicht 40
3.5.1 Entwicklungsoberfläche 41
3.5.2 Prozess-Engine 42
3.5.3 Task Interpreter 44
3.5.4 Test Environment 45
3.6 Erweiterbarkeit 46

4 Evaluation **47**
4.1 Implementierung 47
4.1.1 Einleitung 47
4.1.2 Entwicklungsoberfläche 47
4.1.3 Prozess-Engine Prototyp 49
4.1.4 Fazit 51
4.2 Domänenspezifische Sprache 53
4.2.1 Evaluation der DSL 53
4.2.2 Implementierte Elemente der Sprache 54
4.2.3 Ausblick 55
4.3 Fazit 55

5 Fazit **59**
5.1 Zusammenfassung 59
5.2 Ausblick 59

Abbildungsverzeichnis

2.1 Beispiel Skript für Cucumber . . . 9
2.2 DSL Schema (vgl. Voelter (2009)) . . . 10
2.3 Middleware - Architektur (Eichler, 2014, S.40) . . . 11
2.4 Weather Case Class, mit allen Informationen die der Wetter-Agent versendet . 13
2.5 Modul Komponente . . . 14
2.6 Zeitlich vordefiniertes Lüften anhand von Wetterdaten . . . 17
2.7 Setzten der Properties für einen Task . . . 18
2.8 Zeitlich vordefiniertes Lüften . . . 19
2.9 Event-Basiertes Lüften . . . 20
2.10 Event-Basiertes Lüften anhand von Wetterdaten . . . 21

3.1 Darstellung eines Start-Events . . . 35
3.2 Darstellung eines End-Events . . . 36
3.3 Darstellung eines Timer-Events . . . 36
3.4 Darstellung eines Tasks . . . 37
3.5 Darstellung eines Ad-Hoc Sub-Prozess . . . 38
3.6 Darstellung eines Exklusiven Gateways . . . 39
3.7 Darstellung eines Parallelen Gateways . . . 40
3.8 Komponenten Diagramm . . . 41
3.9 Darstellung der Entwicklungsumgebung . . . 42
3.10 Detailansicht der Prozess Engine . . . 43
3.11 Prozessablauf dargestellt in einem Sequenz-Diagramm . . . 44
3.12 Klassendiagramm: TaskInterpreter . . . 45

4.1 Layer-Model zur Entwicklungsoberfläche . . . 48
4.2 Klassendiagramm zu Node und Link . . . 50
4.3 Simples Szenario erstellt mit der Entwicklungsumgebung aus Unterabschnitt 4.1.2 56
4.4 Ausgabe des ausgeführten Szenarios . . . 57

1 Einleitung

Der Begriff Smart Environment ist nicht neu, bereits in den neunziger Jahren wurde *Ubiguitous Computing* (vgl. Weiser (1999)) als Vorgang beschrieben, der den Menschen im Leben unterstützt ohne aufzufallen oder abzulenken. Aufgrund der stetigen fortschreitenden technologischen Entwicklung können immer kleinere und leistungsfähigere Computer gebaut werden. Dies ermöglicht es immer mehr Objekte aus dem persönlichen Umfeld gegen Smart-Objekts auszutauschen. Smart-Objekt sind in diesem Kontext Gegenstände, die um eingebettete Computer ergänzt wurden. Dabei sollen diese nicht nur Daten aus Sensoren auslesen, sondern mit anderen Smart-Objekts kommunizieren.

Bei einer Vernetzung von Smart-Objekts zu einem komplexen verteilte System werden oft Multiagentensysteme eingesetzt. Bei diesem wird ein Smart-Objekt als ein Agent eingesehen, dieser kann dann mit anderen Agenten durch den Austausch von Nachrichten zusammenarbeiten.

Das Living Place an der Hochschule für Angewandte Wissenschaften Hamburg[1] bietet eine Umgebung, bei der Smart-Objekts mit Hilfe einer Middleware Nachrichten austauschen können. Jene bietet den Studierenden die Möglichkeit ihre eigenen Smarten-Objekte zu entwerfen und in die Middleware zu implementieren. Zusätzlich können auch komplexe Abläufe, die die Zusammenarbeit mehrerer Objekte erfordern, erstellt und implementiert werden.

Die sogenannten komplexen Abläufe, im weiteren Verlauf der Arbeit Szenarien genannt, können jedoch schnell unübersichtlich werden. Einige dieser Szenarien erfordern ferner die Zusammenarbeit von Softwareentwicklern und den jeweiligen Domain Spezialisten. Hilfreich ist dafür eine domänenspezifische Sprache um solche Szenarien zu beschreiben und abbilden zu können. Eine DSL kann durch ein Werkzeug unterstützt werden, welches es dem Anwender erlaubt anspruchsvolle Modelle zu entwerfen.

[1]Hochschule für Angewandte Wissenschaften Hamburg - http://www.haw-hamburg.de; letzter Zugriff: 31.01.2016

1.1 Zielsetzung

Ziel dieser Arbeit ist die Entwicklung einer domänenspezifischen Sprache für die Forschungsumgebung Living Place an der HAW. Anhand dieser domänenspezifischen Sprache und dem dazugehörigen Werkzeug sollen Domain Spezialisten Szenarien abbilden können. Das Werkzeug wird im weiterem verlauf der Arbeit entwickelt. Dabei sollen die entworfenen Szenarien nicht ausschließlich modelliert, sondern ebenfalls ausgeführt werden können. Das Ausführen der Szenarien soll durch eine Prozess-Engine ermöglicht werden. Aus dem Entwurf der domänenspezifischen Sprache, sowie der Prozess-Engine, ergeben sich Anforderungen die in dieser Arbeit analysiert werden.

Es ist geplant einen Prototypen zu entwickeln, der die Machbarkeit dieser Vision demonstriert. Des Weiteren soll die Kommunikation mit der Middleware simuliert werden. Um dies zu ermöglichen muss der Prototyp auch das von der Middleware genutzte Framework in die Prozess-Engine implementiert werden.

1.2 Gliederung

In dem folgenden Kapitel 2 wird zuerst der Begriff Smart-Home definiert und die Motivation dieser Arbeit erläutert. Des Weiteren werden die Grundlagen von domänenspezifischen Sprachen analysiert und die Entitäten im Living Place definiert. Anhand dieser Definitionen werden vier Szenarien erstellt. Im Anschluss daran werden die Anforderungen zur Durchführung der Vision aufgestellt. Diese werden auf einen Minimal-Entwurf reduziert, der in den folgen Kapiteln umgesetzt wird.

In Kapitel 3 wird zunächst die DSL entworfen und einige Design Entscheidungen begründet. Darüber hinaus wird die Sprachstruktur und die Regeln dieser definiert. Ferner werden die Elemente der Sprache definiert, die in die Architektur der Entwicklungsoberfläche und der Prozess-Engine eingebunden werden.

Der Entwurf aus dem Kapitel 3 wird in Abschnitt 4.1 implementiert. Hier wird der Umfang der implementierten Entwicklungsoberfläche und der Prozess-Engine veranschaulicht und deren Funktion erläutert. Anschließend wird die entwickelte domänenspezifische Sprache evaluiert und der implementierte Umfang dargestellt. Abschließend wird auf einige fehlende Elemente dieser Implementierung verwiesen, an die zukünftige Arbeiten verwiesen anknüpfen können.

Das Kapitel 5 fasst alle Kapitel und die Ergebnisse der Arbeit zusammen. Abschließend erfolgt ein Ausblick auf weitere interessante Fragestellungen.

2 Analyse

2.1 Einleitung

In diesem Kapitel wird zunächst erläutert was Smart Homes sind und darauf aufbauend wird die Motivation, die zu dieser Arbeit geführt hat, formuliert. Im Anschluss werden die Grundlagen von domänenspezifischen Sprachen, aber auch das Labor Living Place an der HAW dargelegt. Des Weiteren werden Szenarien, zum Thema Lüften der Wohnung, vorgestellt und anhand der ausgearbeiteten Definition analysiert und diskutiert. Im Anschluss werden die Anforderungen an die Sprache und an die dazugehörigen Werkzeuge formuliert. Diese Anforderungen bilden die Grundlage für die in den Kapiteln 3 und 4.1 ausgearbeiteten Inhalte sein.

2.2 Smart Home

> „Das Smart Home ist ein privat genutztes Heim (z. B. Eigenheim, Mietwohnung), in dem die zahlreichen Geräte der Hausautomation (wie Heizung, Beleuchtung, Belüftung), Haushaltstechnik (wie z. B. Kühlschrank, Waschmaschine), Konsumelektronik und Kommunikationseinrichtungen zu intelligenten Gegenständen werden, die sich an den Bedürfnissen der Bewohner orientieren. Durch Vernetzung dieser Gegenstände untereinander können neue Assistenzfunktionen und Dienste zum Nutzen des Bewohners bereitgestellt werden und einen Mehrwert generieren, der über den einzelnen Nutzen der im Haus vorhandenen Anwendungen hinausgeht." Strese u. a. (2010)

Diese eine Definition bezieht sich dabei nur auf privat genutzte Wohnungen oder Häuser, dabei lässt sich diese Technik auch auf gewerblich genutzte Objekte übertragen. So können z.B. Hotels oder Krankenhäuser die Heizung von nicht genutzten Zimmern ein oder ausschalten, sobald ein Gast oder ein Patient das Zimmer betritt. Dies spart Energie und senkt zudem die Kosten.

Die Kostenersparnis nur ein Vorteil von Smart Home oder Smart Environments.

- **Sicherheit**
  - **Zugangs- und Schließsysteme:** Schlüssel-loser Zugang, Personenerkennung.
  - **Überwachung:** Unbekannte Personen in der Wohnung, die sich unberechtigt Zugang verschaffen haben, werden der Polizei und dem Besitzer gemeldet.
  - **Notfallfunktionen:** Bei einem Unfall in der Wohnung wird automatisch der Krankenwagen gerufen und die Tür aufgemacht, sobald die Sanitäter vor Ort sind.
- **Komfort**
  - **Unterstützende Funktionen des Smart Home/Environments** bei Personen mit Behinderung oder für ältere Personen.
  - **Einfache und zentrale Benutzerschnittstelle** zur Steuerung von vernetzten Elementen, wie z.B. Heizung oder Fenster.
  - **Elektronische Einkaufslisten:** Das Haus merkt welche Produkte fehlen oder abgelaufen sind.
  - **Anpassbare Systeme:** Das Haus merkt sich die Temperatur bei der sich der Bewohner zu Hause wohlfühlt und heizt die Räume in denen er sich aufhält auf genau diese Temperatur.
  - **Benachrichtigung bei anstehenden Terminen**
  - **Automatische Weckfunktion**, die sich anhand von Termine anpasst.
- **Kosten**
  - **Intelligentes Heizen oder Lüften**
  - **Automatisches ausschalten von Licht**, bei verlassen des Raumes

Je mehr Teilsysteme mit einander agieren können, desto mehr Möglichkeiten ergeben sich daraus. Um dies zu ermöglichen wird an vielen verschiedenen Teilsystemen geforscht. Somit ergeben sich Teilsysteme aus den verschiedensten Bereichen, wie z.B. Heizungssysteme, Klimasysteme, Beleuchtungssysteme oder auch Umfeld Systeme. Bei dem letzteren ist z.B. das bewässern des Rasens gemeint.

Zu diesen Teilsystemen kommen aber auch noch Bewohnerbezogene Systeme, wie Kommunikation oder Multimedia Geräte. Diese können in schon manchen Laborumgebungen als Universalfernbedienung eingesetzt werden, wie z.B. das Smartphone oder Tablet.

Der Ist-Zustand sieht momentan etwas anders aus, es fehlen bislang Standards für Smart

Home/Environments. Und so existieren zurzeit nur Insellösungen, wie z.B. RWE Smart Home[1] oder Telekom Smart Home[2]. Beide Unternehmen sind bestrebt neue Partner zu gewinnen und ihr Angebot etwas vielfältiger zu gestalten. Beispiele sind Kooperationen zwischen RWE und Philips bzw. Samsung, zur Steuerung von Licht oder Videoüberwachung. Dennoch ist das momentan von geringen Umfang.

Smart Home als Forschungsumgebung

Solche Forschungsumgebungen sind das Living Place[3] an der HAW, inHaus[4] am Fraunhofer-Institut in Duisburg uvm. Hierbei werden eigene Smarte Objekte entwickelt und in die Forschungsumgebungen implementiert.

Jene Implementierung erfolgt über eigens entwickelte Schnittstellen. Es wird schnell sichtbar, dass mit steigender Komplexität eines Systems für die Steuerung eines Smart Home, die Konfiguration immer schwieriger wird. Dies wird besonders deutlich, wenn ein Entwickler einen Ablauf für ein intelligentes Lüften oder Heizen erstellen soll. Um dies zu bewerkstelligen muss sich der Entwickler nicht nur mit der im Smart Home eingesetzten Entwicklungsumgebung auskennen, sondern auch mit der Infrastruktur des Smart Home.

Für eine Verbesserung bedarf es einer Entwicklungsumgebung, für Softwareentwickler, in der solche Abläufe schnell modelliert und getestet werden können. An diesem Ansatz setzt diese vorliegende Bachelorarbeit an. Zum einen lassen sich dadurch Abläufe leichter Abbilden und zum anderen sind die Entwicklungskosten dafür geringer, je größer ein Szenario ist.

Mit einer domänenspezifischen Sprache (vgl. Abschnitt 2.3), die explizit für die Domäne Smart Home entwickelt worden ist, kann ein Softwareentwickler mit dem Kunden zusammen ein Szenario entwickeln. Es muss zum einen für den Bewohner, bei der Konzeption, leicht zu verstehen sein und zum anderen für den Softwareentwickler kostengünstig zu entwickeln sein.

[1] RWE Smart Home - (RWE, vgl.)
[2] Telekom Smart Home - (Telokom, vgl.)
[3] Living Place HAW - (von Luck u. a., 2010, vgl.)
[4] inHaus - (inHause, vgl.)

2.3 Domain Specific Languages

2.3.1 Einführung

Bei der Entwicklung der Computersprachen haben sich zwei verschiedene Ansätze durchgesetzt, wobei auf dem letzteren bis zum heutigen Zeitpunkt der Fokus liegt.

Der erste Ansatz wurde von der Vorstellung getrieben eine Universalsprache zu entwickeln die allen Anforderungen gerecht wird. Solche Sprachen werden „General-purpose-languages“ (GPLs) genannt, dessen bekannteste Vertreter Java, C# und C++ sind. Dennoch werden immer neue entwickelt und die bestehenden weiterentwickelt. Doch seit einiger Zeit verlagert sich die Aufmerksamkeit der Forschung auf die sogenannten „domain-specific-languages“ (DSLs). Das Konzept von DSLs ist aber nicht neu. Ein bekannter Vertreter einer DSL ist zum Beispiel die BNF aus dem Jahr 1959. Das Augenmerk hierbei liegt auf nur einer konkreten fachlichen Domäne. Dabei soll die DSLs nur die Probleme innerhalb der Domäne darstellen können und nichts was außerhalb der Domäne liegt.

Es existiert keine hundertprozentig scharfe Definition dafür, was eine DSL ist. Im Laufe der Jahre hat sich der Begriff DSL und auch die Bedeutung von DSLs weiterentwickelt, aber eine wirklich fixe akademische Definition wurde hierbei nie getroffen. Fowler (2010)

Eine Annäherung an die Definition einer DSL kann anhand der durch Fowler verwendeten Merkmale, wie folgt, beschrieben werden.

- **Programmiersprache:** Eine DSL soll zwar für einen Menschen leicht verständlich sein, dennoch muss sie von einem Computer interpretierbar sein.
- **Sprachliche Natur:** Eine DSL sollte den Charakter einer Sprache haben und einen gewissen Sprachfluss vorweisen. Hierbei sollte die Ausdrucksstärke nicht nur auf Basis der einzelnen Ausdrücke gegeben sein, sondern auch durch die Art wie sie miteinander kombiniert werden.
- **Beschränkte Ausdruckskraft:** Eine DSL sollte eine minimale Menge an Sprach-Elementen haben, im Gegensatz zu einer GPL. Dabei sollen nur Sprachelemente verwendet werden die zur Unterstützung der Domäne notwendig sind. Dies sollte eine leichtere Erlernbarkeit ermöglichen.
- **Fokus auf Domäne:** Eine DSL sollte den Fokus auf eine Domäne legen und diese so gut wie möglich beschreiben.

2.3.2 Grundlagen

Sprachen die in diese Kategorisierung fallen – und somit domänenspezifische Sprachen sind – können in die folgenden drei Arten unterteilt werden.

- **Interne DSL:** Diese basiert auf einer Trägersprache, meist einer GPL. Hierbei wird die DSL in den geschriebenen Code eingebettet, z.B. mittels einem Skript der von einer GPL ausgeführt wird. Ein Vertreter so einer DSL ist das Test-Framework „Cucumber“, hierbei wird eine Datei geschrieben Abbildung 2.1, die für einen Domain-Spezialisten leicht lesbar ist. Diese wird dann mittels Plug-In für JUnit eingelesen und ausgeführt. Der Test heißt hier „Serve coffee“, eine textuelle Erklärung des Testes wir darunter aufgeschrieben. Das „Scenario“ markiert den eigentlichen Test-Anfang. Mit „Given“ werden die Variablen oder Funktionen im Test initialisiert, so wie bei der „@Before“ Methode. Mit „And“, „When“ und „Then“ werden die einzelnen, zum Test gehörenden Methoden, gestartet.

- **Externe DSL:** Eine externe DSL liegt außerhalb der anderen Sprache und soll komplett eigenständig funktionieren. Diese hat meist eine eigene Syntax und Grammatik oder kann die Syntax einer dritten Sprache verwenden (XML ist ein häufiges Beispiel so einer Sprache). Zu den Vertretern der externen DSLs gehören zum Beispiel SQL oder reguläre Ausdrücke.

- **Language Workbench:** Martin Fowler hat in einem seiner Artikel von 2005 die Bezeichnung „Language Workbench“ geprägt. Sie beschreibt eine Umgebung, in der domänenspezifische Sprachen und ihre Toolunterstützung entwickelt werden können. (Fowler) Dabei handelt es sich um Sprachen, die innerhalb einer speziell dafür konzipierten Anwendung entwickelt werden, und somit über entsprechende Editoren verfügen. Dieses soll das Arbeiten erleichtern, da hier Objekte oder Abbildungen für die Lösung des Problems benutzt werden. Der dritte Ansatz ist auch der mit dem sich diese Bachelor Thesis auseinandersetzt.

```
Feature: Serve coffee
    Coffee should not be served until paid for
    Coffee should not be served until the button has been pressed
    If there is no coffee left then money should be refunded

  Scenario: Buy last coffee
    Given there are 1 coffees left in the machine
    And I have deposited 1$
    When I press the coffee button
    Then I should be served a coffee
```

Abbildung 2.1: Beispiel Skript für Cucumber

2.3.3 Bestandteile von Domain Specific Languages

Im vorgegangenen Kapitel wurde erläutert was DSLs sind. In diesem Kapitel sollen die Bestandteile einer DSL, anhand des Schemas in Abbildung 2.2 erläutert werden. Eine DSL besteht zunächst aus drei essentiellen Elementen:

- **dem Metamodel:** Dieses wird auch als abstrakte Syntax bezeichnet. Das Metamodel definiert welche logische Struktur die Sprache zugrunde liegt. Dabei wird definiert welche Entitäten in der Domäne vorhanden sind und in welcher Beziehung sie zu einander stehen. Im Metamodell sind auch noch Constraints enthalten, diese erzwingen das Verhalten und die Regeln der Domäne. Constraints sind außerdem in der Lage Dinge abzubilden die nicht in der Modellierung definiert sind, wie z.B. die Eindeutigkeit von Attributnamen.
- **konkrete Syntax der Sprache:** Auch Grammatik genannt. Im Gegensatz zum Metamodell legt die Grammatik fest in welcher Form etwas ausgedrückt wird. Dies kann bei einer textuellen DSL die Grammatik sein oder die Notation einer Modellierungssprache.
- **und der Semantik:** Diese liegt meinst in Form eines Dokumentes vor und beschreibt die DSL.

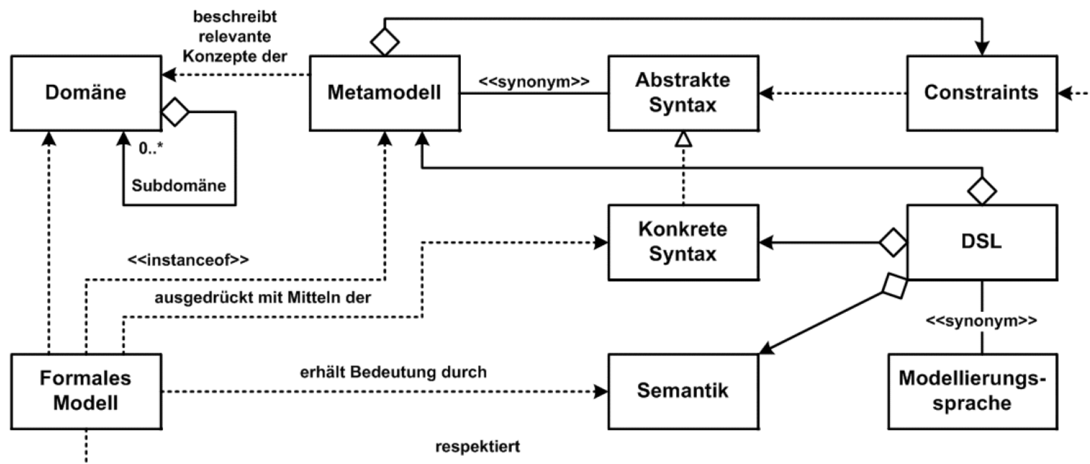

Abbildung 2.2: DSL Schema (vgl. Voelter (2009))

2.3.4 Vor- und Nachteile

Einer der größten Vorteile von DSLs ist wohl die Verwendbarkeit, dieser von Domainspezialisten ohne nennenswerte Programmierkenntnisse, unter der Voraussetzung das ein gutes Maß an Abstraktion gefunden worden ist. Dieses ermöglicht eine schnelle und effiziente Implementierung von Erweiterungen in die Domäne. Des Weiteren ist das Testen auf einer Abstraktionsebene einfacher und Fehler können so, entweder von einem Tool oder Benutzer, schneller gefunden und eliminiert werden.

Nachteile sind zum einen die hohen Entwicklungskosten und zum anderen steigt die Anzahl der Sprachen die in einem Projekt benutzt werden. Dies ist nur ein kleiner Ausblick auf die Vor- und Nachteile. Dennoch sollten diese nicht außer Acht gelassen werden.

2.4 Entitäten im Livingplace

2.4.1 Einleitung

Das Living Place Hamburg ist ein Labor an der HAW in Form einer großen Loft-Wohnung. In der Fragestellungen zu den Themen Smart-Home, Urban-Living, Human-Computer-Interaction und New Storytelling bearbeitet werden. (von Luck u. a., 2010)

Im Laufe der Zeit hat sich der Aufbau des Systems zur Steuerung des Living Place geändert

(Eichler, 2014). Die aktuellste Version dieses Systems ist die Middleware Abbildung 2.3.

Diese basiert auf der Theorie von Multiagentensystemen. Da keine allgemeingültige Definition von Agenten akzeptiert ist Wooldridge (2002), werden diesen, je nach Anwendungsfall, unterschiedliche Eigenschaften zugesprochen. In der Middleware übernehmen Agenten die Aufgabe bestimmte Elemente, wie z.B. Fenster oder Gardienen, im Living Place zu steuern oder deren Status abzufragen. Ein Agent kann aber auch komplexe Berechnungen durchführen oder auch andere Agenten steuern bzw. abfragen.

Das Ziel dieser Arbeit ist die Entwicklung einer Domänenspezifischen Sprache. Diese soll den Entwicklern es ermöglichen schneller und vor allem einfacher Szenarien, in dieser Laborumgebung, zu entwickeln und zu testen.

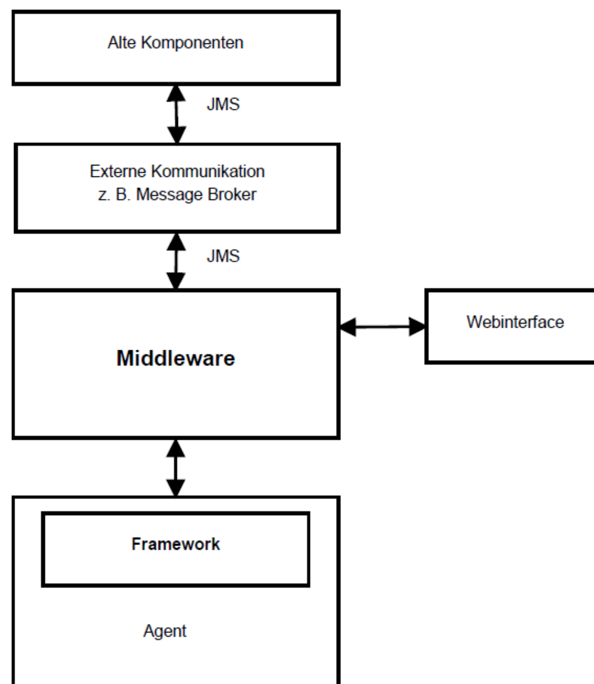

Abbildung 2.3: Middleware - Architektur (Eichler, 2014, S.40)

2.4.2 Agenten

Ein Agent dient nicht nur zur Ausführung von Anweisungen, die er bekommt, er kann auch selber Anweisungen verschicken oder seinen Status weitergeben. Dies wird im Living Place mittels eines Publish – Subscribe Schemas realisiert. Somit können alle Agenten in dem jewei-

ligen Request Channel eines Agenten die Anweisungen veröffentlichen oder in dem Answer Channel den Status auslesen. Dieser wird nur angezeigt wenn mindestens ein Agent diesen Channel abonniert hat und eine Anfrage über den Status versendet hat.

Einige Agenten unterstützen das Ask-Pattern, was es ermöglicht den Agenten abzufragen ohne seinen Channel abonniert zu haben, als Konsequenz wird die Antwort nur an das anfragende Objekt versendet. Da dieses Pattern nicht flächendeckend implementiert ist, muss das Publish–Subscribe Schema benutzt werden.

Hierfür wird das Aktor-Framework von Akka[5] verwendet. Dieses unterstützt Java und Scala[6] als Entwicklungssprachen, wohingegen im Living Place Scala verwendet wird. Der Vorteil liegt in der Lesbarkeit in Bezug auf die eingesetzten Send oder Ask Pattern und die Erstellung von so genannten Case Classes (siehe. Abbildung 2.4). Scala ist somit die bessere Wahl, da Case Classes sich sehr gut für den Versand von Informationen zwischen den Agenten.

Des Weiteren ist die Definition von Attributen oder Objekten implizit. Es können zwar einige Zeilen Code eingespart werden, der Einstieg ist für Java-Entwickler etwas komplizierter. Auch die Tatsache, dass Listen zur Laufzeit *immutable* sind und daher nicht verändert werden können, kann den Einstieg erschweren.

Um diese Aufgabe zu realisieren sind in einem Agenten die Funktionen implementiert die für den Zugriff auf einen Aktor oder Sensor benötigt werden. Dennoch kann ein Agent nicht nur dafür verwendet werden um Aktoren oder Sensoren anzusprechen oder zu steuern. Sie können auch komplexe Berechnungen durchführen oder andere Agenten steuern.

[5]Akka Dokumentation - (Inc, 2015)
[6]Programming in Scala - (Odersky u. a., 2010)

```
case class Weather(id: Long,
  weatherSmall: WeatherSmall,
  tempMinValue: Float,
  tempMaxValue: Float,
  pressureValue: Float,
  pressureUnit: String,
  humidityValue: Float,
  humidityUnit: String,
  windSpeedValue: Float,
  windSpeedUnit: String,
  windDirValue: Float,
  windDirCode: String,
  weatherDescription: String,
  sunrise: String,
  sunset: String)
```

Abbildung 2.4: Weather Case Class, mit allen Informationen die der Wetter-Agent versendet

Aktoren

Das Living Place hat eine Vielfalt an verschiedenen Aktoren, die verschiedenste Aufgaben übernehmen. So steuert ein Aktor die Fenster, ein anderer das Licht.

Es wird die Schnittstelle eines Aktor genutzt, so dass die Anweisung direkt an die Schnittstelle geschickt werden kann. Ähnlich wie bei dem Light-Agent wird eine TCP Verbindung zur Schnittstelle hergestellt und der gewünschte Befehlt von dem Agenten an den Aktor Controller gesendet. Dabei ist es je nach Controller wichtig das sich der Agent den Zustand merkt, damit die Fehleranfälligkeit reduziert wird.

Zusammenfassend wird der Aufbau eines Agenten, der einen Aktor steuert, wie in Abbildung 2.4 dargestellt.

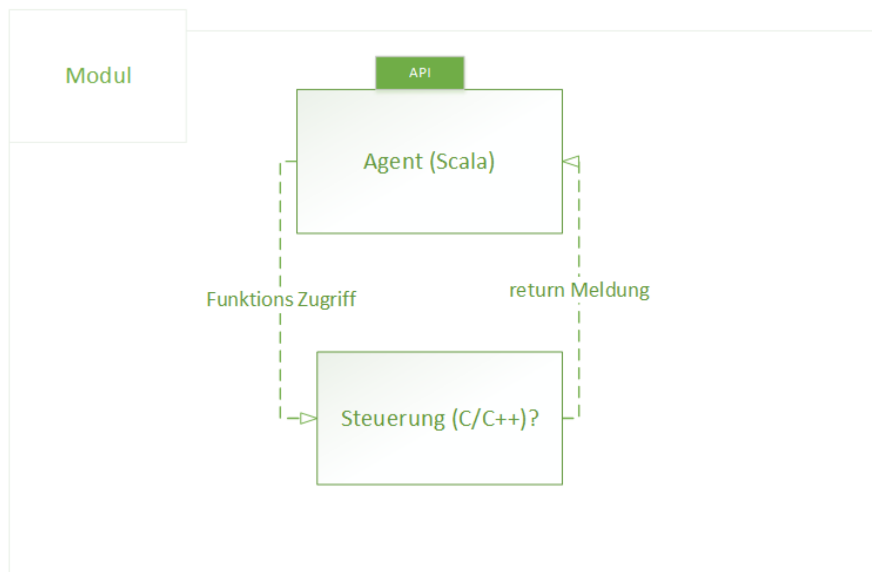

Abbildung 2.5: Modul Komponente

Sensoren

Es gibt Sensoren die den Status eines Aktoren auslesen können. Darüber hinaus existieren auch Temperatur Sensoren oder auch Druck Sensoren. Bei dem Temperatursensor-Agenten erfolgt der Zugriff auf den Sensor mittels einer Webschnittstelle, diese liefert Daten im XML-Format.

Dabei werden die Zeitintervalle in denen die Sensorinformationen abgefragt werden sollen im Agenten festgelegt. Somit kann ein anderer Agent auf dem Antwort Channel darauf warten bis die nächsten Temperaturinformationen veröffentlicht werden oder eine Anfrage an den Temperatursensor-Agenten schicken um präzisere Daten zu erhalten.

Ein weiteres Beispiel eines solchen Sensors ist das Kinect2Agent-Projekt[7]. Hierbei werden die Informationen, die von der Kinect empfangen werden, in dem State Channel des Agenten veröffentlicht und für alle Abonnenten zur Verfügung gestellt.

2.4.3 Timer

Der Timer ist ein theoretisches Konstrukt, wobei ein Timer ein Agent sein kann, der eine bestimmte Zeitperiode wartet bis er eine Aktion durchführt. Es kann aber auch ein Agent sein

[7] http://devsupportgit.informatik.haw-hamburg.de/devsupport-development/kinect2-agent

der den Startzeitpunkt in einen Kalender einträgt und vom Kalender eine Nachricht bekommt, sollte der Startzeitpunkt gekommen sein.

2.4.4 Zusammenfassung

In den vorherigen Abschnitten wurden die einzelnen Entitäten vorgestellt die im Living Place zum Einsatz kommen und wichtig für die Prozess Engine sind. An diesen Entitäten soll im nächsten Kapitel die eigentliche Prozess Engine entworfen werden.

Die bisher definierten Entitäten:

- **Agenten:** Können zum Empfangen oder auch zum Versenden von Nachrichten dienen, aber auch zur Berechnung von komplexen oder einfachen Aufgaben. Dieser steuert Aktoren oder fragt Sensoren über die von den Aktoren und Sensoren angebotene Schnittstelle ab.
- **Aktoren:** Die Aktoren sind die physikalische Steuerung im Living Place, so werden diese zum Beispiel dafür genutzt um Fenster zu öffnen oder zu schließen.
- **Sensoren:** Die Sensoren geben den aktuellen Status oder Zustand von einem Aktor aus. Sensoren können sich aber auch nicht nur auf einen Aktor beziehen, so kann ein Sensor auch die Temperatur im Living Place ausgeben.
- **Timer:** Eine theoretisches Konstrukt das es ermöglichen soll, Aufgaben um eine bestimmte Uhrzeit an einem Bestimmten Datum zu starten. Aber auch ein Szenario an einer Stelle für einen bestimmten Zeitraum zu pausieren.

2.5 Szenarien

2.5.1 Zeitlich vordefiniertes Lüften anhand von Wetterdaten

Ablauf

In diesem Abschnitt wird der Ablauf des Szenarios analysiert. Dabei muss geklärt werden welche Aktionen zuerst durchgeführt werden müssen und auch, wenn vorhanden, die Vorbedingungen analysiert und definiert werden.

Damit ein effizientes Raumklima geschaffen werden kann, muss die Wohnung, wenn möglich, täglich gelüftet werden. Hierbei muss auch auf das Wetter geachtet werden, um das Lüften möglichst effizient, aber auch sicher, zu gestalten. Anhand der Wetterdaten kann entschieden werden welche Fenster zu öffnen sind, damit kein Durchzug entsteht. Zusätzlich ist die

Windstärke, die ebenfalls aus den Wetterdaten zu entnehmen wäre, von Bedeutung. Sollte die Windstärke zu hoch sein zum Lüften, kann es zur Beschädigung von Einrichtungsgegenständen führen. Daher sollte ein Standartwert gesetzt werden, anhand dessen entschieden werden soll, ob das Lüften zu diesem Zeitpunkt sinnvoll ist. Ein effizientes Lüften beinhaltet auch das ausschalten der Heizung, den diese würde beim Lüften nur unnötig Energie verbrauchen.

Nachdem die Fenster geöffnet sind und die Heizung ausgeschaltet wurde, muss der Vorgang eine Zeitlang warten. Dabei soll die Wartezeit frei definierbar sein, somit kann der Benutzer selbst entscheiden wie lange er Lüften möchte. Am Ende dieser Wartezeit sollen die Fenster wieder geschlossen und die Heizung wieder eingeschaltet werden.

Es lässt sich folgende Reihenfolge definieren, wie in Abbildung 2.6 Abbildung veranschaulicht.

1. **Startzeit:** Tägliches starten, anhand eines Startzeitpunkts (Timer).
2. **Wetterdaten:** Abfragen der aktuellen Wetterdaten.
3. **Verarbeitung der Wetterdaten und Entscheidung:** Sollte die Windstärke zu hoch sein, so wird das Szenario beendet.
4. **Entscheidung der zu öffnenden Fenster:** Anhand der Wetterdaten werden die zu öffnenden Fenster ermittelt.
   - Fenster sollen gegenüber liegen.
   - Es darf kein Durchzug entstehen.
5. **Heizung deaktivieren**
6. **Wartezeit:** An dieser Stelle pausiert das Szenario für die eingestellte Wartezeit
7. **Fenster schließen:** Alle Fenster werden geschlossen.
8. **Heizung aktivieren**

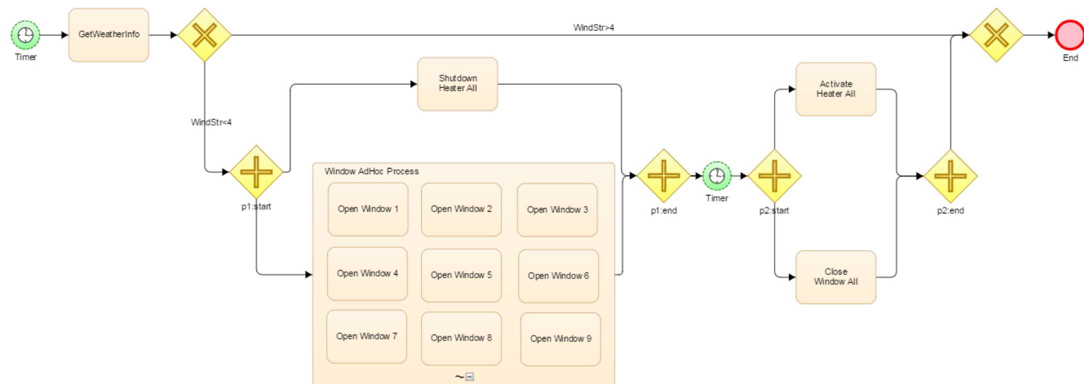

Abbildung 2.6: Zeitlich vordefiniertes Lüften anhand von Wetterdaten

Properties

Einige Elemente, die in Abbildung 2.6 beschrieben werden, erfordern einen Trigger zum Starten oder eine Bedingung. Diese soll der Benutzer über die Properties (vgl. Abbildung 2.7) einstellen können. Jenes soll dem Domain Spezialisten ermöglichen Bedingungen an das Ausführen von Tasks zu setzen.

Besonders wichtig sind solche Bedingungen bei Ad-Hoc Containern. So kann der Domain Spezialist einstellen wann ein Task ausgeführt werden soll oder welche Abbruchbedingung der Ad-Hoc Container hat.

Hat ein Task eine Bedingung, also ein Property, so wird dieser erst ausgeführt wenn die Bedingung zutrifft. Es können mehrere Bedingungen an einen Task übergeben werden. Solche Bedingungen könnten für einfache Tasks wie folgt aussehen: "120<Windrichtung<160". Somit wird der Task nur gestartet wenn die Windrichtung zwischen 120° und 160° liegt. Bei den Bedingungen muss zwischen zwei Arten von Tasks unterschieden werden. Die einen, die sich in einem Ad-Hoc Container und die, die sich außerhalb von einem Ad-Hoc Container befinden. Wenn der Task Außerhalb des Ad-Hoc Container ist wird der Task erst gestartet wenn die Bedingung zutrifft. Trifft die Bedingung nicht zu wartet der Prozess an der Stelle solange, bis diese zutrifft. Im Ad-Hoc Container wird nur solange gewartet wie der Ad-Hoc Container läuft.

Somit sollten die Laufzeiten oder auch die Abbruchbedingungen für den Ad-Hoc Container

als Property gesetzt werden. Dies könnte folgende Form haben: ’Solange:Timeout!=5000ms’ und/oder ’Abbruch:Windstr>4’.

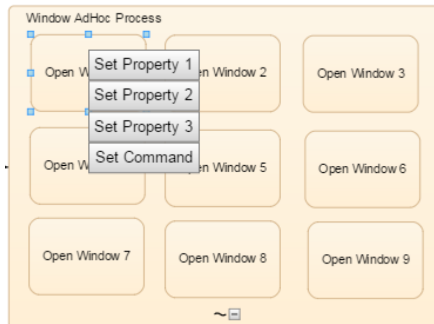

Abbildung 2.7: Setzten der Properties für einen Task

2.5.2 Zeitlich vordefiniertes Lüften

Ablauf

Dieses Szenario ist sehr stark an das vorangegangene angelehnt. Der Unterschied bei diesem Szenario ist, dass hier ein Durchzug erzeugt wird, da alle Fenster geöffnet werden. Dadurch ist das Lüften effektiver und die Wartezeit geringer.

Es ist bei diesem Szenario aber umso wichtiger auf die Windstärke zu achten und auch den Standard-Wert, ab welchem gelüftet werden soll, geringer anzusetzen als bei dem vorangegangenen Szenario. Also lässt sich die Reihenfolge des Szenarios wie folgt darstellen (vgl. Abbildung 2.8):

1. **Startzeit:** Tägliches starten, anhand eines Startzeitpunkts (Timer).
2. **Wetterdaten:** Abfragen der aktuellen Wetterdaten.
3. **Verarbeitung der Wetterdaten und Entscheidung:** Sollte die Windstärke zu hoch sein, so wird das Szenario beendet.
4. **Öffnen der Fenster:** Es werden alle Fenster geöffnet.
5. **Heizung deaktivieren**
6. **Wartezeit:** An dieser Stelle pausiert das Szenario für die eingestellte Wartezeit
7. **Fenster schließen:** Alle Fenster werden geschlossen.

8. **Heizung aktivieren**

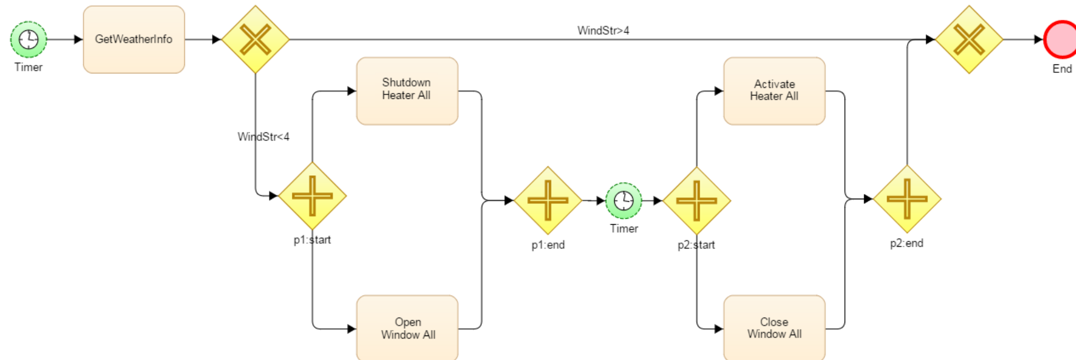

Abbildung 2.8: Zeitlich vordefiniertes Lüften

Der Unterschied zwischen dem vorangegangenen Szenario ist der Wegfall des Ad-Hoc Containers. Somit wird an alle Fenster das Kommando 'Fenster auf' gesendet.

2.5.3 Event-Basiertes Lüften

Ablauf

Beim Event-Basierten Lüften soll der Vorgang erst gestartet werden, wenn es Notwendig ist. Die Notwendigkeit muss von dem Domain Spezialisten definiert werden. Eine Möglichkeit wäre es zu lüften, wenn es stickig ist oder einfach nur zu warm. Diese Information kann von Sensoren im Living Place bezogen werden.

Somit ist eine konzeptionelle Frage zu beantworten: Wird der Prozess erst gestartet, z.B. von einem Agenten, wenn der Sensor einen vordefinierten Wert erreicht oder läuft der Prozess die ganze Zeit und wird und fragt den Sensor selber ab?

Beide Varianten sollten dem Entwickler möglich sein. Damit aber die zweite Variante ermöglicht werden kann, muss dem Domain Spezialisten diese Möglichkeit in der domänenspezifischen Sprache gegeben werden. Dies kann durch einen 'Conditional Start' ermöglicht werden. Hierfür muss der Domain Spezialist an den 'Conditional Start' eine Bedingung anhängen können (vgl. Kapitel 2.5.1).

Die Reihenfolge sieht wie folgt aus (vgl. Abbildung 2.9):

1. **Conditional Start:** Wird gestartet wenn eine oder mehrere Bedingungen erfüllt sind.

2. **Öffnen der Fenster:** Es werden alle Fenster geöffnet.

3. **Heizung deaktivieren**

4. **Wartezeit:** An dieser Stelle pausiert das Szenario für die eingestellte Wartezeit

5. **Fenster schließen:** Alle Fenster werden geschlossen.

6. **Heizung aktivieren**

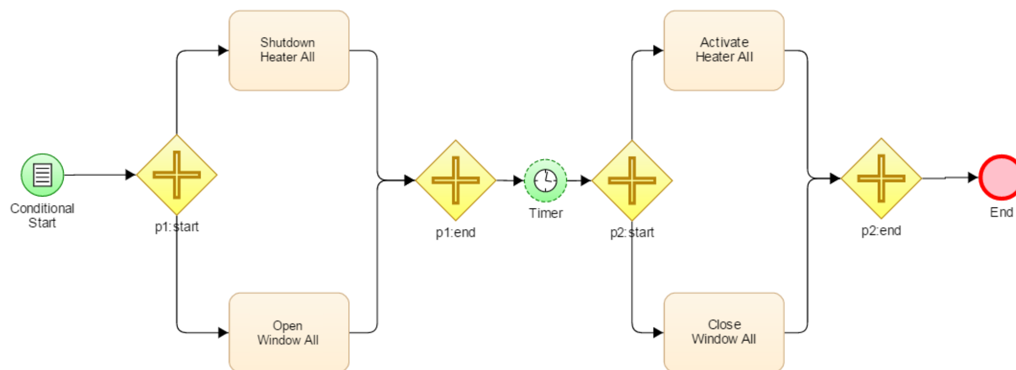

Abbildung 2.9: Event-Basiertes Lüften

2.5.4 Event-Basiertes Lüften anhand von Wetterdaten

Ablauf

Dieses Szenario ist eine Kombination aus dem Szenario 2.5.1 und 2.5.3. Das System soll zum einen auf Raumqualität und zum anderen auf die Windrichtung achten, damit kein Durchzug entsteht. Um die dafür nötigen Einstellen vornehmen zu können, kann der Domain Spezialist wieder auf die Properties aus dem Kapitel 2.5.1 zurückgreifen.

1. **Conditional Start:** Wird gestartet wenn eine oder mehrere Bedingungen erfüllt sind.

2. **Entscheidung der zu öffnenden Fenster:** Anhand der Wetterdaten werden die zu öffnenden Fenster ermittelt.

- Fenster sollen gegenüber liegen.
- Es darf kein Durchzug entstehen.

3. **Heizung deaktivieren**
4. **Wartezeit:** An dieser Stelle pausiert das Szenario für die eingestellte Wartezeit
5. **Fenster schließen:** Alle Fenster werden geschlossen.
6. **Heizung aktivieren**

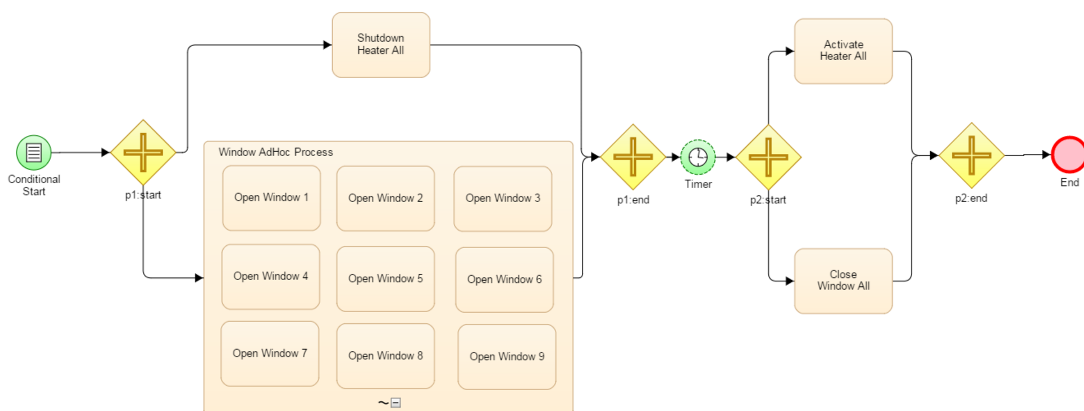

Abbildung 2.10: Event-Basiertes Lüften anhand von Wetterdaten

2.5.5 Angesprochene Entitäten

Anhand der Szenarien kann festgestellt werden welche der Entitäten angesprochen werden und in welchem Zusammenhang dies geschieht.

In Szenario 2.6 und 2.8 werden die Wetterdaten[8] direkt nach dem Start abgefragt und anhand dieser Entschieden, ob ein Lüften möglich ist. Sollte die Windstärke zum Abfragezeitpunkt zu hoch sein, wird das Szenario beendet ohne zu lüften, andernfalls wird fortgesetzt. Aber auch in Szenario 2.10 werden Wetterdaten benötigt, anhand dessen die Fenster bestimmt werden sollen, die zu öffnen sind ohne dabei einen Durchzug zu erzeugen. Dies geschieht auch in

[8]Webseite der Wetter-API http://devsupportgit.informatik.haw-hamburg.de/po1415/weather-api/tree/master; letzter Zugriff: 31.01.2016

Szenario 2.6.

Zusätzlich zu den Wetterdaten werden auch die Einzelnen Agenten angesprochen die, die Fenster[9] und die Heizung[10] steuern. Das wird in den Szenarien unterschiedlich gelöst. In Szenario 2.6 und 2.10 werden die Fenster einzeln angesprochen, dabei werden diese anhand der 'FensterID' unterschieden. In Szenario 2.8 und 2.9 werden alle Fenster angesprochen, dies Geschieht über die vom Agenten angebotene GruppenID. Hierbei wird es dem Entwickler ermöglicht z.B. alle Fenster zu öffnen oder zu schließen oder nur die Fenster in der Küche zu bedienen.

Über diese Gruppenfunktion wird auch die Heizung gesteuert. In allen Szenarien werden hierzu entweder alle Heizungselemente eingeschaltet oder ausgeschaltet.

Es werden demzufolge folgende Entitäten benötigt und angesprochen:

- Wetter-API
- Fenster-API
  - Für jedes Fenster eine Entität
  - Für jede Gruppe eine Entität
- Heizung-API
  - Für jede Gruppen eine Entität

2.5.6 Zusammenfassung

In den vorherigen Abschnitten wurden drei mögliche Szenarien vorgestellt und im Detail erläutert. Anhand dieser Szenarien wurden die benötigten Entitäten für die Erfüllung dieser Szenarien ermittelt und deren Umfang bestimmt.

Dazu gehörten Szenarien dessen Start von einem Timer abhängt, aber auch Szenarien die Event-Basiert gestartet werden. Abschließend wurden die Richtlinien für einen Event-Basierten Start eines Prozessen definiert. Jene Richtlinien wurden von Ad-Hoc Containern und Tasks abgeleitet und auf den in Abschnitt 2.5.3 beschriebenen Conditional-Start angewendet.

[9] Webseite der Fenster-API http://devsupportgit.informatik.haw-hamburg.de/po1415/window-agent/tree/master; letzter Zugriff: 31.01.2016

[10] Webseite der Heizung-API http://devsupportgit.informatik.haw-hamburg.de/po1415/heater-api/tree/master; letzter Zugriff: 31.01.2016

2.6 Anforderungen

2.6.1 Einleitung

Die Szenarien aus dem vorherigen Abschnitt werden hier als Hilfestellung zum Erarbeiten der Anforderungen dienen.

Hierbei werden nicht nur die Anforderungen an die Entwicklung der Sprache definiert. Es sollen auch die dafür benötigten Werkzeuge und Komponenten definiert werden. Das vollständige Potenzial der vollständig ausgereiften Lösung dieser domänenspezifischen Sprache soll in der Vision vermittelt werden. Im Anschluss soll ein Minimal-Entwurf die Anforderungen formulieren, die für eine Realisierung der Vision notwendig sind. Diese Anforderungen sollen in den folgenden Kapiteln 3 und 4.1 aufgegriffen werden.

2.6.2 Vision

Domänenspezifische Sprache

Die domänenspezifische Sprache soll ein möglichst großes Spektrum an Agenten der Middleware abbilden und ansprechen können. Aber auch die Einbindung neuer Agenten, sowie neuer Funktionen möglichst einfach für einen Domain Spezialisten machen.

Damit dies von der domänenspezifischen Sprache ermöglicht wird, müssen die Kernkonzepte der Domäne, die in diesem Fall die Middleware ist, abgebildet werden. Die einfache Lesbarkeit und das schnelle Verständnis sind Vorteile einer DSL. Daher sollten die eingebauten Elemente jeweils nur zur Beschreibung von genau einem Konzept in der Domäne genutzt werden.

Um die Erweiterung und auch die Arbeit an der DSL zu unterstützen und zu erleichtern, müssen Werkzeuge vorhanden sein. Die sowohl das Erstellen, Bearbeiten und das Testen der Sprache ermöglichen, aber auch die Erweiterung der Sprache um neue Elemente unterstützen.

Daraus resultieren folgende Eigenschaften der domänenspezifischen Sprache:

- **Konformität**
- **Orthogonalität**
- **Unterstützung**
- **Erweiterbarkeit**

Entwicklungsumgebung

Die Grundidee zur Entwicklung dieser DSL besteht darin einem Entwickler ein Werkzeug zur Verfügung zu stellen, mit dem er die Abläufe im Living Place visuell entwickeln und testen kann. Somit ist das Ziel die Entwicklung eines Editors, indem der Domain Spezialist nicht nur den Ablauf eines Prozesses planen kann, sondern auch Testen kann.

Um den Zugang für jeden Domain Spezialisten so einfach wie möglich zu gestalten, sollte eine Studie durchgeführt werden. Hierbei können aktuelle Werkzeuge, wie z.B. Visio[11] oder Visual Paradigm[12], zur Erstellung von Prozessen als Richtwert genutzt werden. So soll zum einen ein Werkzeug entstehen, dessen Grad an Komplexität möglichst gering ist, aber dennoch möglichst viele Kernelemente im Living Place abdeckt.

Damit das Arbeiten mit dem Editor möglichst reibungslos abläuft, soll die Entwicklung und das Testen direkt im Editor angeboten werden. Dies erlaubt dem Domain Spezialisten seine Entwicklung direkt in der eingebauten Test-Umgebung, die sich im Editor befinden sollte, zu testen.

Ein leichter Einstieg in die Arbeit mit dem Editor könnte mit eingebauten Test-Prozessen ermöglicht werden. Zusätzlich zu diesen Code-Schnipseln, sollte auch eine umfangreiche Dokumentation vorhanden sein. Die zum einen die wichtigsten Funktionen erläutert und zum anderen die Grundregeln für einen fehlerfreien Ablauf beschreiben.

Test-Umgebung

Wie im vorherigen Abschnitt erwähnt, soll die Test-Umgebung in die Entwicklungsumgebung implementiert sein. Dabei sollen die erstellten Prozesse direkt gestartet werden können und in einer Dummy-Umgebung geprüft werden. Damit dies dem Domain Spezialisten ermöglicht wird, müssen Anfragen und Antworten an eine Middleware ähnliche Test-Umgebung gesendet und ausgelesen werden. Dabei soll der Domain Spezialist auch die Antwortet dieser Test-Umgebung beeinflussen können und somit auch eine fehlerhafte Antwort einstellen können. Es wird dem Domain Spezialist dadurch ermöglicht den erstellten Prozess in Hinblick auf die Fehlerverarbeitung zu testen.

Bevor der Test gestartet wird sollte die Test-Umgebung den Prozess auf bekannte Fehlermuster

[11]Webseite zu Visio http://products.office.com/de-de/visio; letzter Zugriff: 31.01.2016
[12]Webseite zu Visual Paradigm http://www.visual-paradigm.com/; letzter Zugriff: 31.01.2016

prüfen. Diese könnten zum einen eine fehlerhafte Verwendung der Elemente in der domänenspezifischen Sprache sein, zum anderen ein Fehler im Ablauf des Prozesses. Zusätzlich soll die Test-Umgebung auch auf Fehler die möglicherweise auftreten könnten hinweisen. Dies ist Notwendig wenn in einem parallelen Strang der Prozess auf die gleiche Instanz zugreifen würde.

Um diese Fehlerquellen direkt getestet werden können, soll die Test-Umgebung dem Domain Spezialisten ermöglichen, Bruchteile eines Prozesses zu testen. Es wird also nicht der ganze Prozess gestartet, sondern nur der Teil in dem z.B. die Warnung auftaucht. Das spart Zeit bei der Entwicklung und bei der Fehlersuche.

Bei allen Tests, die der Domain Spezialist ausführt, sollen die aktuellen abgefragten und gespeicherten Elemente angezeigt werden. Anhand dieser Elemente kann der Domain Spezialist erkennen, ob der Prozess richtig läuft oder ob es zu Fehlern gekommen ist. Hierfür könnte ein Debug-Modus eingebaut werden, in dem die einzelnen Schritte im Prozesses angezeigt und auf Knopfdruck zum nächsten Element gesprungen werden kann.

Zusätzlich sollen dem Domain Spezialisten auch die Ausführung von Prozessen im Zeitraffer, an einem bestimmten Datum oder um eine bestimmte Uhrzeit ermöglicht werden. Diese können Fehler aufdecken die unter Umständen nur zur bestimmten Uhrzeit oder an bestimmten Tagen auftauchen.

Produktiv-Umgebung

Die Produktiv-Umgebung soll in der Middleware laufen und sollte von einem dafür konzertierten Agenten gesteuert werden. Dafür soll ein Paket entwickelt werden, das von dem Agenten implementiert wird. Dieses Paket muss die eigentliche Prozess-Engine beinhalten.

Gleichzeitig kann der Agent sowohl für das Starten der Prozesse verantwortlich sein, als auch zur Überwachung dieser dienen. Ferner muss der Agent eine Schnittstelle besitzen, die es ihm ermöglicht Prozesse in einem bestimmten Format zu übergeben. Diese werden von dem Agenten entweder gleich gestartet oder für einen späteren Start vorgemerkt.

Auch eine Anbindung des Agenten an eine Datenbank ist für die Verwaltung sinnvoll, dadurch können Prozesse die Regelmäßig auftreten, in der Datenbank dauerhaft gespeichert und bei Bedarf gestartet werden.

Aber auch eine Schnittstelle nach außen, zum Beispiel für die Darstellung des Status von laufenden Prozessen, wäre sinnvoll und zu implementieren, wobei sie auch Fehlermeldungen

versenden können muss.

Zusätzlich sollte die Umgebung mehrere Modi unterstützen. Im normalen Modus kann demnach der Prozess wie geplant gestartet werden und auch wie geplant ablaufen. In einem Demo-Modus könnte der Domain Spezialist den Start eines Prozesses direkt auslösen und den im Zeitraffer ablaufen lassen.

2.6.3 Minimal-Entwurf

Da der Umfang der Vision den der Bachelorarbeit übersteigt, werden in dem Minimal-Entwurf nur die Bestandteile umgesetzt, welche die Durchführbarkeit des Vorhabens belegen. Die formulierten Anforderungen werden in den folgenden Kapiteln umgesetzt.

Anforderungen: domänenspezifische Sprache

Die Implementierung aller Agenten als eigenständiges Element der domänenspezifischen Sprache, würde den Rahmen dieser Bachelorarbeit weit übersteigen, weshalb hier eine Auswahl der Elemente getroffen wurde. Die dabei entwickelte Sprache soll jedoch die beschriebenen Eigenschaften aufweisen:

- **Unterstützung:** Es soll ein Werkzeug entwickelt werden, in dem der Domain Spezialist einfache Prozesse erstellen und in einer Test-Umgebung starten kann.
- **Erweiterbarkeit:** Dabei soll gezeigt werden das die Erweiterung der Sprache möglich ist.

Anforderungen: Entwicklungsumgebung

Die Entwicklungsumgebung soll neben den Kernelementen aus dem Living Place auch einige Grundelemente der Sprache darstellen können, wie beispielsweise Gateways, parallele oder exklusive Gateways, oder einen Timer-Start. Sie müssen für Softwareentwickler leicht zu verstehen und an erwarteter Stelle zu finden sein.

Zusätzlich muss die Entwicklungsumgebung die Aspekte **Unterstützung** und **Erweiterbarkeit** hervorheben.

Anforderungen: Test-Umgebung

Auch bei der Test-Umgebung übersteigt der Umfang aus der Vision den einer Bachelorarbeit. Dennoch sollen einige wichtige Bestandteile aus der Vision hier umgesetzt werden. Hierbei wird die die minimal Umsetzung der Test-Umgebung Fehlermuster erkennen, Prozesse ausführen und deren Ergebnisse anzeigen können.

Die Test-Umgebung soll dabei in die Entwicklungsumgebung implementiert und aus dieser gestartet werden können. Jenes soll den Entwickler dazu ermutigen nach jeder Änderung seines Prozesses, diesen auch gleich testen zu können.

2.7 Zusammenfassung

In diesem Kapitel wurden die grundlegenden Funktion von Smart Homes anhand der Forschungsumgebung Living Place analysiert. Anhand dieser Funktionen und der vorher genannten Grundlagen von domänenspezifischen Sprachen, wurden einige Szenarien entwickelt und abgebildet. Mit Hilfe dieser Szenarien sind die Anforderungen zur Realisierung der Vision erarbeitet worden.

Aus diesen Anforderungen entstand, in einem weiteren Schritt, ein Minimal-Entwurf der Vision, der eine Entwicklungsoberfläche für Domain Spezialisten, sowie eine Prozess-Engine für die Verarbeitung der erstellten Modelle beinhaltet. Des Weiteren ergab die Analyse weitere Anforderungen wie beispielsweise die Erweiterbarkeit. Jene soll dem Domain Spezialisten es ermöglichen die Sprache um neue Elemente zu erweitern.

In den folgenden Kapiteln werden die erarbeiteten Anforderungen aus dem Minimal-Entwurf umgesetzt. Diese werden in dem Kapitel Entwurf definiert und im Kapitel Implementierung in den Prototypen implementiert.

3 Entwurf der Domain Specific Language

3.1 Einleitung

In diesem Kapitel sollen grundlegende Entscheidungen vorgestellt werden, welche relevant für die Entwicklung dieser domänenspezifischen Sprache sind. Die Anforderungen, die bei der Entwicklung berücksichtigt werden müssen, wurden im vorangegangenen Kapitel behandelt (vgl. Abschnitt 2.6). Des Weiteren wird dem Benutzer ein Überblick über die grundlegenden Funktionen dieser domänenspezifischen Sprache gegeben.

Dabei ist die wichtigste Funktion dieser domänenspezifische Sprache, eine einfache und verständliche Entwicklung und Abbildung von Szenarien im Living Place. Um dies zu erzielen muss die domänenspezifische Sprache Erweiterbar (vgl. Abschnitt 3.6) sein und zusätzlich bei der Fehlerbehandlung (vgl. Abschnitt 4.1.4) unterstützen.

3.2 Grundlagen

Um Szenarien abbilden zu können ist die domänenspezifische Sprache als Language Workbench aufgebaut (vgl. Unterabschnitt 2.3.2), was eine leichte Lesbarkeit der Szenarien ermöglicht und außerdem die Fehlersuche erleichtert. Aus diesem Grund basiert sie zum einen auf der JavaScript Object Notation-Syntax, im folgenden JSON genannt, und zum anderen auf den Elementen von Business Process Model And Notation, im folgenden BPMN genannt. Auch wenn nur ein Bruchteil der Syntax aus der BPMN verwendet wird so wird das Hintergrundwissen über BPMN und JSON vorausgesetzt[1][2].

Des Weiteren sollen die Prozesse von einer Prozess Engine eingelesen und ausgeführt werden. Welche auch zur Kommunikation mit der Middleware dienen (vgl. Eichler, 2014, S.39-52). Zusätzlich soll die Prozess Engine bei der Fehlersuche und bei der Fehlervermeidung unterstützen.

[1] Definition zu BPMN - (OMG, 2011)
[2] Aufbau und Definition von JSON - (JSON)

3.2.1 JavaScript Object Notation

Szenarien die von einem Entwickler erstellt werden, müssen gespeichert werden können. Dabei sollte eine Sprache ausgewählt werden, die sowohl leicht zu erlernen ist, als auch das erstellte Szenario abbilden kann. Somit sollte eine Sprache ausgewählt werden, mit der der Entwickler bereits vertraut ist. Bei der Kommunikation der Agenten in der Middleware wird JSON als Nachrichtenformat eingesetzt (vgl. Eichler, 2014, S.43), da sie die Einarbeitung für den Entwickler einfacher macht, wenn hier dieselbe Sprache benutzt wird.

JSON basiert zum einen auf Name/Wert-Paaren und zum anderen auf einer geordneten Liste. Wobei die Reihenfolge in den folgenden zwei Beispielen zu vernachlässigen ist.

Um einen Prozess in JSON darzustellen werden Paare gebildet, zum einen aus Knoten, die die einzelnen Aufgaben/Tasks oder Verzweigungen/Gateways darstellen, und zum anderen aus Kanten, die die Wege zwischen den Tasks/Gateways darstellen und den Start- und End-Knoten angeben.

In Abbildung Listing 3.1 und Listing 3.2 wird die Darstellung von Name/Wert-Paaren für Knoten und Kanten dargestellt. Hierbei ist zu sehen, dass ein Knoten ("nodeDataArray") mehrere Werten hat. Einige dieser Werte sind ein Überbleibsel aus BPMN (Unterabschnitt 3.2.2) und somit nicht für diese Prozess-Engine relevant.

Knoten:

- **Key:** Ist der Schlüssel eines Knotens und dient zur eindeutigen Identifizierung eines bestimmten Knotens in einem Prozess
- **Category:** Die Knoten sind in Kategorien unterteilt, diese geben der Prozess-Engine Auskunft darüber wie mit dem Knoten umgegangen werden muss. Folgende Kategorien sind möglich:
  - **Event:** Als Events werden der Start- und End-Event angesehen, aber auch Exception-Events sind möglich. Letztere werden in der Implementierung nicht vorhanden sein.
  - **Activity:** sind die ausführenden Elemente in der Prozess-Engine. Sie können sowohl Berechnungen durchführen, als auch Nachrichten verschicken oder empfangen.

  - **Gateway:** sind Verzweigungen in der Prozess-Engine, dabei ist zwischen dem Exklusiven- und dem Parallelen-Gateway zu unterscheiden. Für einen fehlerfreien Ablauf eines Szenarios ist die richtige Verwendung von Gateways sehr wichtig.

- **Text:** Der Text ist bei Events nicht von Bedeutung, bei Activities und Gateways gibt dieser jedoch an wie mit den jeweiligen Element zu verfahren ist, bzw. welches Kommando ein Task ausführen soll.

- **EventType:** Da ein Event verschiedene Typen haben kann, wie z.B. ein Gateway, kann mit dem EventType dieser eindeutig Unterschieden werden.

- **EventDimension:** Wird nicht in der Prozess-Engine genutzt.

- **Item:** Der Item eines Knotens gibt an um welche Activity oder um welches Event es sich handelt. Im Beispiel (Listing 3.1) wird bei dem Knoten der Start-Event dargestellt.

- **Loc:** Die Loc, bzw. Location zeigt an an welcher Position der Knoten in der Language Workbench dargestellt werden soll. Dies sind die X- und Y-Koordinaten des Knotens.

Listing 3.1: Name/Wert Paare von Knoten in JSON

```
"nodeDataArray": [ {
                  "key":101,
                  "category":"event",
                  "text":"Start",
                  "eventType":1,
                  "eventDimension":1,
                  "item":"start",
                  "loc":"..."
        }, { ... }        ]
```

Kanten:

- **From:** Start-Knoten-ID.

- **To:** End-Knoten-ID.

- **FormPort:** Wird nicht in der Prozess-Engine genutzt.

- **ToPort:** Wird nicht in der Prozess-Engine genutzt.

- **Points:** Die X- und Y-Koordinaten der Kante, die Menge der Koordinaten hängt von den Anzahl der Ecken einer Kante ab.

Listing 3.2: Name/Wert Paare von Kanten in JSON

```
"linkDataArray": [
        {
                "from":101,
                "to":131,
                "fromPort":"",
                "toPort":"",
                "points":[
                        ...
                ]
        }, { ... } ]
```

3.2.2 Business Process Model And Notation

Die Entwicklung einer DSL ist sehr umfangreich und die Entwicklung eigener Symbole zur Darstellung von Objekten, ist oft sehr zeitintensiv und in vielen Fällen auch nicht Sinnvoll. Daher sollte eine Sprache verwendet werden, die viele Entwickler kennen oder schnell lernen können. Da genug Dokumentation dafür vorhanden ist, so wie z.B. bei BPMN.

Jedoch ist die Business Process Model And Notation für die Darstellung von Szenarios im Living Place sehr überladen, daher wurde eine eigene Domänenspezifische Sprache entwickelt, die sich sehr stark an die Objektdefinition von BPMN[3] orientiert.

Die Objektdefinitionen dieser DSL werden in Unterabschnitt 3.4.2 näher erläutert.

3.3 Definition der Sprache

3.3.1 Sprachstruktur

Szenarien im Living Place folgen meist dem selben Muster (vgl. Abschnitt 2.5). Um solche Szenarien in dieser DSL abbilden zu können, muss die Sprache bestimmte Lösungswege vorgeben, wodurch der Entwickler unterstützt wird. Auch bei speziellen Szenarien, die aus solchen Mustern herausfallen können, muss die DSL einige Standartelemente anbieten können

[3] Auswahl der BPMN 2.0 Poster mit BPMN-Objektdefinitionen - http://www.bpmb.de/index.php/BPMNPoster; letzter Zugriff: 31.01.2016

wie beispielsweise **Tasks**, **Gateways** oder **Start/End-Punkte**. Zusätzlich muss der Entwickler den Ablauf eines Szenarios modellieren und die möglichen Pfade angeben können.

Ein Szenario lässt sich daher in folgende Elemente unterteilen:

- **Start:** Markiert den Start eines Szenarios.
- **Ende:** Markiert das Ende eines Szenarios.
- **Timer:** Der Timer ist kein eigenständiges Objekt in dieser DSL, er wird viel mehr als Erweiterung für Objekte eingesetzt und bestimmt entweder die Startzeit oder die Dauer der Laufzeit eines Objektes mit Timer-Erweiterung.
- **Task:** Ein Task kann entweder Generisch sein, dies bedeutet aber das der Entwickler den Befehl selbst eingeben und für die Richtigkeit sorgen muss. Ein Task kann außerdem an einem, in der DSL vorhandenen, Befehl gebunden sein.
- **Ad-Hoc Sub-Prozess:** Ein Ad-Hoc Sub-Prozess kann eine n-Menge an Tasks beinhalten und diese ausführen, sobald die Bedingung zur Ausführung erfüllt wurde.
- **Gateway:** Ein Gateway ist eine Verzweigung in einem Prozess, diese kann Exklusiv aber auch Parallel sein.
- **Pfade:** Die Pfade geben den Ablauf eines Prozesses an, mit Ausnahme von Ad-Hoc Sub-Prozess, dieser kommt ohne Pfade aus. Die Pfade können bei Exklusiven Verzweigungen einen Standardweg angeben, sollte keine der angegebenen Bedingungen eintreffen.

Die Elemente lassen sich in verschiedene Gruppen unterteilen: **Activities**, **Events** und **Gateways**. Zu den **Activities** gehören die ausführenden Elemente der Sprache, **Tasks** oder **Ad-Hoc Sub-Prozess** zum Beispiel. Zu der Gruppe der **Events** die **Start/Ende** oder auch der **Timer**, letzterer wird aber nur als Start- oder Warte-Element als ein Event angezeigt, sonst nur als Bedingung sichtbar. Die **Gateways** sind die Verzweigungen in dem Szenario, dazu gehören **Exklusive Gateways** und **Parallele Gateways**.

Die Herausforderung bei der Entwicklung dieser DSL bestand darin, die größtmögliche Flexibilität bei der Erstellung von Szenarien zu ermöglichen. Und dies trotz der Vielzahl an unterschiedlichen Elementen und Abfolgen im Living Place.

3.3.2 Regeln

Jede Domänenspezifische Sprache braucht Regeln und Richtlinien zur Benutzung. Dabei ist es sehr wichtig mit diesen Regeln oder Richtlinien den Benutzer bei seiner Arbeit zu unterstützen

und nicht zu behindern. Deshalb hat auch diese Domänenspezifische Sprache einige Regeln die der Fehlervermeidung dienen.

Reihenfolge

In den meisten Fällen ist die Reihenfolge der Elemente unwichtig für die Prozess Engine. Das heißt, dass es für die Prozess Engine nicht wichtig ist, ob zuerst die Fenster oder die Vorhänge geöffnet werden. Damit soll die Domänenspezifische Sprache so flexibel wie möglich einsetzbar sein.

Die Reihenfolge wird aber zu einem wichtigen Element bei einem Einsatz von Gateways. Dieser spezielle Fall in dem folgenden Abschnitt näher erläutert.

Gateways

Gateways sind ein besonderer Fall in dieser Domänenspezifische Sprache. Sie können den Verlauf eines Szenarios steuern oder auch aufteilen. Dabei können viele Fehler auftreten oder es kann im schlimmsten Fall ein Deadlock erzeugt werden.

Um solche Fehler zu vermeiden, müssen die Gateways zum einen mit dem selben Gateway Typ abgeschlossen werden mit dem sie geöffnet wurden, und zum andren sollen folgende Regeln zusätzlich befolgt werden:

- **Exklusives Gateway:** Das Exklusive Gateway ist an eine Bedingung geknüpft und somit muss diese im Prozess Kontext vorhanden sein. Sollte das nicht der Fall sein kommt der Prozess zum Stehen. Es muss daher vor jedem Exklusiven Gateway eine Überprüfung der Bedingung stattfinden.
- **Paralleles Gateway:** Bei dem Parallelen Gateway entfällt die Überprüfung, da beide ausgehenden Pfade benutzt werden. Für die Prozess Engine ist dabei die Benennung der Gateways von Bedeutung. Daher ist folgende Konvention zu beachten: das anfangende Gateway hat die Bezeichnung "p1:start", das beendende Gateway die Bezeichnung "p1:end". Dabei ist die Zahl die ID des Gateways, sollten daher mehrere Verwendet werden, so wird die Zahl inkrementiert.

  Zusätzlich gilt des Weiteren folgendes:

  - **Richtig:** -> p1:start ... -> p2:start ... -> p2:end -> ... -> p1:end ->

- **Falsch:** -> p1:start ... -> p2:start ... -> p1:end -> ... -> p2:end ->

Pfade

Wie im Abschnitt Reihenfolge erwähnt, ist die Reihenfolge in den meisten Fällen irrelevant, daher gibt es auch kaum Einschränkungen wie die Pfade zwischen den einzelnen Knoten gesetzt werden. Wichtig ist nur, wie viele Pfade an einen Knoten gebunden werden.

Jeder Knoten hat eine bestimmte Anzahl an Eingängen oder Ausgängen, diese sieht wie folgt aus:

- **Start:** *Eingänge:* 0 / *Ausgänge:* 1
- **End:** *Eingänge:* 1 / *Ausgänge:* 0
- **Gateways:** *Eingänge:* 1 / *Ausgänge:* 2 oder *Eingänge:* 2 / *Ausgänge:* 1
- **Task:** *Eingänge:* 1 / *Ausgänge:* 1 / innerhalb eines Sub-Processes werden keine Pfade benötigt.
- **Ad-Hoc Sub-Process:** *Eingänge:* 1 / *Ausgänge:* 1 / innerhalb des Sub-Processes werden keine Pfade benötigt.
- **Timer:** je nach Einsatzgebiet dieser Entität können die Ein-/Ausgänge variieren, es können aber nicht mehr als 1 Eingang und 1 Ausgang sein.

3.4 Sprachelemente

3.4.1 Einleitung

In den folgenden Abschnitten werden die einzelnen Sprachelemente und deren Erscheinungsbild in der Entwicklungsumgebung näher erläutert. Zusätzlich wird die JSON-Syntax der einzelnen Elemente definiert und dargestellt. Sämtliche gezeigten Grafiken sind aus der erarbeiteten Entwicklungsumgebung entnommen.

3.4.2 Objektdefinition

Start-Event

Das Start-Event markiert den Start eines Szenarios. Die Darstellung in der Entwicklungsoberfläche sieht wie in Abbildung 3.1 aus. Dieses Element hat keinen Eingangspfad und genau einen Ausgangspfad. Da es sich hier um einen einfachen Start handelt, enthält dieses Element keine zusätzlichen Parameter.

Abbildung 3.1: Darstellung eines Start-Events

End-Event

Abbildung 3.2 zeigt ein End-Event, dieses hat, im Gegensatz zum Start-Event, genau einen Eingangpfad und keinen Ausgangspfad. Er markiert das erfolgreiche Ende eines Szenarios. Sollte die Prozess Engine an dem Punkt ankommen, so geht diese davon aus das keine Fehler im Prozess aufgetreten sind oder diese Erfolgreich bearbeitet wurden.

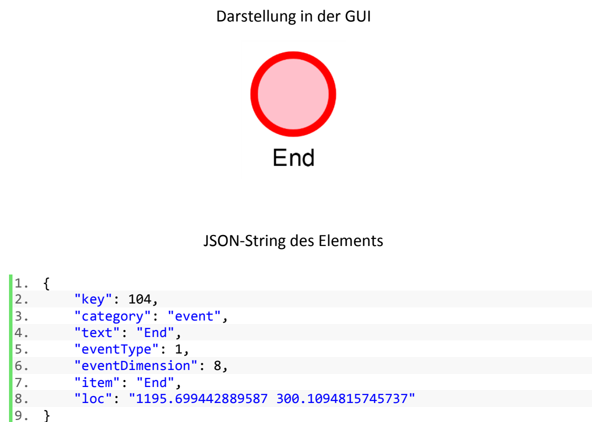

Abbildung 3.2: Darstellung eines End-Events

Timer-Event

Der Timer-Event kann sowohl ein Start-Event, als auch ein Warte-Event sein (vgl. Abbildung 3.3). Dieser soll zusätzlich eine Uhrzeit übergeben bekommen, die angibt wann die Prozess Engine an dieser Stelle weitermachen darf. Die Übergabe von Werten wird mittels Properties ermöglicht.

Abbildung 3.3: Darstellung eines Timer-Events

Task-Activity

Abbildung 3.4 zeigt einen generischer Task, dieser sendet den String-Value aus dem "item" an den TaskInterpreter, sichtbar in dem JSON-Ausdruck. Ein Task kann zur Ausführung oder Berechnung von Aufgaben verwendet werden. Dies ist bei einigen Prozessen Notwendig um zu bestimmen was als nächstes gemacht werden soll.

Zusätzlich kann ein Task mehrere Properties Übergeben bekommen, diese können zum einen als Bedingung zum Ausführen des Tasks eingesetzt werden und zum anderen als zusätzliche Parameter.

Des Weiteren wird auch der generische Tasks implementiert, bei diesem Task-Typ gibt der Entwickler den Befehl ein, der an das Living Place, bzw. an die Middleware versendet werden soll.

Abbildung 3.4: Darstellung eines Tasks

Ad-Hoc Sub-Prozess Activity

Die Ad-Hoc Sub-Prozess Activity ist ein Container, der eine Ansammlung von Tasks beinhaltet, siehe Abbildung 3.5. Dabei wird an den Container ein Set an Properties übergeben, diese geben an wann und welche Tasks gestartet werden sollen und zu welchem Zeitpunkt die Ausführung des Containers beendet werden kann.

Abbildung 3.5: Darstellung eines Ad-Hoc Sub-Prozess

Exklusives Gateway

In Abbildung 3.6 wird ein Beispiel eines Exklusiven Gateways gezeigt. Jenes soll es dem Entwickler ermöglichen seine Szenarien flexibel zu gestalten oder auf bestimmte Ereignisse oder Berechnungen zu reagieren. Ein mögliches Ereignis ist, wie in Unterabschnitt 2.5.1, die Reaktion auf das Wetter. So wird nur gelüftet wenn der Wind nicht zu stark ist.

Hierbei ist darauf zu achten, dass jedes Gateway auch mit dem selben Gateway Typ abgeschlossen werden muss (vgl. Abschnitt 4.1.2).

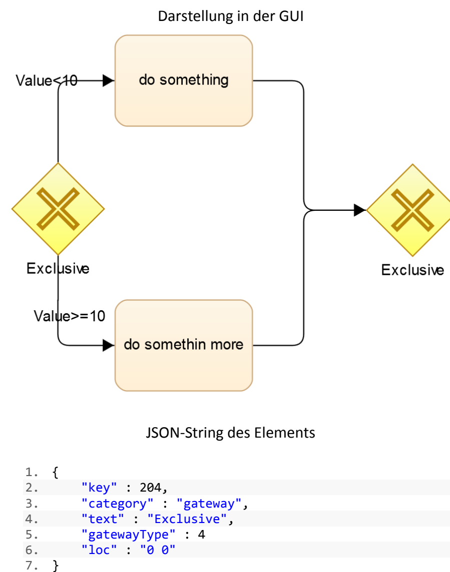

Abbildung 3.6: Darstellung eines Exklusiven Gateways

Paralleles Gateway

Das Parallele Gateway führt alle weiterführenden Pfade aus. Ein Beispiel dieses Gateways wird in Abbildung 3.7 gezeigt. Hierbei sollte der Entwickler darauf achten das die parallelen Zweige nicht auf die selbe Ressource zugreifen. Dies kann zu einem Deadlock in einem Prozess führen und diesen an der Ausführung seiner Aufgabe hindern. Auch bei diesem Gateway ist es wichtig, dass es mit dem selben Gateway Typ abgeschlossen wird (vgl. Abschnitt 4.1.2) und auch die selbe ID wie dem Start Gateway zugewiesen wird. In Abbildung 3.7 ist ein Beispiel wie die ID und Benennung von Parallelen Gateways durchzuführen ist.

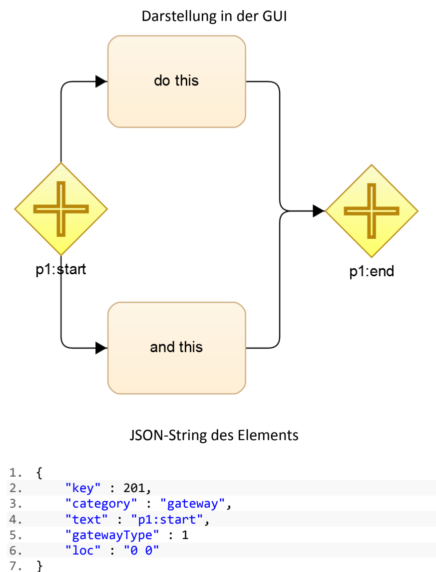

Abbildung 3.7: Darstellung eines Parallelen Gateways

Prozess Variablen

Die Prozess Variablen werden nicht wie die vorher angegebenen Elemente benannt oder implementiert, da sie schon in der Prozess Engine vorhanden sind. Sie sollen Berechnungen oder die Ergebnisse von Abfragen speichern und für andere Elemente der Sprache zur Verführung stellen.

Das setzten dieser Variablen übernehmen die Tasks automatisch, indem der Task seine Berechnung direkt in die Prozess Variablen ablegt.

3.5 Architektur-Übersicht

In diesem Abschnitt wird die grundlegende Architektur der Grundlegenden Komponenten und deren Verknüpfung untereinander vorgestellt. Das in Abbildung 3.8 dargestellte Komponenten-Diagramm zeigt, dass das System modular aufgebaut ist und somit einige Komponenten möglichst einfach ausgetauscht werden können.

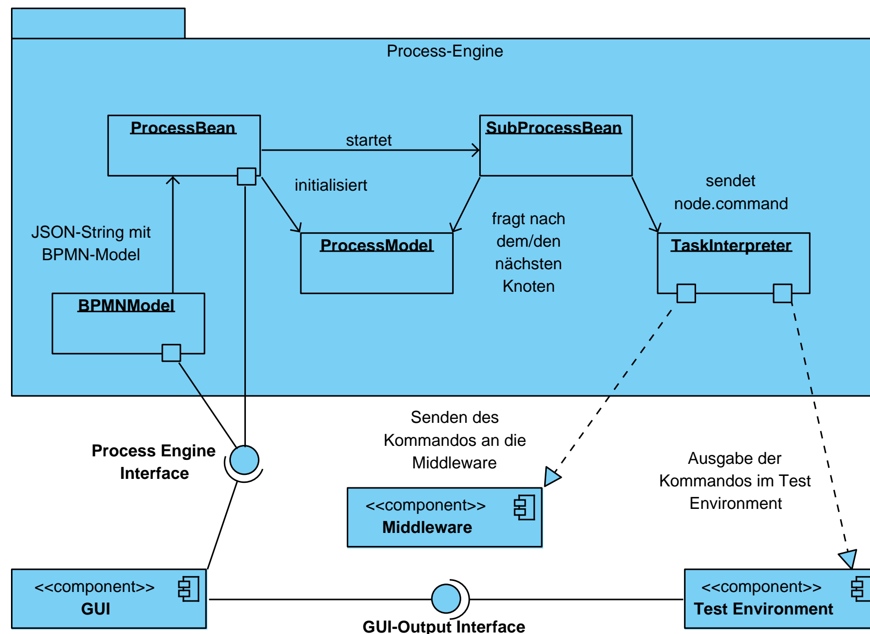

Abbildung 3.8: Komponenten Diagramm

3.5.1 Entwicklungsoberfläche

Bei der Entwicklung der Entwicklungsoberfläche wurde auf eine Erweiterung des go.js[4] Framework zurückgegriffen. Dieses wurde nach den Anforderungen aus Abschnitt 2.6.3 angepasst.

In Abbildung 3.9 dargestellte Entwicklungsoberfläche ist wie folgt strukturiert:

- **File:** Hier findet der Benutzer die Standardfunktionen, **New**, **Open** und **Save**.
- **Edit:** Diese Funktionen sollen den Benutzer bei der Arbeit unterstützen. Es handelt sich hierbei um folgende Funktionen: **Undo**, **Redo**, **Cut**, **Copy**, **Paste**, **Delete** und **Select All**.
- **Align & Space:** Auch hierbei handelt es sich um unterstützende Funktionen, die den Benutzer bei der Positionierung und Ausrichtung der Elemente in der Entwicklungsoberfläche unterstützen sollen.
- **Test:** Dieser Menüpunkt startet das Test Environment, dabei werden die Ausgaben vom TaskInterpreter in eine Datei oder direkt in der Entwicklungsoberfläche ausgegeben.

[4]BPMN Extension von go.js - http://www.gojs.net/latest/extensions/BPMN.html; letzter Zugriff: 31.01.2016

Die in der Entwicklungsumgebung dargestellten Tasks, Events und Gateways (vgl. Abbildung 3.9) sind nur ein Bruchteil der Elemente aus dem Living Place. Das benötigte wissen für die Implementierung weiterer Elemente wird in dem Abschnitt 3.6 vermittelt.

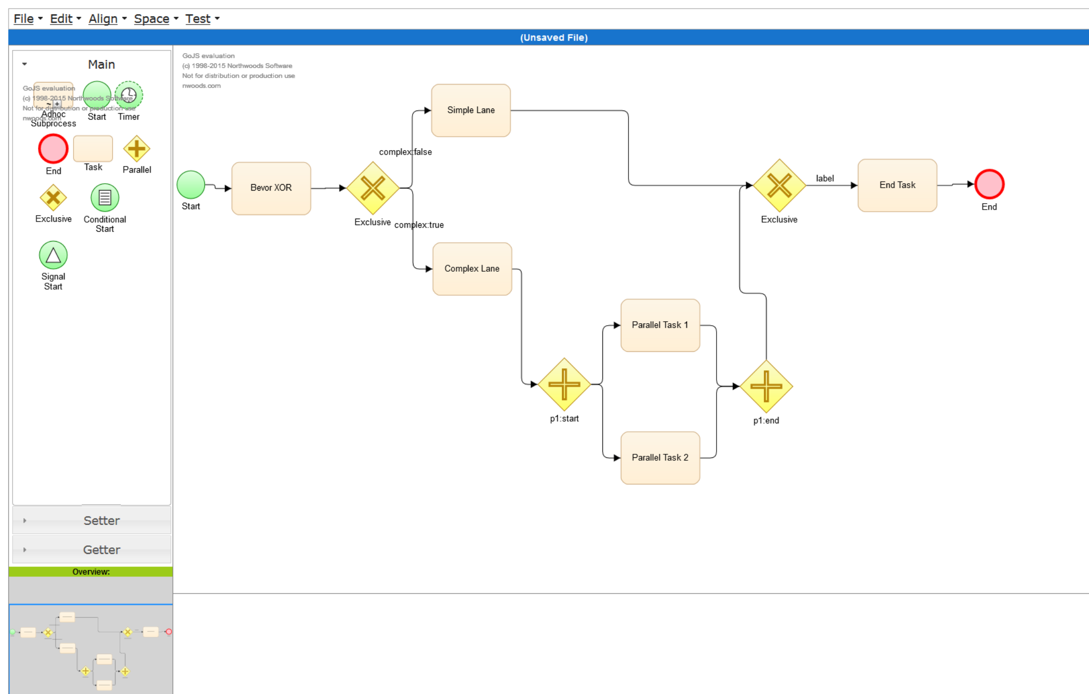

Abbildung 3.9: Darstellung der Entwicklungsumgebung

3.5.2 Prozess-Engine

Das Kernsystem der Prozess Engine besteht aus folgenden Komponenten (vgl. Abbildung 3.10):

- **ProcessBean:** Die ProcessBean ist das zentrale Element der Prozess Engine, diese startet den Prozess, übersetzt das BPMNModel in das ProcessModel und erstellt eine Liste mit unter Prozessen die von der SubProcessBean verwaltet werden.

- **ProcessModel:** Das ProcessModel ist die interne Darstellung des Prozesses, diese besteht aus Knoten und Kanten. Jeder Knoten ist ein Element in dem BPMN-Modell, z.B. ein Task oder Gateway. Zusätzlich enthält der Knoten eine Liste mit Kanten, die von diesem Knoten weggehen. Hat der Knoten keine Kanten, so ist dieser dann der End-Knoten.

- **SubProcessBean:** Das verarbeiten der Knoten übernimmt die SubProcessBean. Nach dem erfolgreichen bearbeiten eines Knoten, ruft dieser sich rekursiv wieder auf mit dem nächsten Knoten. Dazu wird jedes mal ein neuer Thread erstellt und nach dem beenden wieder beendet. Jeder Knoten, bis auf den Start-Knoten, wartet auf das beenden seines Vorgängers. Bei einem Parallelen Gateway werden die jeweiligen Stränge als Unterprozesse angesehen, bei denen der Gateway Start und das Ende die Prozess Start- und End-Punkte sind.

- **BPMNModel:** Das auf der Business Process Model And Notation basierende BPMNModel dient als Schnittstelle für die GUI, die mit diesem Standard die Prozesse erstellt. Diese könnte in zukünftigen Arbeiten durch andere Modellierung-Sprachen ersetzt werden, wie z.B. UML, die dann in das ProcessModel übersetzt werden können.

- **TaskInterpreter:** Das versenden der Kommandos übernimmt der TaskInterpreter, dieser wird in Unterabschnitt 3.5.3 näher erläutert.

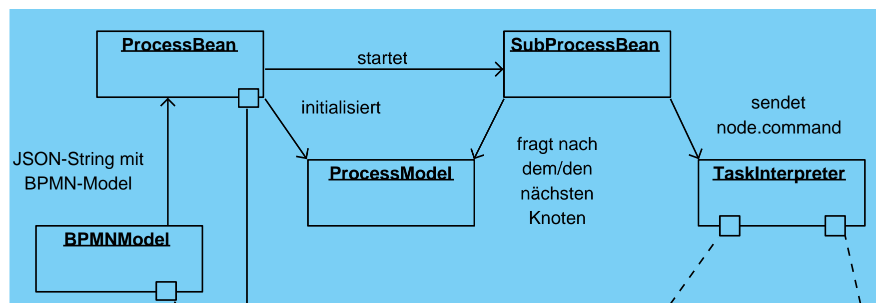

Abbildung 3.10: Detailansicht der Prozess Engine

Prozessablauf

Das in Abbildung 3.11 dargestellte Sequenz-Diagramm zeigt einen möglichen Prozessablauf. Dieser beginnt mit der Erstellung des Szenarios in der Entwicklungsumgebung durch den Domain-Spezialisten. Nach dem Abspeichern kann der Domain-Spezialist das Szenario in der Test-Umgebung starten. Durch das Speichern des Szenarios wird der JSON-String, der das Szenario repräsentiert, an die ProzessBean übergeben. Das Starten des Szenarios erzeugt einen Thread mit der SubProzessBean, diese beinhaltet den Start-Knoten aus dem ProcessModel. Die SubProcessBean liest den Typ des Knotens aus, sollte dieser ein Task sein, so sendet diese das Kommando an den TaskInterpreter. Nach dem versenden des Kommandos fragt die SubProcessBean das ProcessModel nach seinem nächsten Knoten und ruft sich selber mit

diesem neuen Knoten auf. Sollte kein weiterer Knoten vorhanden sein wird der Thread mit dieser SubProcessBean beendet.

Eine Besonderheit ist das Parallele Gateway, in diesem Fall erstellt die SubProcessBean zwei neue Threads mit, bei denen die beiden Knoten nach dem Gateway die neuen Start-Knoten sind.

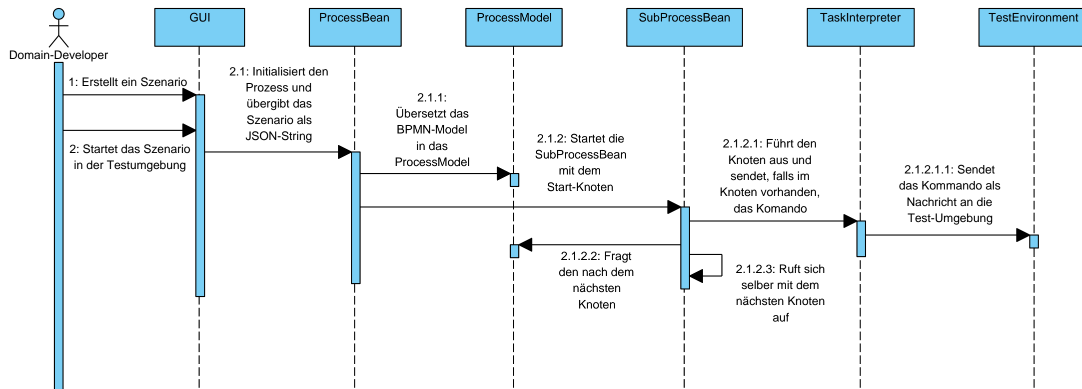

Abbildung 3.11: Prozessablauf dargestellt in einem Sequenz-Diagramm

3.5.3 Task Interpreter

Der Task Interpreter dient als Übersetzter und Übermittler der Nachrichten von der Prozess Engine zur Middleware oder der Test Umgebung. Übersetzt werden die Kommandos die sich in den Tasks befinden. So ein Task mit Kommando kann wie in Listing 3.3 aussehen. Dabei ist der "item" der Key-Value der das Kommando "SET:WINDOW:2:20:FAST" beinhaltet. Das Kommando bedeutet, dass das **Fenster** mit der ID **2** auf den Wert **20** mit dem zusätzlichen Ausdruck **Fast** gesetzt werden soll.

Listing 3.3: Task-Knoten mit Kommando

```
{
        "key":-8,
        "category":"activity",
        "text":"Oeffne Fenster mit ID 2 auf den Wert 20, Schnell",
        "item":"SET:WINDOW:2:20:FAST",
        "taskType":0,
        "property_1": "",
        "property_2": "",
        "property_3": ""
```

}

Die folgende Abbildung 3.12 zeigt das Klassendiagramm des Task Interpreters. Die Methode "parseCommand(...)" wird von der Prozess Engine, mit dem im Knoten enthaltenen Kommando und den Prozess Variablen aus dem Prozesskontext (vgl. Abschnitt 3.4.2), ausgeführt. Diese Methode entscheidet mittels Switch-Case, welche "Commando"-Methode gestartet werden soll. Die Methoden die in der Middleware nur einen Befehl versenden, brauchen nur den Kommando-String als Übergabeparameter. Die Abfragenden-Methoden bekommen zusätzlich zu dem Kommando-String auch die Prozess Variablen übergeben, in denen diese die Antwort auf die Anfrage ablegen. Dies bedeutet ferner, dass die Prozess Engine solange wartet bis dieser Task abgeschlossen und die Antwort in den Prozess Variablen eingetragen worden ist. Sollte ein paralleler Strang vorhanden sein, so läuft dieser wie gewohnt weiter und wird nicht vom warten beeinflusst.

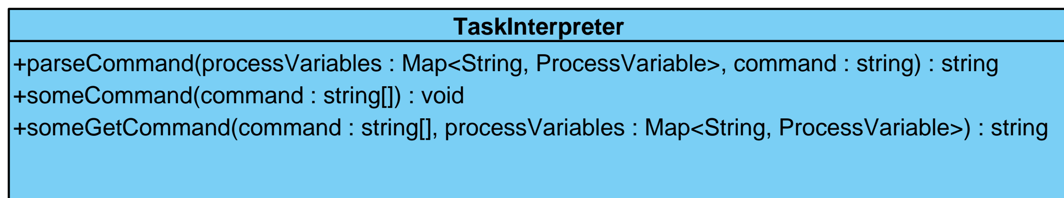

Abbildung 3.12: Klassendiagramm: TaskInterpreter

3.5.4 Test Environment

Um die Process Engine und deren Befehle unter möglichst besten Bedingungen zu testen ist eine Umgebung notwendig die der Ziel-Domäne stark ähnelt. In dieser Arbeit ist die Ziel-Domäne das Living Place Hamburg, welches auf einem Multiagentensystem basiert, dass das Akka-Framework nutzt (Eichler, 2014, S.52-60).

Daher nutzt die Test-Umgebung das selbe Framework. Dabei werden die übersetzten Nachrichten von einem "Publisher-Agent" auf den selben Kanal veröffentlicht, auf dem sie auch in der Middleware veröffentlicht werden würden. Diese Nachrichten können dann mit der erwarteten Nachricht verglichen werden und somit auch geprüft werden. Des Weiteren ist auch eine Ausgabe in der Konsole ist denkbar.

3.6 Erweiterbarkeit

Die Erweiterbarkeit ist ein sehr wichtiger Bestandteil dieser domänenspezifischen Sprache. Hierbei handelt es sich um eine Erweiterung um neue Komponente, die zusätzlich in die domänenspezifische Sprache eingebunden werden sollen. Und nicht um die Erweiterung der domänenspezifischen Sprache an sich.

Dies soll durch die Auslagerung des Task-Interpreters aus dem eigentlichen Prozesskern ermöglicht werden. Dabei behält die Prozess Engine ein zentrales Interface, das die Kommandos aus den Knoten übersetzt. Und der Softwarearchitekt hat die Möglichkeit dieses Interface mit seiner Erweiterung zu erweitern.

Um den Task Interpreter zu erweitern, muss der Softwarearchitekt die zusätzlichen Kommandos in den Task Interpreter einpflegen und die Switch-Case-Anweisung in der Methode "parseCommand" durch die Kommandos erweitern.

Damit diese auch in der Entwicklungsoberfläche sichtbar werden, muss auch diese erweitert werden. Da es sich hierbei um ein externes Framework von "go.js" handelt, besteht die Möglichkeit zur Erweiterung in einem JavaScript. Dabei wird die JavaScript Datei "BPMNLivingplace.js" ab der Zeile 1786 mit dem gewünschten Kommando erweitert. Die könnte wie folgt aussehen:

Listing 3.4: Task-Knoten mit Kommando

```
{
        "key":-8,
        "category":"activity",
        "text":"Setze Licht mit ID 2 auf den Wert 100",
        "item":"SET:LIGHT:2:100",
        "taskType":0,
        "property_1": "",
        "property_2": "",
        "property_3": ""
}
```

4 Evaluation

4.1 Implementierung

4.1.1 Einleitung

Die Implementierung der domänenspezifischen Sprache erfolgte nach den Vorgaben aus Kapitel 3. Es wurden die Frameworks benutzt die sowohl im Analyse-Teil als auch im Entwurf genannt worden sind. Somit wurde die Entwicklungsoberfläche mit dem **go.js** Framework, als Web-Service eingebunden. Die Kommunikation zwischen dem Web-Service und den Server übernimmt Java Server Faces, im folgenden JSF[1] genannt. Es handelt sich hierbei um eine Entwicklung von Oracle, die die Web-Basierte Anwendungsentwicklung erleichtert. Zusätzlich werden standardisierte Interface-Komponenten bereitgestellt, die wiederum auf Server-Komponente zugreifen können.

Abschließend wurde ein Prototyp einer Prozess-Engine implementiert, der die Machbarkeit der Idee demonstriert. Bei der Implementierung bestand die Herausforderung darin, die Tasks, aber auch die parallelen Stränge eines Prozesses in separaten Threads koordiniert ablaufen zu lassen.

4.1.2 Entwicklungsoberfläche

Wie in der Einleitung beschrieben, wurde das **go.js** Framework zur Erstellung und Darstellung von Szenarien benutzt. Das Framework bildet in diesem Zusammenhang den obersten Layer, beziehungsweise den Präsentation-Layer, in diesem Modell. Die Kommunikation mit den vom Server angebotenen Services werden mit JSF realisiert, welches zum UI-Layer gehört. Die Prozess-Engine, welche nur prototypisch implementiert wurde, befindet sich auf dem Service-Layer in diesem Model. Hier wird mittels Controller auf die Schnittstellen der Prozess-Engine zugegriffen (vgl. Abbildung 4.1). Es entfällt folglich der Backend-Layer, da in den Prototypen keine Speicherung der erstellten Modelle in der Datenbank vorgesehen ist.

[1] Das grundlegende Verständnis von JSF wird hier vorausgesetzt, da die Darstellung der GUI mittels dieser Spezifikation realisiert wurde. Einen guten Einstieg in dieses Thema bietet die Dokumentation von Oracle (vgl. (Oracle)).

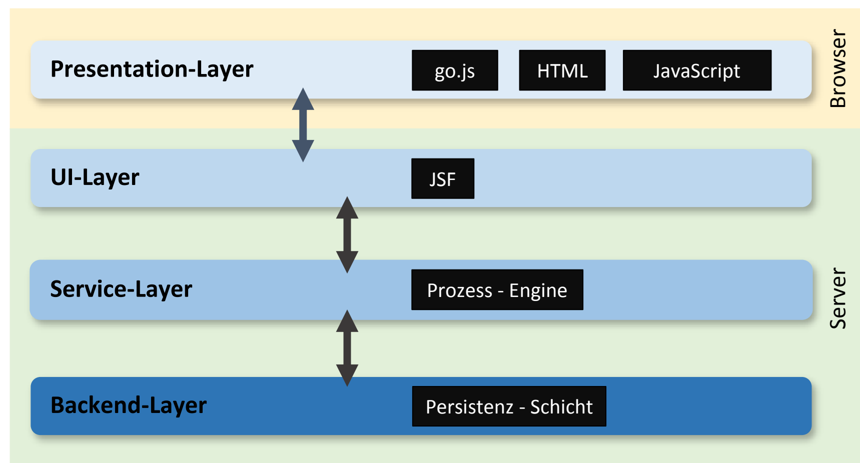

Abbildung 4.1: Layer-Model zur Entwicklungsoberfläche

Funktionen

Folgende Funktionen sind in den Prototypen implementiert worden:

- **New:** Diese Funktion stellt ein leeres Workspace zur Verfügung, dabei wird der bisherige Fortschritt gelöscht.
- **Save:** Das aktuelle Modell im Workspace wird in das BPMN-Modell aus der BPMN-Spezifikation übersetzt.
- **Edit/Align & Space:** Diese Funktionen sind im **go.js** Framework schon enthalten und funktionieren wie in Unterabschnitt 3.5.1 beschrieben.
- **Test:** Startet das gespeicherte Modell mit den zusätzlich eingetragenen Prozess-Variablen.

Regeln

Es handelt sich zu diesem Zeitpunkt um eine sehr frühe Implementierungsphase, bei der der Fokus auf der Funktion und nicht auf der Benutzerfreundlichkeit liegt. Somit muss der Benutzter zunächst das Modell speichern, damit dieses übersetzt werden kann, bevor er es ausführt.

Des Weiteren muss der Benutzer die Prozess-Variablen, die zur Berechnung oder für Entscheidungen bei Gateways benötigt werden, vor dem Prozess Start eingeben, da die Schnittstelle zur Middleware nicht implementiert wurde. An diesem Punkt ansetzende Arbeiten können die

Benutzerfreundlichkeit der Entwicklungsoberfläche, aber auch mit der dem implementieren der Middleware-Schnittstelle näher beleuchten.

4.1.3 Prozess-Engine Prototyp

Um die Grundfunktionen für die Prozess-Engine bereit zu stellen, wurde das **ProcessModel**, nach Unterabschnitt 3.5.2, implementiert. Anhand von diesem Modell wurde die **ProcessBean** und die **SubProcessBean** implementiert, da diese auf den Elementen und Methoden des **ProcessModels** aufbauen.

Zur Simulation der Kommunikation mit der Middleware, wurde das Akka-Framework implementiert. Die Übersetzung der Kommandos, die mittels Publish-Subcribe Schema versendet werden, übernimmt der **TaskInterpreter**, der von der **SubProcessBean** aufgerufen wird.

In den folgenden Abschnitten werden die Elemente genannt die bereits in den Prototypen implementiert sind. Zusätzlich dazu wird deren Umfang erläutert.

ProcessModel

Das **ProcessModel** beinhaltet zum einen den Start- und den End-Knoten und zum anderen zwei HashMaps. Bei der HashMap für Knoten wird die ID des Knotens als Key-Value der Map verwendet. Im Gegensatz dazu beinhaltet die HashMap der Kanten jeweils eine Liste mit n-Elementen pro Key-Value. Dies ist Notwendig, wenn ein Knoten mehr als einen ausgehenden Pfad besitzt, was bei Gateways der Fall ist.

Die in der Einleitung genannten Methoden des **ProzessModels** sind der *Constructor* und die Methode *getNextNodes*. Der *Constructor* hat die Aufgabe das **ProzessModel** zu initialisieren und das übergebene **BPMNModel** in das **ProzessModel** zu überführen. Nach der erfolgreichen Initialisierung des Models kann die Prozess-Engine mit der Methode *getNextNodes* die nächsten Knoten abfragen. Hierbei wird zum einen der momentane Knoten übergeben und zum anderen die Prozess Variablen. Sollte es sich um den Start-Knoten handelt, so kann dieser mit der Methode *getStart* abgefragt werden.

Node/Link

Die Klassen **Node** und **Link** beinhalten bis auf die Getter- und Setter-Methoden keine weiteren Methoden. Jene Klassen haben die Aufgabe die Informationen aus dem Modell für die Prozess-Engine bereit zu halten (vgl. Abbildung 4.2).

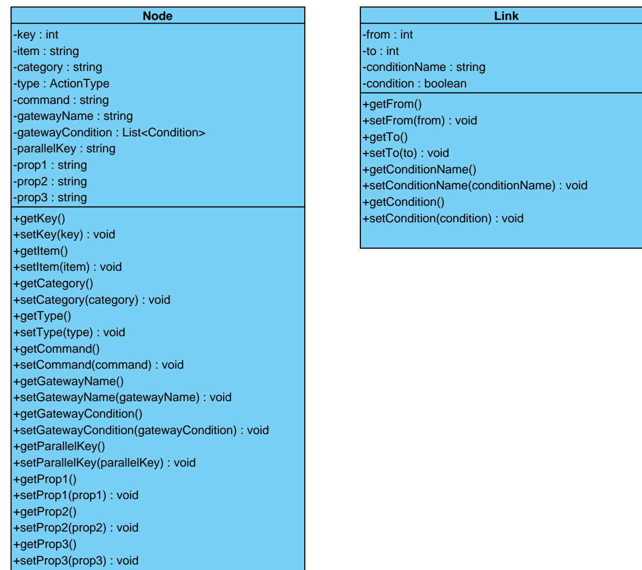

Abbildung 4.2: Klassendiagramm zu Node und Link

ProcessBean

Diese Klasse spielt eine zentrale Rolle in der Prozess-Engine, denn sie beinhaltet die Methode, die die Prozess-Engine startet. Des Weiteren beinhaltet diese die Informationen über die Prozess Variablen und auch über die Verbindung zum Aktor System des Akka-Framework. Die Initialisierung erfolgt dabei über den Konstruktor, dieser hat die Parameter bpmnModel und die HashMap mit den processVariblen. Nachdem dies erfolgt ist wird der Prozess durch die Methode *start* gestartet. Beim Ausführen jener Methode wird die Klasse **SubProcessBean** mit den benötigten Parametern erzeugt und gestartet, da diese in das Interface Thread implementiert und die *run*-Methode überschrieben wurden.

SubProcessBean

Die **SubProcessBean** hat zum einen die Aufgabe die Knoten auszuführen und zum anderen den nächsten Knoten aufzurufen. Das Ausführen der Knoten erfolgt durch die Unterklasse **NodeRunner**, diese ruft die Methode *tryTaskCommand*. Da es sich hierbei um einen Prototypen handelt, werden nur Tasks ausgeführt und deren Kommando vom **TaskInterpreter** übersetzt und an den Actor übergeben, der diese Nachricht auf den richtigen Kannal veröffentlicht. Sollte die Bearbeitung eines Knotens beendet sein, wird der nächste Knoten aus dem **ProcessModel** aufgerufen.

So wie der Name dieser Klasse schon sagt, **SubProcessBean**, kann ein Prozess in mehrere Unterprozesse zerlegt werden. Dies ist der Fall wenn ein Szenario ein paralleles Gateway hat. Hierbei werden die beiden parallelen Strenge in Unterprozesse zerlegt und aufgerufen, wenn der nächste Knoten der Startpunkt eines parallelen Gateways ist.

Außerdem enthält die **SubProcessBean** die Informationen über die Prozess Variablen und auch über das **ProcessModel**.

ProcessVariable

Die Klasse **ProcessVariable** enthält zwei Elemente. Das erste ist das Element value vom Typ Object und das zweite ist die classOfValue vom Typ Class<?>. Ersteres ermöglicht dem Benutzer jedes beliebige Objekt als Wert in die Prozess Variablen abzulegen. Wird der Wert benötigt, so kann dieser mit der Getter-Methode ausgelesen werden und mit Hilfe des zweiten Elements in die vorherige Klasse überführt werden.

Der Benutzer muss darauf achten, dass er an dieser Stelle die richtige Klasse als classOfValue angibt, da es andernfalls zu Exceptions führen kann.

TaskInterpreter

Der **TaskInterpreter** hat die Aufgabe die Kommandos aus den Knoten für die Middleware zu übersetzten, weshalb die Methode *parseCommand* implementiert wurde. Hierbei wird das Kommando aus dem Knoten an die Methode übergeben und in den String-Ausdruck übersetzt, der in der Middleware bei der Kommunikation verwendet wird. Ferner müssen die Prozess Variablen übergeben werden. Dieser Schritt ist Notwendig, um die Antwort von einem Agenten zu speichern und diese im späteren Verlauf des Prozesses verwenden zu können.

4.1.4 Fazit

Mit dieser prototypischen Implementierung erlaubt die Prozess-Engine dem Benutzer einfache Szenarien abzubilden. Solche Szenarien können Verzweigungen oder senden von Kommandos beinhalten. Die gesendeten Kommandos werden von dem **TaskInterpreter** übersetzt und verarbeitet. Das dabei implementierte Multi-Agenten-System, welches mit dem Akka-Framework realisiert wurde, soll die Machbarkeit der Verknüpfung zwischen Middleware und dieser Prozess-Engine demonstrieren.

Jedoch fehlen noch einige, wichtige Bestandteile der Prozess-Engine um diese in die Middle-

ware zu implementieren. In diesem Abschnitt werden einige dieser Bestandteile benannt und aufbereitet.

Entwicklungsoberfläche

Das **go.js**-Frameworks bietet viele Möglichkeiten, ist aber an Lizensen gebunden. Deshalb ist die Entwicklung einer eigenen Entwicklungsoberfläche für den Forschungseinsatz von Nöten. Bei der Entwicklung steht die Erweiterbarkeit im Vordergrund, dennoch sollte die Bedienungsfreundlichkeit nicht außer Acht gelassen werden. Ein möglicher Ansatz sind Frameworks wie beispielsweise AngularJS.

Prozess-Engine

Die im Prototyp implementierten Funktionen sind nur ein Bruchteil derer, die in der Forschungsumgebung des Living Place zur Verfügung stehen.

Zudem sind einige Verbesserungen an der Prozess-Engine notwendig bevor diese in die Middleware implementiert werden kann. Es muss an vielen Stellen eine Fehlerbehandlung stattfinden, da diese Fehler zum Deadlock oder zum Absturz des Systems führen könnten. Des Weiteren müssen Funktionen wie der "Ad-Hoc Sub-Prozess" oder der "Timer-Event" implementiert werden, um eine Vielzahl an Szenarien abdecken zu können.

Auch die Leistungsfähigkeit sollte vor der Implementierung geprüft und möglicherweise überdacht werden. Da diese Prozess-Engine einen rekursiven Aufruf enthält, könnte dieser sich auf die Leistungsfähigkeit bei großen Szenarien auswirken und diese beim Ablauf bremsen. Dies könnte die Engine für einige Szenarien, bei den kleine Latenzen wichtig sind, unbrauchbar machen.

Fehlersuche, -vermeidung und -behandlung

Die Behandlung von Fehlern im Prototypen beschränkt sich auf die Regeln bei dem Einsatz dieser Prozess-Engine (vgl. Abschnitt 4.1.2). Dies muss bei zukünftigen Arbeiten nachgeholt werden.

Dabei sind Logging-Verfahren an kritischen Stellen der Engine nicht ausreichend. Vielmehr sollten diese durch Fehlerbehandlungen ersetzt werden. Auch Deadlocks können Auftreten, da jene von der Middleware nicht ausgeschlossen werden können. Um auftretende Deadlocks in der Middleware abzufangen, müssen an Stellen wie dem **TaskInterpreter** oder der **SubProzessBean** Mechanismen implementiert werden, die diese Erkennen und den Prozess beenden.

Eine Möglichkeit dafür sind Timer-Mechanismen. Sie könnten nach einiger Zeit, in der keine Antwort von der Middleware kommt, dem Prozess beenden oder neu starten.

4.2 Domänenspezifische Sprache

In diesem Abschnitt wird die Entworfene DSL in Verbindung mit der gezeigt Entwicklungsumgebung evaluiert und an einigen Beispielen demonstriert. Des Weiteren werden die Implementierten Elemente der Entworfenen domänenspezifischen Sprache genannt und deren Umfang erläutert. Im Anschluss daran wird ein Ausblick über notwendige oder mögliche Erweiterungen der Sprache gegeben.

4.2.1 Evaluation der DSL

Die Bewertung der implementierte domänenspezifische Sprache erfolgt nach den festgelegten Kriterien aus dem Analyse-Teil (vgl. Unterabschnitt 2.6.2). Dabei wird auf die genannten Punkte einzeln eingegangen. Des Weiteren wird hier die Abdeckung der implementierten domänenspezifischen Sprache im Verhältnis zu der Domäne Living Place erläutert.

Konformität

Die Elemente der domänenspezifischen Sprache müssen die Kernkonzepte der Domäne abbilden können. In diesem frühen Stadium der Implementierung ist dies nicht vollständig gegeben. Es werden zwar grundlegende Elemente wie **Start** oder **Stop** durch die domänenspezifische Sprache abgebildet, jedoch fehlen Elemente die ein zeitverzögerten Start ermöglichen.

Trotz fehlender Elemente können einfache Szenarien abgebildet und verarbeitet werden. Dies beweist, dass die domänenspezifischen Sprache alle Kernkonzepte der Domäne abbilden kann.

Orthogonalität

Die Orthogonalität ist in einer domänenspezifischen Sprache ein sehr wichtiges Element, welches auch in dieser domänenspezifischen Sprache umgesetzt werden sollte. Die Umsetzung dieser Anforderung wurde erreicht, indem den Elementen der Sprache jeweils eine Funktion zugewiesen wurde.

Mit Hilfe des Start-Elements kann das Szenario gestartet oder mit dem Task-Element die Kommunikation mit den Elementen der Middleware ermöglicht werden. Weitere Elemente, die

in zukünftigen Arbeiten hinzugefügt werden könnten, sollen sich an dieser Vorgabe orientieren und umsetzten.

Erweiterbarkeit

Hierbei bezieht sich die Erweiterbarkeit auf die Erweiterbarkeit der Sprach um neue Komponente, diese sollen die Abdeckung der Domäne erhöhen. Es wurde darauf geachtet, dass diese möglichst leicht durchgeführt werden kann. Die Erweiterbarkeit wurde dadurch belegt, dass Komponente, die mit der Middleware kommunizieren können vorhanden sind. Eine Erweiterung um zusätzliche Komponente wird im folgenden Abschnitt 3.6 beschrieben.

Unterstützung

Die Unterstützung des Anwenders soll hierbei zum einen die Entwicklungsoberfläche und zum anderen das Test-Framework sein. Die Entwicklungsoberfläche ist im Prototypen implementiert und soll den Anwender dabei unterstützen die gewünschten Szenarien zu planen und zu entwerfen. Das Test-Framework soll die entworfenen Szenarien überprüfen und den Anwender auf Fehler hinweisen.

Dabei unterstützt die Entwicklungsoberfläche alle Elemente die in dieser domänenspezifischen Sprache vorkommen. Das Ziel zukünftiger Arbeiten sollte sich auf das Debugging in einem Test-Framework konzentrieren.

Abdeckung der DSL

Eine komplette Abdeckung der Domäne Living Place ist nicht möglich, da es sich um eine Forschungsumgebung handelt, die mit jedem durchgeführten Projekt wächst. Dennoch können kritische Komponenten, die sehr häufig zum Einsatz kommen, leicht und schnell implementiert werden. Somit beschränken sich die implementierten Komponente auf ein sehr kleinen Teil der Domäne und sollen nur die Machbarkeit demonstrieren.

4.2.2 Implementierte Elemente der Sprache

In diesem Abschnitt werden die implementierten Elemente der Sprache aufgezählt und der Funktion erläutert.

- **Start:** Dieses Element kennzeichnet den Start eines Szenarios.
- **End:** Analog zum Start wird das End-Element zum Beenden eines Szenarios verwendet.

- **Paralleles Gateway:** Bei diesem Gateway-Typ wird das Szenario in zwei parallele Stränge aufgeteilt und parallel ausgeführt.
- **Exklusives Gateway:** Im Gegensatz zum Parallelen Gateway wird bei diesem Gateway-Typ, anhand einer Variable, die Entscheidung getroffen welcher Pfad genutzt wird.
- **Generischer Task:** In der Entwicklungsphase war es Notwendig einen Task zu haben, bei dem der Befehl in der Entwicklungsumgebung festgelegt werden konnte. Dies wird durch den Generischen Task ermöglicht. Hierbei ist die Kenntnis der Syntax der Befehle notwendig.
- **Spezifischer Task:** Die spezifischen Tasks beinhalten einen vorgegebenen Befehl oder Funktion. Und sollen vorrangig von den Domain Spezialisten benutzt werden.

4.2.3 Ausblick

In diesem Abschnitt werden Themen bearbeitet, die für zukünftige Arbeiten interessant sein könnten. Zunächst sollte die Sprache überprüft werden, dabei ist zu beachten, dass die genannten Punkte, wie **Konformität**, **Orthogonalität**, **Erweiterbarkeit** und **Unterstützung**, besonders betrachtet werden. Des Weiteren sollte die Sprache auch auf Hinblick ihrer Ausdruckskraft überprüft werden.

Ein weiteres Thema kann die Erweiterung der domänenspezifische Sprache um neue Elemente sein. So können sich diese darum kümmern Elemente wie den in Unterabschnitt 3.4.2 genanten Ad-Hoc-Subprozess oder den Timer-Event zu implementieren. Da diese für die Abbildung einiger Szenarien im Living Place notwendig sind. Aber auch die Erweiterung der domänenspezifische Sprache um neue Funktionen könnte ein spannendes Thema für zukünftige Arbeiten sein. So könnten zum Beispiel einige Abläufe zu einem Macro zusammen gefasst werden und diese dem Anwender angeboten werden.

Ein wichtiges Thema, das nicht durch diese Arbeit beleuchtet wurde, sind Sicherheitsaspekte. Dabei können sich zukünftige Arbeiten mit Fragen beschäftigen wie Multi-User zugriff und auch Freigabe von bestimmten Tasks an Benutzergruppen.

4.3 Fazit

Das Ziel dieser Arbeit war die Entwicklung einer domänenspezifischen Sprache. Diese ermöglicht es komplexe Abläufe von vernetzten Smart-Objekts darzustellen und auch zu erstellen.

Dabei wurde das Living Place und dessen eingesetztes Multiagentensystem, die Middleware, als Basissystem genommen und hierfür eine Lösung entwickelt.

Mit der entwickelten Lösung lassen sich einfache Szenarien (vgl. Abbildung 4.3) abbilden, die auch für Personen mit wenig IT-Kenntnisse verständlich sind. Bei dem entworfenen Szenario handelt es sich um ein vereinfachtes Modell der in Abschnitt 2.5 beschriebenen Szenarien.

Dabei wird ein Lüften der Wohnung initialisiert und anhand der übergebenen ProcessVariable entschieden ob gelüftet werden soll oder nicht. Sollte die Wohnung gelüftet werden, so werden 2 parallele Stränge ausgeführt. In dem einen wird die Heizung deaktiviert und in dem anderen werden die Fenster geöffnet. Beim beenden des Lüftens werden nochmal 2 parallele Stränge ausgeführt, jene schließen die Fenster und schalten die Heizung wieder ein. Das beenden des Szenarios wird mit dem End-Event eingeleitet.

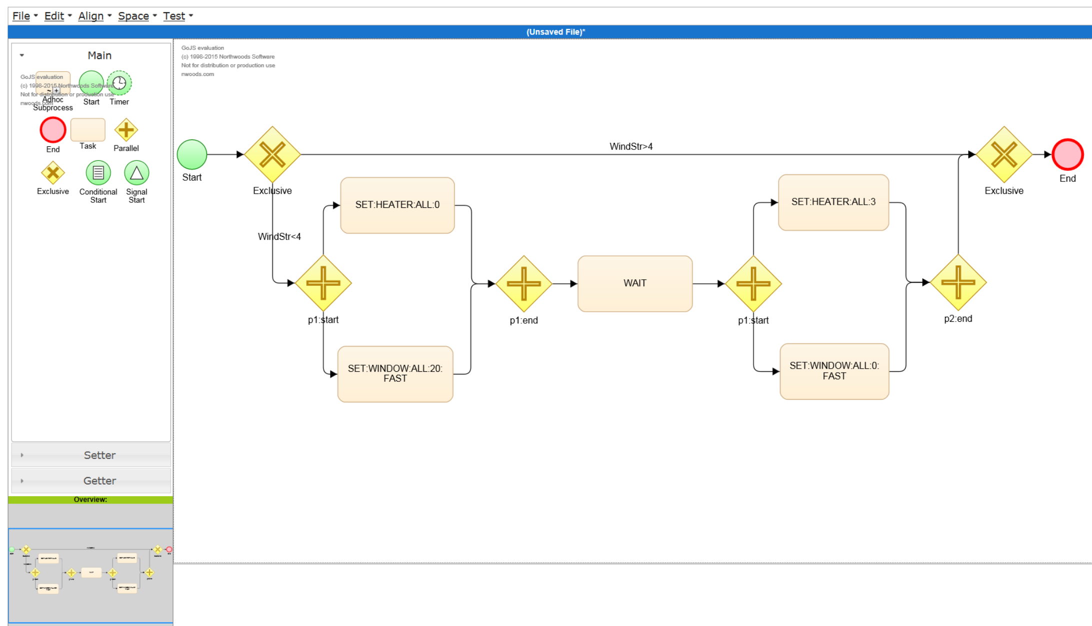

Abbildung 4.3: Simples Szenario erstellt mit der Entwicklungsumgebung aus Unterabschnitt 4.1.2

Das abgebildete Szenario (vgl. Abbildung 4.3) konnte von der Prozess-Engine eingelesen und ausgeführt werden. Die dabei versendeten Kommandos, von der Prozess-Engine an das

Akka-Framework, konnten aus der Console ausgelesen werden (vgl. Abbildung 4.4). Diese sind in den Zeilen 8, 11, 15 und 18 zu sehen.

```
1  [TestNG] Running:
2    C:\Users\LGMTR\AppData\Local\Temp\testng-eclipse--501593080\testng-customsuite.xml
3
4  Start des Prozesses: test_files/eva_proc.json
5  ...
6  [INFO] [01/31/2016 15:17:13.564] [ClusterSystem-akka.actor.default-dispatcher-20]
7      [akka://ClusterSystem/user/WINDOW] Publish Channel: WINDOW, got:
8          ControlWindowGroup(ALL, 20, FAST);
9  [INFO] [01/31/2016 15:17:13.806] [ClusterSystem-akka.actor.default-dispatcher-21]
10     [akka://ClusterSystem/user/HEATER] Publish Channel: HEATER, got:
11         ControlHeaterGroup(ALL, 0);
12 ...
13 [INFO] [01/31/2016 15:17:14.798] [ClusterSystem-akka.actor.default-dispatcher-4]
14     [akka://ClusterSystem/user/WINDOW] Publish Channel: WINDOW, got:
15         ControlWindowGroup(ALL, 0, FAST);
16 [INFO] [01/31/2016 15:17:15.488] [ClusterSystem-akka.actor.default-dispatcher-4]
17     [akka://ClusterSystem/user/HEATER] Publish Channel: HEATER, got:
18         ControlHeaterGroup(ALL, 3);
19 PASSED: processBeanTestEvaProcessComplex
20
21 ===============================================
22     Default test
23     Tests run: 1, Failures: 0, Skips: 0
24 ===============================================
25
26
27 ===============================================
28 Default suite
29 Total tests run: 1, Failures: 0, Skips: 0
30 ===============================================
31
32 [TestNG] Time taken by org.testng.reporters.jq.Main@10a035a0: 41 ms
33 [TestNG] Time taken by [FailedReporter passed=0 failed=0 skipped=0]: 0 ms
34 [TestNG] Time taken by org.testng.reporters.JUnitReportReporter@5204062d: 0 ms
35 [TestNG] Time taken by org.testng.reporters.SuiteHTMLReporter@2eda0940: 50 ms
36 [TestNG] Time taken by org.testng.reporters.XMLReporter@704921a5: 0 ms
37 [TestNG] Time taken by org.testng.reporters.EmailableReporter2@27d415d9: 0 ms
38
```

Abbildung 4.4: Ausgabe des ausgeführten Szenarios

Ausblick

Zukünftige Arbeiten könnten sich mit der Fragestellung beschäftigen, ob eine Portierung auf andere Smart Environments möglich oder ob diese sinnvoll ist. Für die Beantwortung dieser Fragen stehen die Ergebnisse dieser Arbeit für zukünftige Projekte zur Verfügung. Dabei

können sich andere Arbeiten an dieser Orientieren.

Bei der Portierung ist auch zu untersuchen in wie weit das entwickelte System die Performance beeinflusst. Diese Frage kann auch anhand des Entwurfs und der Middleware untersucht werden. Hierbei können auch Fragen auftauchen die die Notwendigkeit solch einer domänenspezifische Sprache in Frage stellen oder auch diese unter dem Aspekt der IT-Sicherheit beleuchten können.

Eine weitere grundlegende Frage die zu beantworten wäre, ist das bereitstellen der Software für Anwender mit keinen oder geringen IT-Kenntnissen. So können sich zukünftige Arbeiten mit Usability-Tests, in Hinblick auf Installateure oder Bewohner eines Smart-Environments, beschäftigen. Daraus können sich weitere Anforderungen ergeben, die eine Anpassung der der domänenspezifischen Sprache erfordert. Im Moment kann diese Frage jedoch nicht beantwortet werden, da jede weitere Generation, in Hinblick auf deren IT-Kenntnisse, mehr Erfahrung mit Smart-Objekts hat.

5 Fazit

5.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde eine domänenspezifische Sprache entwickelt. Diese sollte die Domäne Living Place abdecken. Um dies zu erreichen wurden zunächst die möglichen Szenarien und die Entitäten im Living Place analysiert. Die Erkenntnisse aus dieser Analyse flossen in den Entwurf der domänenspezifische Sprache ein.

Des Weiteren wurde ein Werkzeug entworfen. Jenes konnte die domänenspezifische Sprache abbilden. Dieser Entwurf floss in die Entwicklung des Minimal-Entwurfs und bildete somit die Anforderungen an den Prototypen. Dieser Prototyp sollte die grundlegenden Funktionen enthalten und die Machbarkeit der Vision darstellen. Die Architektur des Minimal-Entwurfs besteht aus der Entwicklungsoberfläche, der Prozess-Engine und dem Test Environment.

Die Entwicklungsoberfläche wurde mit Hilfe des *go.js*-Frameworks erstellt. Jenes ermöglicht die Darstellung und auch die Erstellung von Szenarien. Die Entwicklungsoberfläche ist auch die Basis der Test-Umgebung. Aus der Entwicklungsoberfläche ist es möglich Szenarien testweise zu starten und die versendeten Kommandos auszulesen.

Die Evaluierung in Kapitel 4 beleuchtete zum einen die Implementierung der Entwicklungsumgebung und zum anderen die domänenspezifische Sprache. Hierbei wurden nicht nur die implementierten Komponente erläutert, sondern auch die Kernpunkte der domänenspezifische Sprache, die im Analyse-Teil festgelegt wurden. Des Weiteren wurde ein simples Szenario erstellt und die Ausführung von jenem demonstriert. Im Anschluss daran wurden einige Anregungen für zukünftige Arbeiten gegeben.

5.2 Ausblick

Aufgrund der stetigen Entwicklung von Smart-Environments könnten sich zusätzliche Anforderungen an die domänenspezifische Sprache ergeben. Jene könnten eine Erweiterung oder Anpassung dieser erfordern. Hierbei können nachfolgende Arbeiten diese erweitern oder auch

verbessern. Eine mögliche Ergänzung könnte die Speicherung des Zustandes eines Szenarios in einer Datenbank sein. Dies könnte es ermöglichen auch Szenarien zu erstellen die über einen sehr langen Zeitraum laufen und dabei möglichst wenig Ressourcen verbrauchen.

Da mit der Entwicklung einer DSL der Versuch unternommen wird Abläufe oder Szenarien zu generalisieren kann es bei heutiger stetiger Entwicklung von Smart-Environments auch zu der Frage führen ob eine domänenspezifische Sprache zur Abbildung dieser sinnvoll ist? Jene angestrebte Generalisierbarkeit von Abläufen kann die Sprache zu starr machen und somit uninteressant für Entwickler und Benutzer werden.

Eine weitere spannende Fragestellung könnte das Abbilden mehrerer Smart-Environments in einem Szenario sein. Dies würde der domänenspezifische Sprache ermöglichen auch das Ïnternet der Dingeäbzubilden und zu steuern. Welches als *Ubiguitous Computing* von Weiser (1999) schon in den neunziger Jahren beschrieben wurde.

Literaturverzeichnis

[Eichler 2014] Eichler, Tobias: *Agentenbasierte Middleware zur Entwicklerunterstützung in einem Smart-Home-Labor*, Hamburg University of Applied Sciences, Diplomarbeit, 2014. – URL http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/master/eichler.pdf

[Fowler 2010] Fowler, Martin: *Domain-Specific Languages* -. 1. Aufl. Amsterdam : Pearson Education, 2010. – ISBN 978-0-131-39280-9

[Inc 2015] Inc, Typesafe: *Akka Scala Documentation*. November 2015. – URL http://doc.akka.io/docs/akka/2.4.1/AkkaScala.pdf

[inHause] inHause: *inHause Frauenhofer Institut - Smart Home*. – URL http://www.inhaus.fraunhofer.de/content/dam/inhaus/de/documents/DokumentezumDownload/PP_Standardpr%C3%A4sentation_Fraunhofer-inHaus-Zentrum.pdf

[JSON] JSON: *Einführung in JSON*. – URL http://www.json.org/json-de.html

[von Luck u. a. 2010] Luck, Prof. Dr. K. von ; Klemke, Prof. Dr. G. ; Gregor, Sebastian ; Rahimi, Mohammad A. ; Vogt, Matthias: Living Place Hamburg - A place for concepts of IT based modern living / Hamburg University of Applied Sciences. URL http://livingplace.informatik.haw-hamburg.de/content/LivingPlaceHamburg_en.pdf, Mai 2010. – Forschungsbericht

[Odersky u. a. 2010] Odersky, Martin ; Spoon, Lex ; Venners, Bill: *Programming in Scala*. 2. Aufl. CALIFORNIA : ARTIMA PRESS, 2010. – ISBN 978-0-981-53164-9

[OMG 2011] OMG: *Business Process Model and Notation (BPMN)*. January 2011. – URL http://www.omg.org/spec/BPMN/2.0/PDF

[Oracle] ORACLE: *JavaServer Faces Technology - Documentation.* – URL http://www.oracle.com/technetwork/java/javaee/documentation/index-137726.html

[RWE] RWE: *RWE - Smart Home.* – URL http://www.rwe-smarthome.de/web/cms/de/2852396/home/wie-funktioniert-smarthome/

[Strese u. a. 2010] STRESE, Hartmut ; SEIDEL, Uwe ; KNAPE, Thorsten ; BOTTHOF, Alfons: Smart Home in Deutschland / Institut für Innovation und Technik (iit). Berlin, DE, Mai 2010. – Forschungsbericht. – URL http://www.solarmobil-deutschland.de/bmbf-aal/Publikationen/studien/extern/Documents/iit-studie-smart-home.pdf

[Telokom] TELOKOM: *Telekom - Smart Home.* – URL http://www.smarthome.de/

[Voelter 2009] VOELTER, Markus: *Werkzeuge zur Erstellung und Verarbeitung von DSLs.* March 2009. – URL http://www.voelter.de/data/articles/ToolsFuerDSLs.pdf

[Weiser 1999] WEISER, Mark: The Computer for the 21st Century. In: *SIGMOBILE Mob. Comput. Commun. Rev.* 3 (1999), July, Nr. 3, S. 3–11. – URL http://doi.acm.org/10.1145/329124.329126. – ISSN 1559-1662

[Wooldridge 2002] WOOLDRIDGE, Michael: Intelligent Agents: The Key Concepts. In: *Proceedings of the 9th ECCAI-ACAI/EASSS 2001, AEMAS 2001, HoloMAS 2001 on Multi-Agent-Systems and Applications II-Selected Revised Papers.* London, UK, UK : Springer-Verlag, 2002, S. 3–43. – URL http://dl.acm.org/citation.cfm?id=645699.665762. – ISBN 3-540-43377-5

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 1. Februar 2016 Johann Bronsch